**Object Interconnections: Dynamic CORBA, Part 3 - The Dynamic Skeleton Interface**

Some server applications, such as gateways or monitors, cannot know a priori the types or identities of the objects they must serve. In this column, we show how the CORBA Dynamic Skeleton Interface enables developers to construct such applications portably.

**Introduction**

Welcome to the third installment of our four part series covering Dynamic CORBA. In Part 1 [[1]](http://www.ddj.com/cpp/184403847#1), we discussed the CORBA DII (Dynamic Invocation Interface), which allows applications to invoke operations on target objects without having compile-time knowledge of target interfaces. In Part 2 [[2]](http://www.ddj.com/cpp/184403847#2), we explained the basics of the Dynamic Any, which enables applications to handle any value of any IDL type, whether simple or complex, without having compile-time information about that type. The Dynamic Any is essential for constructing truly dynamic CORBA client or server applications.

In this column, we present the CORBA DSI (Dynamic Skeleton Interface), which is essentially the server-side counterpart of the DII. Some server applications, such as gateways or monitors, cannot know *a priori* the types or identities of the objects they must serve. The DSI enables CORBA developers to construct such applications portably. There are many capabilities associated with the DSI, so we first cover the basics and then present the more advanced features.

**Dynamic Skeleton Basics**

To build working C++ DSI applications, you need to understand the following two key classes:

1. The **CORBA::ServerRequest** class, which is a pseudo-object that is the server-side equivalent to the client-side **CORBA::Request**. When the POA dispatches a request to a DSI servant, it passes an instance of **ServerRequest** to communicate all information about the request needed to allow the servant to fulfill it.
2. The **PortableServer::DynamicImplementation** servant base class, from which concrete DSI servant classes inherit. This base class provides the following pure virtual methods that DSI servant classes are expected to override:
   * **void invoke (ServerRequest\_ptr server\_request)**, which is upcalled by the POA to allow the servant to handle requests. The POA passes the **ServerRequest** representing the request being processed to this method.
   * **RepositoryId \_primary\_interface (const ObjectId& id, POA\_ptr poa)**, which is called by the ORB/POA run time when it needs the repository ID string identifying the most-derived interface of the object that the servant is incarnating. The ORB run time passes **ObjectId** for the specific object whose repository ID is requested by a DSI servant since such a servant normally incarnates multiple objects. The second argument supplies a reference to the particular POA in which the repository ID request is occurring. The **\_primary\_interface** method is called when the application calls POA methods, such as **servant\_to\_reference** and **id\_to\_reference**, so that the run time can create object references containing the appropriate repository ID.

The **\_primary\_interface** method is straightforward, so our examples below focus on the complexities of the **invoke** method.

In our first example, we return to the simple IDL we used in Part 1 of this series, but this time we use it to show a simple DSI server.

// IDL

interface A {

void op ();

};

When a client invokes **A::op** on an object incarnated by a DSI servant, the dispatching POA delivers a **ServerRequest** object to the **invoke** method on the servant. **ServerRequest** supplies methods that allow the servant to deal with all aspects of the request, including determining its name, examining its input arguments and setting its output arguments, setting a return value, and raising exceptions.

The first thing the servant must do to handle a request in its **invoke** method is to determine which operation is being invoked. It does this by calling the **operation** method on **ServerRequest**, like this:

std::string op = server\_request->operation ();

if (op == "op") {

// process op

request

}

The **operation** method returns the simple name of the IDL operation being invoked. Note that we store the returned name into a C++ **std::string**. Like the methods provided by the DII **Request** class, **ServerRequest**'s methods do not follow normal C++ mapping rules. Since the **operation** method returns a **const char\*** for which it maintains ownership, we're able to copy the returned string into a **std::string** here without creating a memory leak.

After the servant determines which operation it's servicing, it must then deal with the operation's arguments. With the DSI, the servant is responsible for informing the ORB of the number of arguments the operation expects, along with their types and their directions (i.e., whether the argument is **in**, **inout**, or **out**). The situation is actually the same for static skeleton invocation on the server. In that case, however, the skeletons generated automatically by the IDL compiler take care of providing argument information to the ORB, thereby shielding the servant developer from these details.

You might think that the ORB would automatically know what's in a marshaled request. To minimize time and space overhead, however, CORBA is designed such that the receiver knows what to expect. CORBA does not mandate self-describing requests because it would mean that unnecessary information would be sent over the network for the majority of cases. As a result, IIOP requests never contain **TypeCode**s to describe the argument types (except, of course, for the **TypeCode**s that are always marshaled for **Any** parameters). When a server ORB receives a request, the request's header provides only enough information to allow the ORB and POA to dispatch the request. Until the skeleton tells it how to demarshal everything, therefore, the server ORB must view the arguments as an array of indecipherable bytes.

The servant tells the ORB what arguments to expect by calling the **ServerRequest::arguments** method. Although the **op** method in our example has no arguments, the servant must still call **arguments** exactly once, whether the invoked operation has arguments or not. The servant passes argument information in the form of an **NVList**, as shown below:

CORBA::NVList\_var args;

orb->create\_list (0, args);

The ORB's **create\_list** method takes two parameters: (1) a count of the number of items the list should contain and (2) an **out** **NVList** argument that the ORB uses to return the result. In our example, we expect zero arguments, so **create\_list** returns an empty **NVList**, which we then pass to the **arguments** operation, like this:

server\_request->arguments (args);

Our next example will show the details of argument processing.

The final step is to complete the operation invocation by either returning normally or by raising an exception. To indicate a normal return, the servant calls **ServerRequest::set\_result**, as we'll show in our next example. To raise an exception, the servant calls **ServerRequest::set\_exception**. For example, the code below shows how to throw a **BAD\_OPERATION** exception to inform clients that they've invoked a method that the target object does not support:

CORBA::BAD\_OPERATION exception;

CORBA::Any any;

any <<= exception;

server\_request->set\_exception (any);

Both **set\_result** and **set\_exception** take a single **Any** parameter. For **set\_result**, **Any**'s type must match the return type of the invoked operation. For **set\_exception**, **Any** must contain an exception that is allowed by the operation's **raises** clause. A servant may call either **set\_result** or **set\_exception**, but not both. Moreover, whichever one it calls, it can call it only once.

**A More Realistic DSI Use Case**

Our example above was straightforward, but overly simplistic since in practice DSI servants usually deal with arguments and return values. To show this, let's return to our stock **Info** example from our previous column. The IDL for this example is shown below:

// IDL

module Stock {

exception Invalid\_Stock {};

struct Info {

long high;

long low;

long last;

};

interface Quoter {

Info get\_quote (in string stock\_name) raises

(Invalid\_Stock);

};

};

Let's focus on the argument handling required to implement **get\_quote** in a DSI servant. We create an **NVList** as before, but this time, we must populate it with type and argument direction information. For simplicity, our example will assume that our servant already knows the necessary information, so it doesn't need to interact with the IFR (Interface Repository).

CORBA::NVList\_var args;

orb->create\_list (1, args);

args->add (CORBA::ARG\_IN)->value ()->replace

(CORBA::tk\_string, 0);

server\_request->arguments (args);

Here, we've changed the second line to create an **NVList** with one argument. The most significant differences occur on the third line, however, where we add a **NamedValue** to our **NVList** and indicate that it will be an **in** argument by setting its **Flag** value to **CORBA::ARG\_IN**. The **add** method returns this new **NamedValue**, for which the **NVList** retains ownership. On this returned **NamedValue**, we invoke the **value** method to gain access to the **Any** it holds, and on that **Any** we use the **replace** method to set the argument type. In our example, the argument's type is **string**, so we use the **CORBA::tk\_string** enumerator to indicate this. The second argument to **replace** is normally a pointer to the value to be stored in **Any**. In our case, however, the value will be demarshaled by the ORB, so we simply indicate that we are not setting the value by passing a null pointer.

When we pass this populated **NVList** to the **arguments** method, the ORB demarshals the **string** argument and inserts it into the **Any** inside the **NVList**. After **arguments** returns, we can access the value of the argument using regular **Any** extraction or by using the Dynamic Any. We'll use the latter, since we explained the Dynamic Any in our previous column [[2]](http://www.ddj.com/cpp/184403847#2) and also because we'll need to use it to deal with the **Info** return value. We first obtain a Dynamic Any factory, which we then use in conjunction with the argument stored in the **NVList** to create a Dynamic Any:

CORBA::Object\_var obj =

orb->resolve\_initial\_references ("DynAnyFactory");

DynamicAny::DynAnyFactory\_var dynfactory =

DynamicAny::DynAnyFactory::\_narrow (obj);

DynamicAny::DynAny\_var dynany =

dynfactory->create\_dyn\_any (\*args->item (0)->value ());

CORBA::String\_var stock = dynany->get\_string ();

Since we know that our argument is a **string**, we can easily access its value through the basic **DynAny** interface and store it into our **stock** variable.

After we've obtained the stock name, our servant would presumably use it to access a database of stock information to retrieve the values needed to populate our **Info** return value. Let's assume we've already retrieved the necessary data and stored it into three **long** variables named **stock\_high\_value**, **stock\_low\_value**, and **stock\_current\_value**. Let's also assume that we've already created a Dynamic Any to hold our **Info** **struct**, exactly as shown in our previous column [[2]](http://www.ddj.com/cpp/184403847#2). We then insert our stock values into our **Info** **struct** as follows:

dynstruct->insert\_long (stock\_high\_value);

dynstruct->next ();

dynstruct->insert\_long (stock\_low\_value);

dynstruct->next ();

dynstruct->insert\_long (stock\_current\_value);

To return this **Info** value from our DSI servant's **invoke** method, we obtain an **Any** from our Dynamic Any and then call **ServerRequest::set\_result** to indicate that it's our return value.

CORBA::Any\_var any = dynstruct->to\_any ();

server\_request->set\_result (any);

Alternatively, if the input argument is a stock name for which we have no information, we want to raise a **Stock::InvalidStock** exception. Assuming we had enough information to create a Dynamic Any to hold such an exception (for an actual dynamic application, we would need IFR access), we could raise it as our exception by obtaining its value as an **Any** and then passing that **Any** to **ServerRequest::set\_exception**:

CORBA::Any\_var any = dynexception->to\_any ();

server\_request->set\_exception (any);

**POA Considerations**

A DSI servant normally incarnates multiple CORBA objects concurrently. Under these circumstances, the DSI servant is best used as a default servant [[3]](http://www.ddj.com/cpp/184403847#3), which means that it incarnates all objects served by the POA in which it's registered. Such a POA should be created with the following policies:

* **USE\_DEFAULT\_SERVANT**, which directs the POA to dispatch requests to its default servant that's set by the application using the **POA::set\_servant** method.
* **MULTIPLE\_ID**, which allows the default servant to incarnate multiple objects simultaneously.
* **USER\_ID**, which indicates that the application, not the system, supplies the object IDs used to identify objects in this POA.
* **NON\_RETAIN**, which means we're not registering any other servants with our POA other than our default servant, so this policy ensures that the POA avoids wasting space and time by creating and looking up servants in an Active Object Map.

With these POA policies in place, including the default **ORB\_CTRL\_MODEL** (which allows a POA to process multiple requests simultaneously), a DSI servant can expect to service requests for multiple CORBA objects concurrently. This introduces another problem, however: for any given request, how does the servant know which object it's expected to incarnate?

We have shown how a DSI servant can obtain the name of the operation it's servicing, via the **ServerRequest::operation** method. While the target object ID is technically part of the server request as well, it cannot be obtained through **ServerRequest**. Instead, the servant must obtain this information from the dispatching POA via its **Current** object, which supplies the following operations:

* **get\_POA**, which returns the POA that's dispatching the current request.
* **get\_object\_id**, which returns the ID of the target object for the current request.
* **get\_reference**, which returns an object reference for the target object.
* **get\_servant**, which returns the servant that's incarnating the target object for the current request.

The DSI servant gains access to the POA **Current** through the ORB's **resolve\_initial\_references** call:

CORBA::Object\_var obj =

orb->resolve\_initial\_references ("POACurrent");

PortableServer::Current\_var current =

PortableServer::Current::\_narrow (obj);

Since the POA **Current** resides in thread-specific storage [[4]](http://www.ddj.com/cpp/184403847#4), each thread can access the POA **Current** in this manner to obtain information about its specific target object, even when the same servant is servicing multiple requests for separate objects concurrently.

**Concluding Remarks**

The CORBA DII discussed in [[1]](http://www.ddj.com/cpp/184403847#1) provides the means to assemble and invoke a request dynamically, which is useful only to an application playing the role of a CORBA client. CORBA servers can also benefit from this type of dynamically request-processing capability, however. Dynamic CORBA therefore provides the DSI that enables servers to receive and handle requests without compile-time knowledge of operations or their signatures. In Dynamic CORBA, the DSI defines a **ServerRequest** that contains methods that allow the servant's **invoke** method to demarshal request arguments into **NamedValue** and **NVList** containers, as well as to marshal the reply to the client.

This column described the Dynamic CORBA DSI mechanisms and illustrated how these mechanisms can be applied to develop a server that's more flexible than one developed using static skeletons. There are two primary tradeoffs between using DSI and static skeletons in a CORBA server:

1. *Static versus dynamic type information.* A static skeleton has all necessary type information about arguments and return values compiled into it, which enables the IDL compiler to perform advanced optimizations [[5]](http://www.ddj.com/cpp/184403847#5). In contrast, the DSI approach normally requires type information to be discovered at run time, which is much more flexible but harder to optimize. The normal source for this information is the IFR. Our DII and DSI examples in this column series to date have not used the IFR, which we will show in our next and final column on Dynamic CORBA.
2. *Complexity versus flexibility*. A static skeleton turns an operation invocation into an ordinary call on the servant's C++ method call that implements the operation, which is easy to program but not very flexible since the servant's type must be known at compile time. A DSI servant, on the other hand, is much more flexible since it is possible for it to implement a wide range of operations that aren't known until run time. It is much more complex to program the DSI, however, since servant developers must not only implement the operation(s), but it must also follow the precise protocol required to inform the ORB of the number and types of arguments to demarshal, and to set the return value or raise an exception.

Because of these issues, the skill set required for Dynamic CORBA is greater than that required for developing static CORBA applications. Because static applications follow a natural function invocation paradigm, a minimum of C++ programming knowledge is needed to write a simple static CORBA client and server. Getting started is easy, given that ORB installations usually contain demo applications that a novice can easily modify to create a working system. Some ORBs even supply programming wizards or code generators that handle all of the details for you, allowing you to make quick work of all your small to medium static CORBA applications. Dynamic CORBA, on the other hand, requires a much broader understanding of CORBA's facilities, including **Request** and **ServerRequest** pseudo-objects, **TypeCode**s, **Any**, Dynamic Any, **NVList**s, **NamedValue**s, and for truly dynamic applications, the IFR and all of its constituent interfaces and data types. It's no surprise, therefore, that most CORBA C++ developers stick with static applications.

If you have comments, questions, or suggestions regarding Dynamic CORBA or our column in general, please let us know at **mailto:object\_connect@cs.wustl.edu**.

**References**

[1] Steve Vinoski and Douglas C. Schmidt. "Object Interconnections: Dynamic CORBA: Part 1, The Dynamic Invocation Interface," *C/C++ Users Journal*, July 2002, <www.cuj.com/experts/2007/vinoski.htm>.

[2] Steve Vinoski and Douglas C. Schmidt. "Object Interconnections: Dynamic CORBA: Part 2, Dynamic Any," *C/C++ Users Journal*, September 2002, <www.cuj.com/experts/2009/vinoski.htm>.

[3] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++* (Addison-Wesley, 1999).

[4] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* (Wiley and Sons, 2000).

[5] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. "Flick: A Flexible, Optimizing IDL Compiler," Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, NV, June 1997.

***Steve Vinoski****, <www.iona.com/hyplan/vinoski/>, is vice president of Platform Technologies and chief architect for IONA Technologies and is also an IONA Fellow. A frequent speaker at technical conferences, he has been giving CORBA tutorials around the globe since 1993. Steve helped put together several important OMG specifications, including CORBA 1.2, 2.0, 2.2, and 2.3; the OMG IDL C++ Language Mapping; the ORB Portability Specification; and the Objects By Value Specification. In 1996, he was a charter member of the OMG Architecture Board. He is currently the chair of the OMG IDL C++ Mapping Revision Task Force. He and Michi Henning are the authors of* Advanced CORBA Programming with C++*, published in January 1999 by Addison Wesley Longman. Steve also represents IONA in the W3C (World Wide Web Consortium) Web Services Architecture Working Group.*

***Doug Schmidt****, <www.ece.uci.edu/~schmidt/>, is an associate professor at the University of California, Irvine. His research focuses on patterns, optimization principles, and empirical analyses of object-oriented techniques that facilitate the development of high-performance, real-time distributed object computing middleware on parallel processing platforms running over high-speed networks and embedded system interconnects. He is the lead author of the books Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, published in 2000 by Wiley and Sons, and C++ Network Programming: Mastering Complexity with ACE and Patterns, published in 2002 by Addison-Wesley. He can be contacted at* ***schmidt@uci.edu****.*