Object Interconnections

Developing C++ Servant Classes
Using the Portable Object Adapter
(Column 13)

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 June 1998 issue of the SIGS C++ Report magazine.

1 Introduction

This is the third column in our series explaining the new OMG *Portable Object Adapter*. An Object Adapter is the CORBA component responsible for adapting CORBA's concept of objects to a programming language's concept of servants. As you'll recall from previous columns [1, 2], CORBA objects are "abstract" entities defined by IDL interfaces. Likewise, servants are "concrete" implementations of CORBA objects written in a particular programming language, such as C++ or Java.

The Portable Object Adapter (POA) [3] is a replacement for the Basic Object Adapter (BOA) defined in earlier CORBA specifications. The newest version of CORBA, CORBA 2.2, [4], contains both the POA specification and the BOA specification. The BOA specification will be removed in future versions of CORBA, however, because the POA specification supersedes the BOA. As POA implementations are provided by vendors, therefore, C++ developers creating new CORBA applications should use the POA to satisfy the object adapter requirements of their servers.

The examples we showed in the previous column showed how to obtain object references from servants, as well as how to register servants with a POA instance. However, we did not show the definitions of the C++ servant classes. In addition to addressing this omission, this column explains how the new POA specification separates the client-side stub hierarchy from the server-side skeleton hierarchy in order to facilitate collocation and ensure source-level portability.

2 The PortableServer::Servant Type

CORBA interfaces are specified in OMG IDL to make them independent of any particular programming language. An OMG IDL compiler is responsible for translating IDL interfaces into a particular programming language. For instance, a POA-compliant IDL-to-C++ compiler would map the following interface for the stock quoting service described in [5]:

```
module Stock
{
  interface Quoter
  {
    // Return the current value of <stock_name>.
    long get_quote (in string stock_name);
  };

  // ...
}
```

into the C++ code shown below:

```
namespace POA_Stock
{
  class Quoter
  {
  public:
    virtual CORBA::Long
      get_quote (const char *stock_name)
         throw (CORBA::SystemException) = 0;

    // ...
  };

  // ...
};
```

Naturally, a POA-compliant IDL compiler could also generate Ada, C, COBOL, Java, Smalltalk, or any other language supported by CORBA. At the time of this writing (March 1998), however, only the C and C++ POA mappings have been finalized by the OMG, and the Java mapping is nearly complete.

Using an interface definition language like OMG IDL allows developers to select the most suitable programming language for different aspects of their systems, *e.g.*, Java on the client and C++ on the server. In addition, IDL simplifies the integration of legacy components written in different programming languages.

While the programming language independence of IDL is well suited for application-defined interfaces, it is not entirely appropriate for standard POA interfaces. As mentioned in Section 1, the POA provides the "glue" that adapts the abstract world of CORBA objects to the concrete world of programming language servants. Therefore, to allow applications to register programming language entities to serve CORBA objects, certain aspects of the standard POA interface must be specified in a manner that can be mapped to specific programming language features.

The POA specification introduces a new OMG IDL type that allows programming language-specific entities, such as Java objects or pointers to C++ instances, to be defined as part of the language-independent POA IDL specification. This new type is denoted by the new `native` keyword. The `native` keyword provides a language-specific "escape" from IDL. In this respect, it is similar to the C++ keyword `asm`, which provides a CPU-specific "escape" to allow embedded assembly language instructions in a C++ program. However, the `native` keyword is only intended for use by the OMG to help specify core CORBA interfaces. Thus, it is not meant for use by application developers.

The first and foremost use of `native` is to define the `Servant` type, which is the basis for all skeletons. `Servant` is defined in the POA's `PortableServer` module as follows:

```
module PortableServer
{
  native Servant;
  // ...
};
```

The `native` construct is similar to a `typedef` in that it introduces a new typename. Unlike `typedef`, however, `native` expresses the fact that the new type is defined *outside* of OMG IDL *for each programming language*.

In the POA mapping of IDL-to-C++, the `Servant` type is defined as follows:

```
namespace PortableServer
{
  class ServantBase
  {
  public:
    // Virtual destructor ensures correct
    // deletion semantics.
    virtual ~ServantBase (void);

    // Returns the default POA for each Servant.
    virtual POA_ptr _default_POA (void);

  protected:
    // Make ServantBase an ``abstract class.''
    ServantBase (void);
  };

  typedef ServantBase *Servant;
  // ...
}
```

In C++, the `Servant` type is defined as a pointer to a `ServantBase`, which serves as the base class from which all servant classes are (indirectly) derived. All C++ servant classes implement IDL interfaces. Before we can show an example of an application-defined C++ servant class definition, however, we need to supply an IDL interface to base it on.

3 A User-Defined Servant Class

The IDL interface shown below was first shown in [5]. It defines a `Quoter_Factory` that creates instances of the `Quoter` interface shown in Section 2.

```
// IDL
module Stock
{
  // ...
  interface Quoter_Factory
  {
    // Factory method that creates the
    // <Quoter> specified by <name>.
    Quoter create_quoter (in string name);
  };
};
```

The `create_quoter` operation is a factory method [6] that returns an object reference to a `Quoter` service specified by its `name`. As usual, this interface has been simplified to concentrate our focus on C++ servant class definitions. Production quality IDL interfaces would define exceptions that could be raised if run-time errors occurred.

3.1 Inheritance-Based Servant

An application might define a C++ servant class for the `Quoter_Factory` interface as follows:

```
class My_Quoter_Factory :
  public POA_Stock::Quoter_Factory
{
public:
  My_Quoter_Factory (void);
  ~My_Quoter_Factory (void);

  virtual Stock::Quoter_ptr create_quoter
      (const char *name)
    throw (CORBA::SystemException);
};
```

There are several interesting features to note about the `My_Quoter_Factory` class:

1. Standardized naming for the base class: A significant improvement of the POA specification over the BOA is that the names of skeletons are finally standardized. In general, the POA specification mandates that server-side names are formed by prepending "`POA_`" to the name of the outermost scope. For example, the C++ namespace generated by the IDL compiler for the `Stock` module is `POA_Stock`. Likewise, the name of the generated skeleton class corresponding to the `Quoter_Factory` interface must be "`Quoter_Factory`" since it's already scoped by the `POA_Stock` namespace. If the `Quoter_Factory` interface were defined at global scope instead of within a module, however, its corresponding C++ class would be named "`POA_Quoter_Factory`." Section 4 describes how these naming rules facilitate pure clients and collocation optimizations.

2. Exception specifications: Each method explicitly declares the CORBA exception types it is allowed to throw. The operation declared in our `Quoter_Factory` interface does not raise any user-defined exceptions. Therefore, our C++ exception specifications only allow CORBA system exceptions to be raised since they can be thrown by all OMG IDL operations.

The `My_Quoter_Factory` servant class uses the *inheritance-based approach* to servant class definition, which is an example of the "class form" of the Adapter pattern[6]. As shown in Figure 1, the application derives a concrete class from the abstract skeleton base class, in this case named `Quoter_Factory` from the `POA_Stock` namespace, generated by the IDL compiler. Each pure virtual method, *i.e.*, `create_quoter`, inherited from the base class must be overridden and implemented by the application servant class.

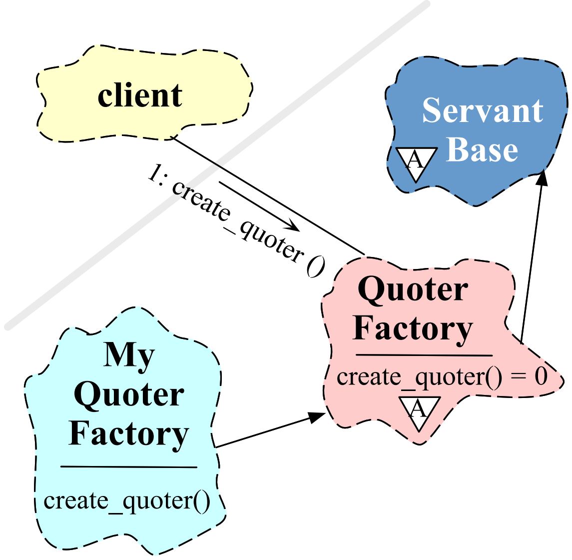

Figure 1: The Class Form of the Adapter Pattern

3.2 Tie-Based Servant

Another way to define a C++ servant is to use delegation, *i.e.*, the "object form" of the Adapter pattern[6], instead of inheritance. The POA specification calls this the "tie" approach[1] because the C++ instance being delegated to is "tied" into the servant. As shown in Figure 2, the servant class is actually a C++ parameterized type that itself derives from the abstract skeleton base class, *e.g.*, `Quoter_Factory` from the `POA_Stock` namespace. This tie class overrides all skeleton pure virtual methods so they delegate to another C++ object, called the "tied object," which is supplied by the application.

The following illustrates code that might be generated by an IDL compiler using the tie approach[2]:

```
namespace POA_Stock
{
  // ...

  template <class T>
  class Quoter_Factory_tie : public Quoter_Factory
  {
  public:
    // = Initialization/termination methods.
    Quoter_Factory_tie (T &t);
    Quoter_Factory_tie (T &t,
                        POA_ptr poa);
    Quoter_Factory_tie (T *tp,
                        Boolean release = 1);
    Quoter_Factory_tie (T *tp, POA_ptr poa,
                        Boolean release = 1);
    ~Quoter_Factory_tie (void);

    // = Tie-specific methods.
    T *_tied_object (void);
    void _tied_object (T &obj);
    void _tied_object (T *obj,
                       Boolean release = 1);
    Boolean _is_owner (void);
    void _is_owner (Boolean b);

    // = Delegated IDL operation.
    virtual Stock::Quoter_ptr create_quoter
        (const char *name)
      throw (CORBA::SystemException)
    {
      return impl_->create_quoter (name);
    }

    // = Override ServantBase operation.
    virtual POA_ptr _default_POA (void);

  private:
    T *impl_;
    // other data members not shown.

    // = Copying and assignment not allowed.
    Quoter_Factory_tie(const Quoter_Factory_tie&);
    void operator=(const Quoter_Factory_tie&);
  };
}
```

[1] The term "tie" comes from Orbix, the first ORB to promote this style of servant implementation.

[2] Jon Biggar originally pointed out that if the C++ compiler doesn't support namespaces, the `POA_Stock` module will be mapped to a C++ class instead of a namespace, and this code won't work. The OMG C++ Revision Task Force, of which Steve is the chair, needs to fix this portability issue.

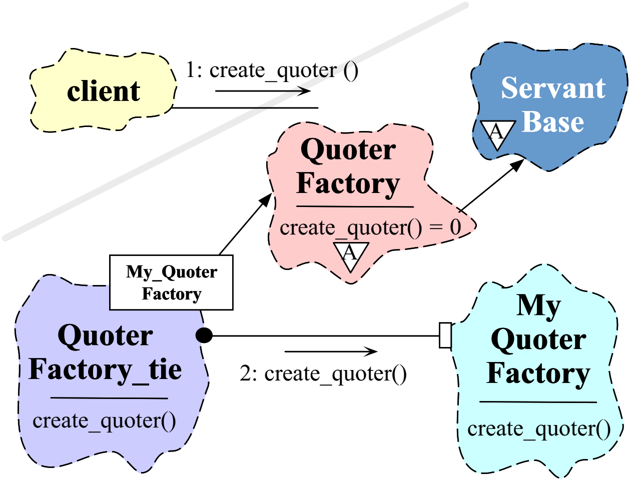

Figure 2: The Object Form of the Adapter Pattern

The standard tie template class overrides the virtual method it inherits from `ServantBase` and from the type-specific POA skeleton base classes. In addition, it supplies a number of other operations. These extra operations control ownership of the tied object, *e.g.*, letting the tie instance own the tied object or just making it refer to it without owning it. Operations for accessing and replacing the tied object are also provided.

With respect to POA terminology, it's the instance of the tie template class that is the servant, not the tied object. This is because it's the tie template class instance that derives from `ServantBase`, allowing it to be registered as a servant with the POA. Internally, POA implementations maintain a map of servants that are registered by applications. At run-time this map is used by the POA to demultiplex incoming client requests to the appropriate servant [7].

The main purpose of ties is to allow classes that aren't related to skeletons by inheritance, *i.e.*, the tied objects, to implement CORBA object operations. For each operation they supply, they expect the tied object to have a method with exactly the same signature. Thus, a common criticism leveled against ties is that they're not as flexible as they might seem to be. In particular, it is unlikely that the signature of a non-CORBA class will serendipitously match the signature of the tie class.

The problem with signature mismatches can be alleviated somewhat for the IDL-to-C++ mapping using the Adapter pattern [6]. One application of the Adapter pattern uses C++ *template specializations*. For example, if we had an existing class called `Stock_Factory`, with a method called `create` instead of `create_quoter`, we could specialize the `Quoter_Factory_tie` implementation of the `create_quoter` operation, as follows:

```
Stock::Quoter_ptr
Quoter_Factory_tie<Stock_Factory>::create_quoter
  (const char *name)
{
  return impl_->create (name);
}
```

Though it requires manual intervention, C++ template specialization allows application developers to adapt the tie parameterized type to conform with interfaces of myriad tied object implementations. Of course, if the application requires specializing each and every tie operation to correctly adapt to the class of the tied object, other variants of the Adapter pattern can be applied to integrate legacy C++ classes with generated POA tie templates. For instance, it may be easier to just use the inheritance-based approach and write a custom tie using the class form of the Adapter pattern.

3.3 Implementing the Server Program

Regardless of whether we use the inheritance or the tie approach for integrating automatically-generated skeletons with servant classes, the servant methods can be written generically. For instance, a C++ programmer could define the `create_quoter` method for the `My_Quoter_Factory` class as follows:

```
Stock::Quoter_ptr
My_Quoter_Factory::create_quoter (const char *name)
{
  POA_Stock::Quoter *quoter;

  // Select the desired subclass of Quoter.
  if (strcmp (name, "Dow Jones") == 0)
    quoter = new Dow_Jones_Quoter;
  else if (strcmp (name, "Reuters") == 0)
    // Dynamically allocate a new  object.
    quoter = new Reuters_Quoter;
  else // ...

  // This call will create a Stock::Quoter_ptr
  // object reference and register the servant
  // with its default_POA.
  return quoter->_this ();
};
```

The main program for our stock quote server could be defined as follows, assuming we use the tie approach:

```
typedef POA_Stock::Quoter_Factory_tie
          <My_Quoter_Factory>
        MY_QUOTER_FACTORY;

void main (int argc, char *argv[])
{
  ORB_Manager orb_manager (argc, argv);

  // Dynamically create the "tied object."
  My_Quoter_Factory *qf = new My_Quoter_Factory;

  // Create the "tie servant."
  MY_QUOTER_FACTORY factory (qf);

  // Explicitly register the servant with the POA.
  orb_manager.activate (&factory);

  // Block indefinitely waiting for incoming
  // invocations and dispatching method callbacks.
  orb_manager.run ();
  // After run() returns, the ORB has shutdown.
}
```

Note how our server is simplified with the help of the following `ORB_Manager` class:

```
class ORB_Manager
  // = TITLE
  //   Helper class for simple ORB/POA
  //   initialization and registering servants
  //   with the POA.  Works with standard OMG
  //   POA interface.
{
public:
  // Initialize the ORB manager.
  ORB_Manager (int argc, char *argv[]) {
    orb_ = CORBA::ORB_init (argc, argv);

    CORBA::Object_var obj =
      orb->resolve_initial_references ("RootPOA");

    poa_ = PortableServer::POA::_narrow (obj.in ());
    poa_manager_ = this->poa_->the_POAManager ();
  }

  // Register <servant> with the <poa_>.
  void activate (PortableServer::Servant servant) {
    PortableServer::ObjectId_var id =
      poa_->activate_object (servant);
  }

  // Run the main ORB event loop.
  void run (void) {
    poa_manager_->activate ();
    orb_->run ();
  }

private:
  CORBA::ORB_var orb_;
  PortableServer::POA_var poa_;
  PortableServer::POA_Manager_var poa_manager_;
};
```

Not only does this class simplify common use-cases of the POA, but it also can be used to alleviate portability issues while ORB vendors transition from the BOA to the POA. For example, the following implementation of `ORB_Manager` works with Orbix, which currently supports the BOA rather than the POA specification:

```
class ORB_Manager
  // = TITLE
  //   Helper class for simple ORB/POA
  //   initialization and registering
  //   servants with the POA.  Works
  //   with Orbix BOA interface.
{
public:
  // Initialize the ORB manager.
  ORB_Manager (int argc, char *argv[]) {
    // First argument is assumed to be
    // server name.
    svr_name_ = argv[1];
  }

  void activate (void *servant) {
    // Nothing to do, since servant base
    // class constructor performs activation
    // when the servant is created.
  }

  // Run the main ORB event loop.
  void run (void) {
    CORBA::Orbix.impl_is_ready (svr_name_);
  }

private:
  char *svr_name_;
};
```

4 Collocation Issues

One of the most significant benefits of the POA specification is that it defines standard names for stubs, skeletons, and servant classes. It also defines how these entities are related and how to integrate them with application code. The earlier BOA specification did not define these names and relationships explicitly, which made it hard to write portable CORBA server applications.

This section outlines the improvements provided by the POA and illustrates how they can be applied to enable efficient *collocation* of clients and servants. Collocation optimizes away network transport overhead when clients and servants are configured into the same address space.

If you look carefully at the `My_Quoter_Factory` you'll notice some interesting naming conventions for the `Stock` module namespace:

```
class My_Quoter_Factory :
  public POA_Stock::Quoter_Factory
{
public:
  // ...
  virtual Stock::Quoter_ptr create_quoter
      (const char *name)
    throw (CORBA::SystemException);
  // ...
};
```

In particular, note how the `Quoter_Factory` class is qualified by a `POA_Stock` prefix, whereas the `Quoter_ptr` is simply qualified by a `Stock` prefix. There two general reasons for these seemingly curious naming conventions:

1. Pure clients: One benefit of the use of separate namespaces for client-side and server-side definitions is the ability to define *pure clients*. Pure client applications contain no CORBA objects and perform no server functions. Therefore, they need not include definitions for unused server-side classes and data types. This results in pure client applications with smaller memory footprints.

2. Collocation support: Contemporary ORBs often utilize direct pointers to servants when the target object is located in the same process as the invoking client. This optimization uses a very efficient virtual method dispatch to invoke the operation on the collocated target object. The POA naming conventions allow servants to be separated completely from the object reference class hierarchy, thereby facilitating collocation. Figure 3 illustrates the distinction between these two hierarchies.

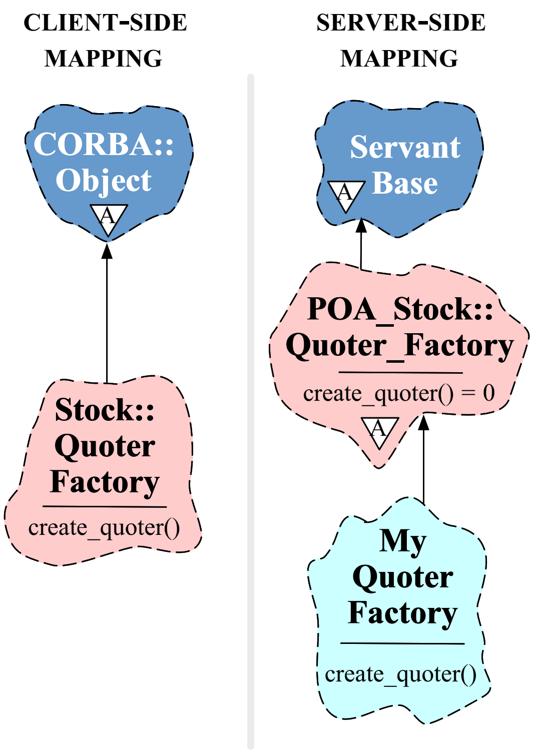

Figure 3: Client-side and Server-side POA Mappings

The left half Figure 3 shows the object reference C++ class hierarchy. This hierarchy has as its root the C++ class representing `CORBA::Object`, which is the base of all OMG IDL interfaces. Most ORB implementations implement object references as pointers to these class types. The right half of the figure shows the corresponding skeleton C++ class hierarchy. Note that all by the `My_Quoter_Factory` are generated by the IDL compiler.

ORB implementations are allowed, but not required, to have their skeletons derive from `CORBA::Object`. Therefore, portable applications must assume that no such derivation exists, *i.e.*, that skeletons are not part of the object reference C++ class hierarchy. In particular, taking the address of a servant and using it as a collocated object reference is not portable.

This restriction may seem unusual because existing BOA implementations of popular contemporary ORBs like Orbix and VisiBroker *do* derive their skeleton classes from the corresponding object reference classes. Thus, application developers often wonder why the OMG chose a different model for POA skeletons. The remainder of this section explains why POA skeletons are separate from the `CORBA::Object` class hierarchy.

4.1 Multi-object Servants

With the POA, a single servant may incarnate multiple CORBA objects. Therefore, taking the address of a servant derived from such a skeleton, or otherwise using implicit C++ conversion mechanisms to convert it to an object reference, results in an ambiguity problem: it is impossible to know which CORBA object incarnated by the servant is referred to by such an object reference. Furthermore, there is no way to prevent this error, because it relies solely on the C++ conversion of a pointer to derived servant class to a pointer to a base object reference class. C++ (correctly) provides no hooks to allow a user to prevent or verify that such conversions should be allowed to take place.

The scalability afforded by allowing a single servant to incarnate multiple CORBA objects was deemed by the POA designers[3] to be far more important than allowing implicit conversions of servants to object references.

4.2 Collocation Transparency

Although collocation bypasses the relatively slow delivery of the request via the network protocol stack, it can cause subtle problems related to the transparency of collocation optimizations. For instance, a collocated client may be left holding a dangling pointer when a CORBA object is destroyed. Likewise, a servant can be deleted out from under a request while an invocation is in progress. Moreover, certain POA features may not function correctly. In particular, the `POACurrent` interface (which allows servant to determine which POA and ObjectId it was invoked for) and the POAManager (which allows servers to hold or discard incoming requests for servants) will not behave properly if a client holds a direct pointer to its collocated servant.

Recent changes to CORBA and its services also affect collocation transparency. For instance, the OMG Security Service[8] officially introduced to the CORBA specification the notion of *interceptors*, which are examples of the Chain of Responsibility pattern [6]. Interceptors greatly enhance ORB flexibility by separating request processing from the traditional ORB networking mechanisms required to send and receive requests and replies.

The CORBA interceptor concepts are based on Marc Shapiro's work on flexible bindings[9], and on Orbix filters, which were the first application of Shapiro's work to CORBA-based systems. Other ORB vendors such as Sun and Borland/Visigenic now support interceptors as well.

Interceptors are intimately tied into the connection between the client and server. Therefore, they can affect the contents of CORBA requests and replies as they're exchanged. For example, a client-side *security interceptor* might transparently add authorization information to a request before it leaves the client process. The matching server-side security interceptor in the receiving server would verify that the client is authorized to invoke requests on the target object before the request is dispatched. If authorization fails, the request should be rejected.

Another example of an interceptor is a *transaction interceptor*. This interceptor adds a transaction ID to a request before it leaves the client. The corresponding server-side transaction interceptor then ensures that the request is dispatched to the target object within the context of that particular transaction.

Interceptors work because they are (transparently) interposed between the client and the target object. Therefore, for collocated client and target servant to work properly, interceptors must still perform their duties. For example, an application should not have to perform extra work to ensure that invocations on local objects were correctly included as part of a distributed transaction.

Clearly, if object references to collocated objects were direct pointers to servants, no interceptors could be invoked by the ORB for operations on local objects. Such a design would result in a loss of local/remote transparency.

4.3 Object-oriented Typesystem Conformance

In the object-oriented programming paradigm, inheritance signifies that a derived type can be used where a base type is expected. In other words, a `Derived` class "IS-A" `Base` class, *i.e.*, a derived type completely fulfills the interface of each of its base types.

Our last two columns have focused on the significant differences between a CORBA object and a programming language servant. For instance, one difference is that a single servant can incarnate multiple CORBA objects. Such differences make it clear that a subclass of `Servant` does not have an IS-A relationship with `CORBA::Object`. Therefore, inheritance of POA skeletons from the `CORBA::Object` class hierarchy does not follow widely-accepted OOP principles with respect to inheritance.

[3] Steve is one of those designers, so we're not just guessing here!

4.4 CORBA Object Destruction

Consider a server application that invokes requests on its own collocated objects, and also allows remote clients to do the same. Both remote clients and different threads within the application itself are allowed to destroy any of these CORBA objects at any time. The following problems arise if a remote client destroys a CORBA object incarnated by a particular servant, and the local application holds a collocated object reference (a C++ pointer directly to the servant) to the same object:

Wrongly-extended lifetime: The collocated client might still be able to use the direct pointer to invoke operations on a CORBA object that has already been destroyed, when in fact it should receive a `CORBA::OBJECT_NOT_EXIST` exception instead.

Dangling Pointers: A more serious problem is that such invocations would most likely be performed using a dangling pointer to the already-deleted servant, and the application would probably crash.

For these reasons, the POA was designed to avoid requiring that POA skeletons must derive from the `CORBA::Object` C++ object reference class hierarchy. To do otherwise would have been too restrictive, and would have decreased the utility of collocated objects.

4.5 Evaluating the POA Hierarchies

The separation of POA skeleton classes from object reference classes avoids the problems described above because the lifetimes of servants are decoupled from the lifetimes of object references. The downside of adding these semantics to the POA and to the ORB Core, however, is that collocation may not be implemented as efficiently as it can with a direct pointer to the servant.

Collocation overhead can arise in a number of places. For instance, there is overhead associated with setting up a `POACurrent` object, checking the state of the POAManager, and possibly transferring the request to be dispatched on the POA's thread. For these reasons, most responses to the recent OMG Real-Time Special Interest Group (SIG) request for proposals (RFP)[10] called for omitting many of the more complex POA features for the forthcoming version of real-time CORBA.

5 Concluding Remarks

This column addressed the structure of POA skeleton classes and the servant classes that derive from them. Both inheritance-based and tie-based servants were described. In keeping with our tradition of showing working C++ code, examples of each were shown. We then provided an in-depth look into collocation of clients and target objects, and described the benefits of separating client-side object reference class hierarchies from server-side skeleton class hierarchies. Though the POA approach differs from most (but not all) contemporary ORB implementations, it provides better consistency, safety, and scalability. We expect that most CORBA vendors will enhance their ORBs to conform to the POA specification shortly.

Our next column will be the last in our POA series. It will cover on-demand servant activation via *servant managers*, which are extremely useful for creating highly-scalable server applications. After that, we'll begin covering the new *Objects By Value* specification[11] and its ramifications on CORBA applications. At the time of this writing (March 1998), this specification is in the process of OMG adoption.

As always, if you have any questions about the material we covered in this column or in any previous ones, please email us at `object_connect@cs.wustl.edu`.

References

[1] D. C. Schmidt and S. Vinoski, "Object Adapters: Concepts and Terminology," *C++ Report*, vol. 11, November/December 1997.

[2] D. C. Schmidt and S. Vinoski, "Using the Portable Object Adapter for Transient and Persistent CORBA Objects," *C++ Report*, vol. 12, April 1998.

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

[4] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Mar. 1998.

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

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

[7] 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.

[8] Object Management Group, *OMG Security Service*, OMG Document ptc/98-01-02, revision 1.2 ed., January 1998.

[9] M. Shapiro, "Flexible Bindings for Fine-Grain, Distributed Objects," Tech. Rep. Rapport de recherche INRIA 2007, INRIA, Aug. 1993.

[10] Object Management Group, *OMG Real-time Request for Proposal*, OMG Document ptc/97-06-20 ed., June 1997.

[11] Object Management Group, *Objects-by-Value*, OMG Document orbos/98-01-18 ed., January 1998.