Object Interconnections

Object Adapters: Concepts and Terminology (Column 11)

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

Steve Vinoski
vinoski@iona.com
IONA Technologies, Inc.
60 Aberdeen Ave., Cambridge, MA 02138

This column will appear in the November/December 1997 issue of the SIGS C++ Report magazine.

1 Introduction

In this column, we'll start presenting issues surrounding CORBA *Object Adapters* (OAs). We'll focus on what Object Adapters are and describe their roles within a CORBA-based system. In addition, we'll begin an in-depth discussion of the new Portable Object Adapter (POA) specification that was recently adopted by the OMG. Our subsequent columns will continue this discussion and show some examples using C++.

2 Terminology

Before describing what Object Adapters are, we first need to define some key terms. Even if you're already familiar with CORBA, you should pay close attention to these terms and their definitions since some of them have been introduced recently with the POA specification.

CORBA object: A "virtual" entity capable of being located by an ORB and having client requests delivered to it. A CORBA object is identified, located, and addressed by its object reference. Within the context of a request invocation, the CORBA object to which the request is sent is called the "target object."

Servant: A programming language entity that exists in the context of a server and implements a CORBA object. In non-OO languages like C and COBOL, a servant is implemented as a collection of functions that manipulate data (*e.g.*, an instance of a struct or record) that represent the state of a CORBA object. In OO languages like C++ and Java, servants are object instances of a particular class.

It is extremely important to understand the relationship between a CORBA object and a servant. A CORBA object is considered "virtual" because it never exists on its own. Therefore, to perform client requests, it must have a servant that reifies its existence and fulfills those requests. Note, however, that CORBA objects may have state, but servants may not necessarily have state. For example, the state of a CORBA object can be kept in a database. In this case, the servant is merely used as a way to examine and modify that state, but the servant need not maintain any state of its own.

The relationship between a CORBA object and a servant is very much like the relationship between virtual memory and physical memory in an operating system. Just as virtual address space does not actually exist, neither does a CORBA object. A virtual memory location can be read and written by a computer program because of the work performed by the computer's memory management unit (MMU). The MMU maps virtual memory addresses into physical memory addresses and ensures that each valid virtual memory address is mapped to a physical memory storage location. Similarly, the ORB and the OA cooperate to allow client applications to invoke requests on CORBA objects and ensure that each valid CORBA object is mapped to a servant. In addition, the ORB and the OA cooperate to transparently locate and invoke the proper servants given the addressing information stored in CORBA object references.

Skeleton: A programming language entity that connects a servant to an OA, allowing the OA to dispatch requests to the servant. In C, a skeleton is a collection of pointers to servant-specific functions. In C++, a skeleton is a base class from which the servant class derives. For static request invocation, skeletons are typically generated automatically by an IDL compiler. In addition, CORBA supports the Dynamic Skeleton Interface (DSI), which enables a server to handle requests for objects that have no static skeletons available.

Object Id: A user- or system-specified identifier used to "name" an object within the scope of its OA. Object Ids are *not* guaranteed to be globally unique, nor are they necessarily unique within a single server process. The only constraint is that each is unique within the OA where it is created or registered.

Activation: The act of starting an existing CORBA object to allow it to service requests. Since servants ultimately perform client requests, activation requires that the CORBA object be associated with a suitable servant. Note that activation does not imply CORBA object creation since a CORBA object can't be activated if it does not exist already. Activation may, however, cause the creation of the servant.

This discussion of activation may beg the question of how CORBA objects are created. Unlike C++ classes, OMG IDL interfaces do not have a notion of constructors or any other special object creation functions. From a client perspective, CORBA objects are often created by invoking normal CORBA operations on *factory objects*. From the server perspective, the servants for factory objects are implemented such that they invoke operations on the OA in order to associate servants with the CORBA objects they create, as well as to create object references for those new objects. In addition, factory operations activate the new CORBA object as well.

Deactivation: The act of shutting down an active CORBA object. Obviously, deactivation is the opposite of activation. Deactivation requires the destruction of the association between a CORBA object and its servant. Note that deactivation does not imply CORBA object destruction. After deactivation, a CORBA object can be reactivated to receive more requests. Deactivation may, however, result in the destruction of the servant.

Incarnation: The dictionary defines this word as “the act of giving bodily form or substance to.” In the general context of CORBA objects and servants, incarnation is the act of associating a servant with a CORBA object so that it may service requests. In other words, incarnation provides a servant “body” for the virtual CORBA object.

Etherealization: The opposite of incarnation, this is the act of destroying the association between a CORBA object and its servant. Etherealization takes away the “body” that was associated with the CORBA object at the point of incarnation. After etherealization, the CORBA object still exists as a virtual entity, but it has no associated body to carry out requests for it.

Active Object Map: A table maintained by an object adapter that maps its active CORBA objects to their associated servants. Active CORBA objects are named in the map via Object Ids. Note that the term “active object” is something of a misnomer since the servants in this table need not implement the Active Object pattern [1].

Much of the terminology introduced here is concerned with the lifetimes of entities in a CORBA system. Perhaps the easiest way to remember the distinction between servants and CORBA objects is that a single CORBA object may be represented by one or more servants over its lifetime. Likewise, a servant may represent one or more CORBA objects simultaneously. *Activation* and *deactivation* refer only to CORBA objects, while the terms *incarnation* and *etherealization* refer to servants. In fact, the latter two terms were introduced to allow servant lifetimes to be discussed separately from the lifetimes of CORBA objects.[1]

Figure 1 shows the lifecycle of a CORBA object with respect to the lifecycle of the servant(s) used to incarnate it. A CORBA object is first created and then is activated, either immediately or later when a request is received for it. When activated, it can be immediately incarnated by a servant, or during its activation period multiple servants may be incarnated and etherealized for it for each request is receives. (The appearance of three dots in the figure denote “one or more occurrences of.”) The CORBA object is then deactivated. Later, it can be reactivated, at which point the servant incarnation and etherealization cycle begin again. Finally, the CORBA object is destroyed and its lifecycle is complete.

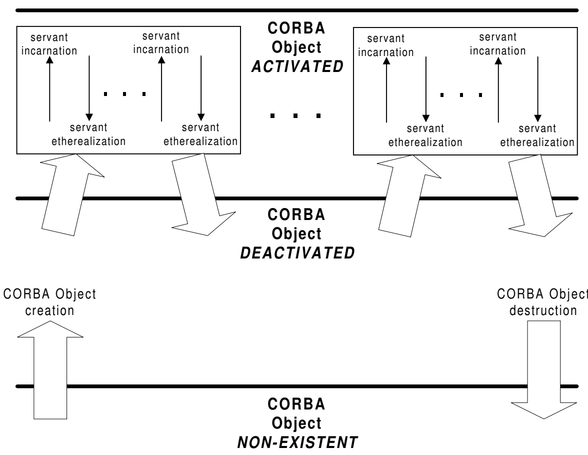

Figure 1: Request Lifecycle in the Portable Object Adapter

Failure to recognize the difference between the lifetimes of CORBA objects and servants often causes confusion for CORBA developers. This is because it is easy to think that the lifetime of the C++ servant is also the lifetime of the CORBA object it represents. However, this is not always the case. The consequence of failing to understand this distinction can be applications that unnecessarily create new CORBA objects rather than simply reactivating existing CORBA objects and incarnating them with new servants.

3 An Overview of CORBA Object Adapters

3.1 Motivation for Object Adapters

As its name implies, a CORBA Object Adapter is a reification of the Adapter pattern [2] since it adapts the programming language concept of servants to the CORBA concept of objects. This adaptation role is clearly shown in Figure 2, where the OA fits directly between the ORB Core and the static and dynamic Skeletons.

In their role as adapters, different OAs can support dif-

[1] There’s no truth to the rumor that the latter terms were coined by Zen buddhists ;-).

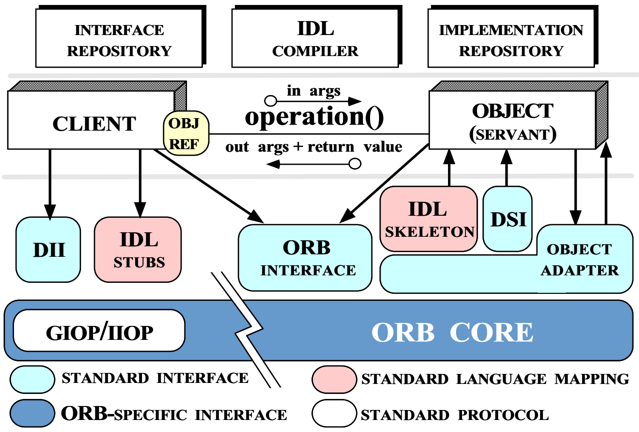

Figure 2: Components in the CORBA Reference Model

ferent servant implementation styles. For example, the Orbix *Object Database Adapter Framework* [3] allows objects implemented using an object database (such as ODI's ObjectStore) to be used as servants for CORBA objects. Another example is the real-time Object Adapter provided by TAO [4].

It is certainly possible to define a distributed object computing middleware architecture where servants are registered directly with the ORB Core rather than with an OA. However, such an architecture would have one of the following limitations:

Large footprint: One approach would require the ORB Core to support multiple interfaces and services to support multiple servant implementation styles. However, this approach would make the ORB Core larger, slower, and more complicated for all applications. Therefore, this approach would violate the "principle of parsimony" by forcing applications to incur the cost of features they may not use.

Lack of flexibility: An alternative would only be to support one form of servant implementation. For example, all servants might be required to inherit from one base class. However, this approach would limit the ORB's utility as an integration mechanism capable of dealing with widely-varying programming languages, legacy systems, and software design styles.

Therefore, using OAs means that support for different servant implementation styles can be isolated into different OAs. This results in a smaller and simpler ORB Core.

3.2 Object Adapter Functionality

CORBA Object Adapters provide the following functionality:

Request demultiplexing: OAs demultiplex each CORBA request to the appropriate servant. When the ORB Core receives a request, it cooperates with the OA through a private (*i.e.*, non-standard) interface to ensure that the request reaches the proper servant.

Operation dispatching: Once the OA locates the target servant, it dispatches the requested operation. At this point, the skeleton is used to transform the parameters in the request into the arguments passed to the intended servant operation.

Activation and deactivation: OAs can activate CORBA objects. In the process, they can also incarnate servants to handle requests for those objects. Similarly, OAs can deactivate objects and can etherealize their corresponding servants if they're no longer needed.

Generating object references: An OA is responsible for generating object references for the CORBA objects registered with it. Object references normally identify a CORBA object and contain addressing information on how to reach that object in the distributed system. This means that OAs must ultimately cooperate with the communication facilities built into the ORB and the underlying operating systems to ensure that the information necessary for reaching the object is present in the object reference. For instance, an Interoperable Object Reference (IOR) supporting the Internet Inter-ORB Protocol (IIOP) transport will contain the Internet address of the server host, as well as the port number where the server process is listening.

OAs are intimately involved in the dispatching of requests to object operations. Therefore, they must be carefully designed so that they do not become a bottleneck in the request dispatch path. Conventional ORBs demultiplex client requests to the appropriate operation of the servant using the following steps shown in Figure 3.

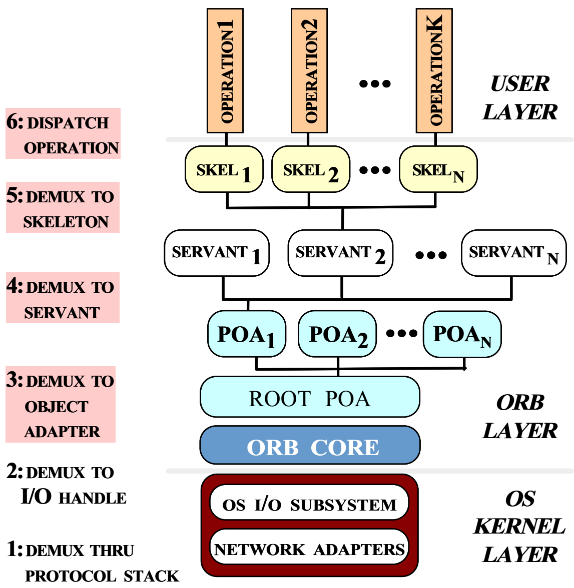

Figure 3: Layered CORBA Request Demultiplexing

Steps 1 and 2: The OS protocol stack demultiplexes the incoming client request multiple times, *e.g.*, from the network interface card, through the data link, network, and transport layers up to the user/kernel boundary (*e.g.*, the socket) and then dispatches the data to the ORB Core.

Steps 3, and 4: The ORB Core uses the addressing information in the client's object key to locate the appropriate POA and servant. POAs can be organized hierarchically. Therefore, locating the POA that contains the servant can involve multiple demultiplexing steps through the hierarchy.

Step 5 and 6: The POA uses the operation name to find the appropriate IDL skeleton, which demarshals the request buffer into operation parameters and performs the upcall to code supplied by servant developers.

Demultiplexing client requests through all these layers is expensive, particularly when a large number of operations appear in an IDL interface and/or a large number of servants are managed by an Object Adapter. [5] measures the performance of several OA demultiplexing and dispatching optimization strategies. In general, demultiplexing techniques based on perfect hashing or active demultiplexing are substantially faster and more predictable than those based on linear search or dynamic hashing.

4 Standard Object Adapters

Over the life of the CORBA specification, only two OAs have ever been officially adopted by the OMG. The first was the *Basic Object Adapter* (BOA), which was specified in the original CORBA specification. More recently, the OMG adopted a new *Portable Object Adapter* (POA). An understanding of the BOA is not needed to understand and use the POA. However, examining the features and the history of the BOA can help us understand the design and capabilities of the POA.

4.1 CORBA Basic Object Adapter (BOA)

4.1.1 Basic Features

The first version of the CORBA specification contained a description of the *Basic Object Adapter* (BOA). The BOA was intended to be a multi-purpose OA that could support various styles of servants. Because of the purported flexibility of the BOA, the original CORBA architects thought that, in total, there would only be a few OAs (*e.g.*, a library OA, a database OA, etc), with most applications using the BOA.

The BOA supported four different *activation models*, described below. Note that these models refer to the activation of *server processes*, not CORBA objects.

Unshared server: which is a server that supports only a single CORBA object. An unshared server typically deals with an object of just one type. Whenever a new CORBA object of that type is created, the ORB automatically starts a new server process to contain it.

Shared server: which is a server that supports multiple CORBA objects, often of different types. For example, such a server can support CORBA objects that play the role of factories (such as the `Quoter_Factory` we defined in [6]), as well as the CORBA objects of the type created by such factories. In practice, most existing CORBA servers are shared servers.

Persistent server: which is a server that is not automatically started by the ORB. These types of servers might be started by a script that runs whenever the system boots, for example. Note that the use of the term "persistent" is an unfortunate misnomer since it implies that the state of the servants in the server will persist automatically across crashes and shutdowns. Since this is not the case, a better term would have been "manually launched" server. In addition, note that the mechanisms used to launch a server process are completely orthogonal to whether a server is a shared server or an unshared server, so this particular activation model doesn't quite belong with the others.

Server-per-operation: this type of "server" is not just a single process. Instead, it's a collection of processes, each implementing a given operation of a particular type of CORBA object. This type of server is theoretically useful for servants that are implemented using shell scripts. For example, different shell scripts implement different operations of the CORBA object. Because of the difficulty of supporting this server style, and because of its limited utility, few commercial ORBs support server-per-operation.

4.1.2 Evaluation of the BOA

Unfortunately, the BOA from the original CORBA specification is woefully incomplete. The following are some of the key problems:

Not defining a portable way to associate skeletons with servants.: The BOA specification does not describe what skeletons looks like, nor how servants are associated with them. As a result, the OMG IDL C++ Mapping Specification originally only defined how the bodies of servant methods and the method signatures should appear. It could not specify the names of base classes that servants had to derive from. Naturally, this omission made writing portable CORBA C++ code very difficult.

Failing to describe how servants are registered: Implementations of the BOA typically allow servants to be registered *implicitly* (e.g., in the constructor of a servant when it is incarnated) and/or *explicitly* (*e.g.*, by calling a method on the BOA). The original BOA specification doesn't define these APIs, however. Therefore, each ORB tends to implement this functionality differently.

Completely ignoring the issues of multi-threading a server process: Multi-threaded ORBs are important since they allow long-running tasks to execute simultaneously without impeding the progress of other tasks. However, the CORBA 2.0 BOA specification does not address the issue of multi-threading, leaving the decision to ORB developers. As a result, different ORBs tend to implement multi-threading differently.

Failing to accurately define the functions required to make a server listen for requests: The BOA supplies two operations that appear to make the server start processing requests: `impl_is_ready` and `obj_is_ready`. The BOA specification associates these operations with its activation models, stating that one or the other should be called depending upon the activation model chosen for the server application. Unfortunately, this part of the specification is extremely vague and unclear, resulting in different ORB products using the two operations in very different ways.

Due to the limitations with the BOA specification, each ORB vendor that implemented the BOA did it their own (inconsistent) ways. This, in turn, resulted in almost no portability of server-side code between different ORB products. Therefore, the OMG issued an RFP for portability enhancement [7] that called for either repairing the problems with the BOA[2] or replacing it with a new OA.

4.2 Portable Object Adapter (POA)

In March 1997, the Portable Object Adapter (POA) [8] was submitted to the OMG as a replacement for the BOA. The POA finally provides portability for CORBA server applications. It also provides new and very useful functionality.

One of the enhancements specified in the Portability Joint Submission document is the deprecation of the BOA. Thus, once the submission has been fully integrated with the CORBA specification (which should occur by early 1998) the OMG will fully remove the BOA from CORBA. Of course, ORB vendors that currently support the BOA will be allowed to continue to do so for as long as they wish for the sake of their existing CORBA customers.

4.3 Overview of POA Features

The newly defined POA specification supports a wide range of features, including: *user- or system-supplied object identifiers*, *persistent and transient objects*, *explicit and on-demand activation*, *multiple servant → CORBA object mappings*, *total application control over object behavior and existence*, and *static and DSI servants*. Each of these features is outlined below. Subsequent columns will illustrate these and other POA features in detail using C++.

4.3.1 Object Identifier

An Object Identifier (`ObjectId`) is a `sequence` of `octets` used to identify a CORBA object within the context of a POA. An `ObjectId` can be assigned by the application or by the POA. The `ObjectId` is a "demultiplexing key" used to associate client requests with CORBA objects.

It is important to recognize that `ObjectIds` do not necessarily identify the object beyond the context of its POA. Thus, they are not guaranteed to be unique global identifiers. CORBA objects are identified only by their object references, of which `ObjectIds` are potentially only one part.[3] Also, since object references are opaque to applications, clients are unable, and in fact have no need, to "reach into them" to examine any `ObjectId` portion. Clients merely use object references to denote the targets of their requests; the ORB takes care of the rest.

In IORs, `ObjectIds` make up only a part of the overall *object key* used by the ORB to locate the CORBA object. The CORBA General Inter-ORB Interoperability Protocol (GIOP)[4] defines an object key as a portion of the IOR used to identify the CORBA object at the communication endpoints given in the IOR. For example, if a request for the CORBA object is sent over IIOP to the host and port given in the object's IOR, the entity receiving the request on that port can use the object key to uniquely identify the target object. This implies that the object key is unique only with respect to the communication endpoint.

The object key is more than just the `ObjectId`, however. For instance, it might also contain an indication of which POA in the server process is expected to dispatch the request to the target object. In general, GIOP does not define the contents of the object key, so ORB implementations put whatever information they need in the object key so they can locate the target object at the given endpoints.

It's important to note that this flexibility in the GIOP specification does not cause any interoperability problems between ORBs because it is assumed that the ORB that created the IOR is the same ORB that's listening at the specified communication endpoints, and is thus capable of decoding the object key because it created it. However, it is the case that an object created by one ORB can't be reactivated in another vendor's ORB since the new ORB won't necessarily be able to decode the object key created by the original ORB.

4.3.2 Persistent vs. Transient Objects

Persistent objects are CORBA objects whose lifetimes are independent of the lifetimes of any server processes in which they are activated. In contrast, transient objects are CORBA objects whose lifetimes are bounded by the lifetimes of the server processes in which they're created. When server processes die, their transient objects die with them.

Persistent objects are "normal" CORBA objects that can be activated when necessary to fulfill a client request. Transient objects, on the other hand, typically involve less overhead than persistent objects because the ORB and POA need

[2] For a comprehensive list of problems with the BOA, please see the cited RFP document. It devotes seven pages to listing and describing problems with the BOA.

[3] Note that the POA specification does not *require* `ObjectIds` to appear within object references, but most POA implementations will very likely put them there.

[4] IIOP is the specification for how GIOP is used to deliver requests and responses over TCP/IP.

not keep track of activation information for them. They're useful for "temporary" objects such as certain kinds of callback objects [9].

The use of the term "persistent" for CORBA objects that outlive any single server process is potentially confusing since the term is used differently in several other contexts in CORBA and in the database community. In the context of the POA, a persistent object refers mainly to the fact that it "persists" across server process activations. However, one might expect that the use of the term also implies that object state is kept in persistent storage between activations. While persistent CORBA objects generally do have persistent state, the POA itself has no persistent state, nor does it provide any functions to directly help objects save and load their persistent state. In addition, note that persistent objects do not imply the use of the ill-named BOA "persistent" activation model, though it is certainly possible to have persistent objects within a persistently (*i.e.*, manually) launched server.

4.3.3 Activation

The activation modes supported by the BOA were largely centered around server processes. In contrast, the activation facilities supplied by the POA are focused solely on CORBA object activation and servant incarnation. The POA supports the following activation styles:

Explicit activation: the server application programmer registers servants for CORBA objects using direct calls on a POA. This is useful for server applications that have just a few CORBA objects.

On-demand activation: the server application programmer registers a servant manager that the POA upcalls when it receives a request for a CORBA object that is not yet activated. When it receives such an upcall, the servant manager typically performs one of the following operations:

1. It incarnates the servant if necessary and registers it with the POA, which then dispatches the request to that servant.
2. It raises a `ForwardRequest` exception (defined in the `PortableServer` module) to send the request to another object. This exception has an object reference data member that denotes the object to which the request should be redirected. This feature could be used to develop a server that performs application-specific location or load balancing, for example.
3. It raises a `CORBA::OBJECT_NOT_EXIST` exception to indicate that the CORBA object has been destroyed.

Implicit activation: an action on a servant results in activation without any explicit calls on a POA. In the IDL $\rightarrow$ C++ mapping, implicit activation of a servant on a POA created with the proper policies can be achieved by invoking the servant's `_this` method.

Default servant: the application registers a default servant that is used if a request arrives for a CORBA object that is not yet activated, and there are no servant managers registered. This feature is very useful for DSI-based server applications since it allows a single DSI servant to incarnate all CORBA objects by default without requiring the intervention of a servant manager.

4.3.4 Object ID Uniqueness

A POA may require that each CORBA object be activated with a unique servant. This is a very straightforward implementation technique because it provides a one-to-one correspondence between CORBA objects and their servants. An example where this technique is useful is for transient CORBA objects whose state data are kept directly in servant data members. While this technique is conceptually straightforward, it does not scale well when the server contains a large number of CORBA objects since the server must maintain separate servants for every CORBA object it implements.

Alternatively, a POA may allow one servant object to incarnate multiple CORBA objects. Allowing a single servant to incarnate multiple CORBA objects is very useful for minimizing server resource usage. For example, consider a key/value database where each entry in the database is treated as a separate CORBA object, with each object having one key as its object identifier. A useful way to implement a server for the database is to utilize a single C++ servant that determines which database entry is the target of the request by examining the `ObjectId` accompanying the request. This approach eliminates the need to have a separate C++ servant for each database entry. Therefore, it scales well regardless of the number of database entries.

Such an application could also use on-demand activation to avoid the need to register all the database entries as CORBA objects every time the server started up. If only a few entries are ever the targets of CORBA requests, this activation approach eliminates unnecessary server registration overhead. In addition, it reduces the amount of resources the POA needs to track servant registrations.

4.3.5 Object Behavior and Existence

Applications control object state and behavior completely. A POA has no persistent state; applications are responsible for any required persistence of object state. Applications determine object existence and can throw a `CORBA::OBJECT_NOT_EXIST` exception to indicate that a CORBA object no longer exists. Likewise, they can throw `PortableServer::ForwardRequest` exceptions to tell the ORB to look elsewhere for the object. Such forwarding is completely transparent to the invoking client application.

POAs do nothing to keep track of the persistent states of CORBA objects because there are so many possible ways to implement such persistence. If the POA specification mandated one or even a few possible approaches to persistence, it would unduly limit other classes of applications for which those approaches proved unsuitable. Moreover, object persistence is better provided by a higher-level service. The existing Object Persistence service is largely considered to be too complicated[10], so at this time the OMG is working on a replacement for that service.

Allowing applications to raise `CORBA::OBJECT_NOT_EXIST` exceptions gives them complete control over object existence if they so desire. Consider once again the database example described above. If a database entry is removed, the application's servant can simply raise the `CORBA::OBJECT_NOT_EXIST` exception if it receives a request for that entry. This approach is relatively simple due to the fact that the POA keeps no persistent state. If it had to keep track of objects that had already been destroyed, the POA would require per-object persistent state, even for CORBA objects that had already been destroyed (*i.e.*, an "object graveyard"), and application scalability would suffer.

4.3.6 Static and DSI Servants

Not surprisingly, servants based on static skeletons are supported. Traditionally, most CORBA C++ server applications have been based on the static approach. The popularity of static skeletons stem mainly from its efficiency, its familiarity for C++ programmers, and because the DSI was not available in CORBA until version 2.0.

Two types of static servants are supported in the new POA specification:

Inheritance-based servants: Inheritance-based servants are an example of the "class" form of the Adapter pattern and are largely unchanged from the original OMG IDL C++ Mapping Specification. In this approach, the servant class derives from an abstract skeleton base class.

One important difference from the original IDL C++ mapping is that the method of naming skeleton classes has changed. For instance, for an interface `B` defined in module `A`, the fully-scoped name of the skeleton is `POA_A::B`. For an interface `C` defined at global scope, the corresponding skeleton class is named `POA_C`.

The motivation for changing the naming scheme is to create a server-side namespace for POA-based skeletons that is separate from the client-side namespace. It also allows ORB vendors to support legacy BOA applications and POA applications in the same program if they so desire, without having to worry about the BOA and POA skeleton naming conflicts.

Tie-based servants: Tie-based servants are an example of the "object" form of the Adapter pattern and must now be supported by all conforming ORBs. They are actually just like normal servants that are derived from the abstract skeleton base class. Each tie-based servant is an instance of a special auto-generated servant template class. This template class takes a single type parameter (*i.e.*, the application-specific implementation class) and delegates all of its requests to an instance of the template parameter type.

When using ties with the IDL C++ mapping, it is interesting to note that the tie instance is the servant and the C++ class instance that "gets tied into the tie" is called the *tied object*. The tied object typically has no inheritance relationship to any IDL-generated skeleton classes, which makes it easier to integrate legacy code. Tie classes are named in the same fashion as skeletons, except that they also have a `_tie` suffix. For example, for an IDL interface `A`, the corresponding tie class would be named `POA_A_tie`.

For server applications that require the use of the DSI, a standard `DynamicImplementation` abstract base class (defined in the `PortableServer` module) is provided as part of the POA specification. The use of this base class is much like the use of skeletons by inheritance-based servants since DSI servant classes must derive from this class.

4.4 POA Policies

An interesting difference between a POA and a BOA is that a server application can have multiple POAs nested within it. A server application might want to create multiple POAs to support different kinds of CORBA objects and/or different kinds of servant styles. For example, the application might have two POAs, one that supports transient CORBA objects and one that supports persistent CORBA objects.

A nested POA can be created by invoking a factory operation on another POA. All servers have at least one POA called the *Root POA*. To create a POA nested under the Root POA, the application invokes the `create_POA` operation on the Root POA. The object reference for the Root POA is available from the ORB. Note that this implies that applications can't provide their own POA implementations, which is unlikely to be a problem in practice.

The characteristics of each POA other than the Root POA are controlled at POA creation time using different *policies*. The POA specification defines the policies described below.

4.4.1 Threading Policy

A POA can either be single-threaded or have the ORB control its threads. If single-threaded, all requests the POA dispatches to servants will be serialized, either by running them in a single thread or using multiple threads that are synchronized to only execute one request at a time. In contrast, if the ORB-controlled threading policy is specified, the ORB determines the thread or threads that the POA dispatches its requests on.

Given that the BOA specification made no mention of threads, the POA specification, by explicitly supporting threading policies, is much better in this regard. Unfortunately, the specified POA threading policies don't really go far enough. For example, there is no way for an application to specify that a POA should use a Thread Pool [11] or a separate Thread-per-CORBA-object [6], nor are there any hooks similar to Orbix Thread Filters [12] to allow applications to define their own concurrency models. What's worse is the fact that the specification does not even definitively say whether the ORB-controlled model always means multiple threads or not! This leaves the door open for an ORB vendor to define their ORB-controlled threading policy as being single-threaded, which is a potentially serious portability concern.

All in all, while it's great that the specification actually addresses multi-threading, there is not enough threading flexibility supplied by the POA for all applications. Thus, the ORB-controlled threading policy is a new portability problem waiting to happen. Hopefully, the OMG Portability Enhancement Revision Task Force will fix the ORB-controlled model and add enhancements to allow applications to control their threading models in the next revision of the POA specification, which should be ready by early 1998.

4.4.2 Servant Retention Policy

A POA either retains the associations between servants and CORBA objects or it establishes a new CORBA object $\rightarrow$ servant association for each incoming request. Most CORBA C++ applications today use something like the RETAIN value of this policy since they register servants with the OA and expect it to then dispatch the appropriate requests to those servants.

If the NON-RETAIN retention policy is used, however, it allows the application to control the allocation of servants to CORBA objects. This is because each request arriving at a NON-RETAIN POA causes it to invoke the application to obtain the servant from it. The NON-RETAIN policy allows, for example, the application to supply its own servant manager object that intervenes in every single request dispatch to ensure that the target CORBA object still exists.

4.4.3 Request Processing Policy

When a request arrives for a given CORBA object, the POA can either:

Consult its active object map only: it can check to see if it has a servant associated with the ObjectId of the target object. If it does, the request is dispatched. If no such association is found, the POA throws a `CORBA::OBJECT_NOT_EXIST` exception.

Use a default servant: the application can register a *default servant* that is invoked whenever the POA has no servant associated with the ObjectId of the target object in its Active Object Map.

Invoke a servant manager: if a *servant manager* has been registered with the POA, it is invoked by the POA to obtain a servant whenever the POA has no servant associated with the ObjectId of the target object. The servant manager is an application-supplied CORBA object that can incarnate or activate a servant and return it to the POA for continued request processing. Two forms of servant manager are supported: `ServantActivator`, which is used when the POA has the RETAIN policy, and `ServantLocator`, used with the NON_RETAIN policy.

The combination of these policies with the retention policies provide a great deal of flexibility to control servant registration and allocation within the server process.

4.4.4 Implicit Activation Policy

A servant may be activated into a POA *implicitly* if this policy is supported. This is very useful for registering servants for transient CORBA objects. For example, a C++ server can create a servant, and then by setting its POA and invoking its `_this` method, it registers the servant and creates an object reference for the CORBA object in a single operation.

4.4.5 ObjectId Uniqueness Policy

This policy allows the application to control whether a servant can be associated with only a single CORBA object or whether it can handle requests for multiple CORBA objects. Unique servants are fairly common today since most CORBA C++ server applications use a separate C++ object for each CORBA object. However, as described above in the database example, servants that represent multiple CORBA objects can also be very useful since they reduce the server's memory utilization.

4.4.6 Lifespan Policy

This policy allows an application to specify whether the CORBA objects created within a POA are transient or persistent. Note that since policies are set upon the creation of the POA, this is an "all or nothing" decision – a POA either supports transient objects or persistent objects, but not a mix of the two.

4.4.7 ObjectId Assignment Policy

This policy controls whether the POA assigns `ObjectIds` or whether they are supplied by the application.

4.5 Evaluation of the POA

As you can see, the new POA specification defines a wide range of policies that enable developers to tailor an ORB's behavior to meet many different application use-cases. One potential portability problem with these policies, however, is the fact that ORB vendors are likely to extend the available policies with their own proprietary policies. Hopefully, the vendors will continue to feed their ideas for useful new POA policies into the OMG Portability Enhancement Revision Task Force so as to maintain portability.

At first glance, the flexibility provided by all of these POA policies may seem overwhelming and complex. However, the POA can also be quite simple to use for simple applications. For example, here's a simple, compliant, and portable server program:

```
int main (int argc, char *argv[])
{
  using namespace CORBA;
  using namespace PortableServer;

  // Initialize the ORB.
  ORB_var orb = ORB_init (argc, argv);

  // Obtain an object reference for
  // the Root POA.
  Object_var obj =
    orb->resolve_initial_references ("RootPOA");
  POA_var poa = POA::_narrow(obj);

  // Incarnate a servant.
  My_Servant_Impl servant;

  // Implicitly register the servant
  // with the RootPOA.
  obj = servant._this ();

  // Start the POA listening for requests.
  poa->the_POAManager ()->activate ();

  // Run the ORB's event loop.
  orb->run ();

  // ...
}
```

This example first initializes the ORB, then obtains an object reference for the Root POA from it. It then creates an instance of a servant class called `My_Servant_Impl` and invokes its `_this` method. This implicitly registers the servant with the Root POA and generates an object reference for the newly-created CORBA object. Because of the default policies required for the Root POA by the Portability Enhancement specification, this new CORBA object is a transient object. The POA's manager is then activated to enable it to handle requests. Finally, the `ORB::run` operation is called to run the main event loop of the server application's ORB, which handles all requests and performs upcalls to `My_Servant_Impl`.

5 Concluding Remarks

In this column, we've provided a detailed overview of the new CORBA Portable Object Adapter (POA). We've defined and clarified the terms used in the POA specification and discussed some of the POA features. In general, the POA is much more portable and more powerful than the BOA. Yet, as shown by the short example above, it can be used quite simply as well.

In our next column, we'll return to our traditional practice of providing actual C++ code examples that show how to apply these features. For instance, we'll provide more details about our simple POA example and we'll show other C++ examples to illustrate the power and flexibility of the POA policies. We'll also discuss the circumstances where POA features and policies can be used most effectively. In future columns, we'll discuss issues associated with implementing the POA, such as how to minimize request demultiplexing overhead and how to add real-time scheduling support.

Thanks to Thorsten Albrecht, Jonathan Biggar, Laurent Chardonnens, Anton van Straaten, and Irfan Pyarali for their helpful comments on this column.

References

[1] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[3] IONA, "IONA's Object Database Framework Adapter (ODAF)." www-usa.iona.com/Press/PR/odaf.html, 1997.

[4] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[5] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, November 1997.

[6] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Object," *C++ Report*, vol. 8, July 1996.

[7] Object Management Group, *ORB Portability Enhancement RFP*, OMG Document 1995/95-06-26 ed., June 1995.

[8] Object Management Group, *Specification of the Portable Object Adapter (POA)*, OMG Document orbos/97-05-15 ed., June 1997.

[9] D. Schmidt and S. Vinoski, "Distributed Callbacks and Decoupled Communication in CORBA," *C++ Report*, vol. 8, October 1996.

[10] J. Klefndfenst, F. Plasfl, and P. Tuma, "Lessons Learned from Implementing the CORBA Persistence Object Servicesj," in *Proceedings of OOPSLA '96*, (San Jose, CA), ACM, October 1996.

[11] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool," *C++ Report*, vol. 8, April 1996.

[12] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Request," *C++ Report*, vol. 8, February 1996.