**Dynamic CORBA, Part 1: The Dynamic Invocation Interface**

(Page [2](http://www.ddj.com/cpp/184403833?pgno=2) of [2](http://www.ddj.com/cpp/184403833?pgno=2))

**Object Interconnections: Dynamic CORBA, Part 1: The Dynamic Invocation Interface**

**by Douglas C. Schmidt and Steve Vinoski**

**Introduction**

Dynamic CORBA is a capability provided by CORBA that we have not covered in our column yet, and that readers ask us about occasionally. Despite the fact that the CORBA specification has defined dynamic invocation features ever since version 1.0 was published in 1991, those features have not been as widely used by CORBA developers as the static features, such as the stubs and skeletons generated automatically by an IDL compiler. There are several reasons for this:

* Most CORBA developers use statically typed programming languages, such as C++ and Java, and Dynamic CORBA can be awkward to use in such languages.
* The Dynamic CORBA features are less efficient than the static features.

Recently, however, there's been increased interest in Dynamic CORBA, due in part to the OMG standardizing mappings for two scripting languages, CORBAscript [[1]](http://www.ddj.com/cpp/184403833?pgno=2#1) and Python [[2]](http://www.ddj.com/cpp/184403833?pgno=2#2). It's also a natural outcome of developers continuing to apply CORBA to an ever-increasing number of varied and diverse problem domains.

In this column, we'll cover the basics of the DII (Dynamic Invocation Interface), the client-side interface used for dynamic CORBA applications. In future columns, we'll discuss the Dynamic Any, used to create and examine values in Dynamic CORBA applications; the DSI (Dynamic Skeleton Interface), the server-side counterpart to the DII; and the IFR (Interface Repository), a distributed service that provides run-time access to CORBA type information.

**Dynamic Invocation Basics**

When you compile an IDL file, the compiler typically generates a C++ stub file for the client and a C++ skeleton file for the server. For the client, you compile the stub file into your application to allow it to access CORBA objects. Likewise, for the server, you compile both the stub file and the skeleton file and link them with your servant implementations. [Figure 1](http://www.ddj.com/showArticle.jhtml?documentID=cujcexp2007vinoski&pgno=1) illustrates the general process.

It is probably no exaggeration to claim that *all* CORBA C++ programmers are familiar with stub-based CORBA client applications. The first sample program that ORB providers direct their C++ customers to work with is invariably a stub-based application. Books, papers, articles, and even newsgroup postings regarding CORBA and C++ rely almost exclusively on static stubs as the basis for their discussions. Given that C++ is a statically typed [programming language](http://www.ddj.com/cpp/184403833?pgno=2), this focus is not surprising, because static stubs are a natural fit for the way that C++ programmers write their applications.

There's another way to invoke requests on CORBA objects, however, without relying on static stubs. This approach involves the following steps:

1. Invoking a built-in operation on the object's interface to dynamically construct a request object.
2. Providing details for the request such as the operation name, argument types and values, and return type.
3. Calling an operation on the request object itself to invoke it, causing the request to be sent to the target CORBA object.

Although a static stub has type information compiled directly into it, applications must explicitly provide this information to dynamically constructed request objects. These extra steps are precisely the source of the difference in the names of these approaches: *static* implies that all information required to invoke the request is built in, whereas *dynamic* implies that such information must be supplied at run time.

**A Simple DII Use Case**

The DII allows a client to identify an operation via its string name and to "push" arguments onto a **CORBA::Request** stack. To illustrate how the DII works, we'll start by creating and invoking a DII request with no arguments and a **void** return type, as defined by the following simple IDL:

// IDL

interface A {

 void op();

};

All object references support the ability to create DII requests because the base **CORBA::Object** interface, inherited by all IDL interfaces, defines these operations. To create a DII request, therefore, the client must first have an object reference, as shown below:

// C++

CORBA::Object\_var obj = // ...obtain object reference...

CORBA::Request\_var req = obj->\_request ("op");

On the first line above, we initialize an object reference variable named **obj**. We do not show the details of this initialization, but we assume that it's done typically, such as by looking up an object reference in the Naming or Trading services or by invoking an object creation operation on a factory object. By passing in the name of the operation we want to invoke on the target object, we then use this object reference to create the **CORBA::Request** pseudo-object that represents our DII request.

To actually direct the **Request** to the target object, we first initialize it and then invoke it explicitly, as follows:

req->set\_return\_type (CORBA::\_tc\_void);

req->invoke ();

By default, a **Request** created in the manner we've shown does not have its arguments or its return type initialized, but it does have its argument count set to zero by default. Our operation **op** has no arguments, so we rely on this default. We do, however, have to explicitly set the **Request**'s return type using the static **TypeCode** constant for the **void** type. Finally, after initializing the **Request**, we invoke it on the target object by calling its **invoke** operation.

Let's look at the equivalent code for static invocation:

CORBA::Object\_var obj = // ...obtain object reference...

A\_var a\_obj = A::\_narrow (obj);

obj->op ();

There are three major differences between these static and dynamic invocation examples:

1. *Narrowing.* Due to static typing, the static code requires object-reference narrowing. Without narrowing, you won't be able to compile the code invoking the **op** operation. The code using the DII does not require narrowing, on the other hand, because all object references support the DII by virtue of the fact that they inherit from the base **CORBA::Object** interface.
2. *Request initialization.* In the static code, invoking **op** invokes a C++ method — the static stub code — that uses a private interface to the underlying ORB to send the request. All information about the request is compiled into the stub code, and any initialization it might require to get the ORB to send the request is hidden entirely from the application. In the dynamic invocation scenario, however, the application is completely responsible for supplying all the information required to complete the **Request** object. In our example, this difference is shown by the fact that we initialize the **Request** return type explicitly.
3. *Explicit invocation.* The static code implicitly invokes a request by calling a method on the static stub class. Generally, a stub's methods correspond to the operations defined in the IDL interface (and its base interfaces) supported by that stub. The DII has no specific operation names built into it since it's essentially a "generic stub." It therefore requires the application to explicitly invoke the **Request** via the general **invoke** operation.

**More Realistic DII Use Case**

For a more realistic example of DII, we'll go back to our old friend from past columns, the stock quote interface:

// IDL

module Stock {

 exception Invalid\_Stock {};

 interface Quoter {

 long get\_quote (in string stock\_name) raises (Invalid\_Stock);

 };

};

As shown below, invoking the **Stock::Quoter::get\_quote** operation through the DII is somewhat more complicated.

1 CORBA::Object\_var obj = // ...obtain object reference...

2 CORBA::Request\_var req = obj->\_request ("get\_quote");

3 req->add\_in\_arg () <<= "IONA";

4 req->set\_return\_type (CORBA::\_tc\_long);

5 req->exceptions ()->add (Stock::\_tc\_Invalid\_Stock);

6 req->invoke ();

7 CORBA::Environment\_ptr env = req->env ();

8 if (!CORBA::is\_nil (env) && env->exception () == 0) {

9CORBA::Long retval;

10 req->return\_value () >>= retval;

11 }

In this example, we create a **Request** object as in our first example. The rest of the example differs significantly from the first example, however, mainly due to the need to fully initialize the **Request**. We describe the necessary steps in detail below.

* *Line 3: argument initialization.* This line looks rather simple, but it does a lot behind the scenes. What line 3 does is add an input argument to the **Request**. In our stock quoter IDL, the **get\_quote** operation takes one argument of type string as an **in** argument. In our example code, we call the **add\_in\_arg** operation on the **Request**, which creates an uninitialized **in** argument as part of the **Request** and returns a C++ reference to that argument as a **CORBA::Any&**. On the same line, we use the **Any** insertion **operator <<=** to insert a string value into the newly-created argument. The net effect of this single line is to add a single typed **in** argument and its value to the **Request**.
* *Line 4: set the return type.* As in our first example, we call **set\_return\_type** on the **Request** to initialize the return type to that of the operation we're invoking. In this example, the return type is IDL **long**, so we use the static **TypeCode** for the **long** type for this initialization.
* *Line 5: list possible user exceptions.* Like line 3, the simple look of this line belies its actual complexity. On this line, we augment the **Request** with information about the user-defined exceptions that could be raised if the **Request** is invoked. Like IDL **struct**s, user-defined IDL exceptions can contain any number of data members of any type. Unlike CORBA system exceptions, which are all structurally identical, the structure of a user-defined exception is defined completely by the IDL developer. ORB implementations generally have built into them the ability to demarshal system exceptions, since they're all structured the same way, but they cannot know *a priori* how to demarshal a given user-defined exception. By adding to the **Request** structural information about each possible user-defined exception that might be raised, we are essentially giving the ORB run-time engine the information it needs to recognize each user-defined exception and demarshal it properly. In our example above, there's only a single user-defined exception that could result from an invocation of the **get\_quote** operation, so we add that to the **Request**'s exception list.
* *Line 6: invoke the operation.* Now that we've set up all the information needed to complete the **Request**, we invoke it.
* *Line 7-8: check for exceptions.* After **invoke** returns, we can check the **Request** to see whether the return was normal or exceptional. On line 7, we get the **CORBA::Environment** pseudo-object from the **Request**. In a DII **Request**, the **Environment** pseudo-object is used to contain exception information. On line 8, we perform a test to see if the **Environment** is valid and whether it holds an exception. If the **Environment\_ptr** is nil, or if its exception accessor method returns a null **CORBA::Exception** pointer, then the operation returned normally. DII implementations do not throw exceptions the way static stubs do, so **try**/**catch** blocks are not used with the DII.
* *Line 9-10: extract the result.* Now that we've determined that the operation returned normally, we can obtain the result. We do this by accessing the **CORBA::Any** representing the return value via the **Request::return\_value** function and then extracting the **CORBA::Long** return value from it using normal **Any** extraction.

Perhaps you've heard that using the DII is complicated, but as this example shows, it need not be. However, be aware that we've achieved some simplicity in our example by avoiding the issue of creating or using values based on complex IDL types. Our example uses only the built-in IDL **string** and **long** types, which both map to simple C++ types. If we had to pass a value of an IDL **struct** or **union** type, our example would be far more complex. We'll show such an example in our next column.

Another reason our example is so simple is that we assume that our application has built into it all the necessary information needed to create the DII request, such as the operation name, the number and types of the arguments, information about the user-defined exceptions, and the return type. In reality, if an application already has all this information built into it, there is little reason to use the DII. Dynamic applications do not normally possess such information, and thus they must obtain it elsewhere at run time. Applications needing dynamic access to such information normally obtain it via the IFR (Interface Repository). We'll cover the IFR in a future column and show how you use it for Dynamic CORBA applications.

**Deferred Synchronous Invocations**

So far our examples have assumed that the caller performs strict request/reply invocation using **Request::invoke**. The ORB's **invoke** implementation sends the request to the target object and then waits for the reply. The blocking semantics of **invoke** are therefore identical to that of an invocation through a static stub. Sometimes, however, applications want to avoid blocking and continue processing while awaiting responses from servers. The DII supports this capability via a *deferred synchronous invocation*.

When a client makes a deferred synchronous invocation, the ORB sends the request and allows the client to perform other work. The client can retrieve the response later when it needs the results. Changing our first example to use deferred synchronous invocation is easy:

// C++

CORBA::Object\_var obj = // ...obtain object reference...

CORBA::Request\_var req = obj->\_request ("op");

req->set\_return\_type (CORBA::\_tc\_void);

req->send\_deferred ();

This code creates the **Request** and sends it, which allows the application to continue processing without waiting for the response. To get the response, we simply call **get\_response**:

req->get\_response ();

The **get\_response** call blocks the caller until the response becomes available. To avoid blocking, we can call **poll\_response**:

if (req->poll\_response ())

 req->get\_response ();

The **poll\_response** operation returns true if the response is available, false otherwise. After calling **get\_response**, the application can examine the **Request** to see whether the return was normal or exceptional, exactly as we showed in our stock quoter example above.

The DII also supports the ability to call operations using **oneway** semantics via the **Request::send\_oneway** operation. (You can even use **send\_oneway** to invoke operations that are not defined as **oneway** in their IDL definitions, though this feature is seldom used.) You use **Request::send\_oneway** the same way you use **invoke** and **send\_deferred**, except that by definition, a **oneway** call normally has no response. If you want a response, you must set the **SyncScopePolicy** to either:

* **SYNC\_WITH\_SERVER** — With this option, the server sends a reply before it dispatches the request to the target object.
* **SYNC\_WITH\_TARGET** — This option is equivalent to a synchronous two-way CORBA operation (i.e., the client will block until the server ORB sends a reply after the target object has processed the operation).

In either case, you must call **get\_response** to ensure proper request processing. The **SyncScopePolicy** can be set by a client and uses new flags in the **response\_requested** field of the GIOP header. The server ORB checks this field to determine what type of a reply, if any, is required for a one-way invocation.

The deferred synchronous invocation feature of the DII was originally the only portable way to perform request invocations using anything other than the strict request/response model. There are a broader range of options now that the AMI (asynchronous method invocation) and TII (time-independent invocation) messaging capabilities have been added to CORBA [[3]](http://www.ddj.com/cpp/184403833?pgno=2#3). Many ORBs now support AMI, which is an efficient and elegant way of decoupling request invocations from their responses.

**Creating Requests**

All of the examples above use the **\_request** operation to create **Request** objects. This operation, which does not appear in the **CORBA::Object** pseudo-IDL definition, was originally added as a helper function in the first IDL C++ Language Mapping specification. The IDL Java Language Mapping later adopted the same function. The **\_request** function allows you to create an unpopulated **Request** based only on the name of the target operation.

Another way to create a **Request** requires you to supply most of the information for the request up front, rather than adding it to the **Request** after creating it. You do this by calling **\_create\_request** on an object reference. There are two forms of **\_create\_request**:

* The one that we recommend that you always use.
* The one based on the original CORBA 1.0 DII specification that we recommend you avoid because it doesn't support all the features needed for real-world DII applications.

We therefore show only the first form of **\_create\_request** here, as shown in the following signature:

// C++

void Object::\_create\_request (Context\_ptr operation\_context,

 const char \*operation\_name,

 NVList\_ptr argument\_list,

 NamedValue\_ptr result,

 ExceptionList\_ptr exception\_list,

 ContextList\_ptr context\_list,

 Request\_out request,

 Flags flags);

Below we rework our stock quoter example to use **\_create\_request** rather than **\_request**. First, we create an **NVList** via the ORB's **create\_list** operation and then populate it with the name of the stock we're interested in.

// C++

CORBA::NVList\_var nvlist;

orb->create\_list (1, nvlist.out ());

\*(nvlist->add (CORBA::ARG\_IN)->value ()) <<= "IONA";

This code creates an **NVList** of length **1**, sets the **ARG\_IN** flag for the input argument, and then accesses the **Any** to set the **string** argument using **Any** insertion.

Next, we create a single **NamedValue** to hold the operation's return value, again using the ORB's factory operation:

CORBA::NamedValue\_var result;

orb->create\_named\_value (result.out ());

result->value ()->replace (CORBA::\_tc\_long, 0);

This code creates the **NamedValue** to hold the return value and sets the appropriate **TypeCode** for that value using the static **TypeCode** constant for the IDL **long** type.

Next, we again use the ORB to create and populate an **ExceptionList** to hold information about the **Invalid\_Stock** user exception:

CORBA::ExceptionList\_var exc\_list;

orb->create\_exception\_list (exc\_list.out ());

exc\_list->add (Stock::\_tc\_Invalid\_Stock);

Now that we've set up the information needed to create the **Request**, we pass it all to the **\_create\_request** operation:

CORBA::Object\_var obj = // ...obtain object reference...

CORBA::Request\_var request;

obj->\_create\_request (CORBA::Context::\_nil (),

 "get\_quote",

nvlist,

result,

exc\_list,

CORBA::ContextList::\_nil (),

request.out (),

0);

request->invoke ();

The remainder of the code, needed to check for exceptions and examine the result, is the same as in the original example.

As our example shows, using **\_create\_request** can be more complicated than creating a **Request** using the **\_request** operation. This is because the **Request** interface supplies a number of short-cut operations that make setting arguments, exception types, and return types easier than setting them individually on the underlying **NVList**, **ExceptionList**, and **NamedValue**. We therefore recommend using the **\_request** operation in preference to **\_create\_request**.

**Concluding Remarks**

CORBA provides two different ways for clients to communicate with servers:

* The SSI (Static Invocation Interface) is provided by static stubs generated by a CORBA IDL compiler and is useful when client applications know the interface offered by the server at compile time.
* The DII (Dynamic Invocation Interface) is provided by an ORB's dynamic messaging mechanism and is most useful when client applications do not have compile-time knowledge of the interfaces offered by servers.

Many distributed applications can be written using CORBA's SII. However, an important and growing class of applications, such as interface browsers, network management applications, distributed visualization tools, debuggers, [configuration management](http://www.ddj.com/cpp/184403833?pgno=2) tools, and scripting languages, require the type of flexibility provided by Dynamic CORBA features like the DII. The DII enables applications to construct and invoke CORBA requests at run time by querying an Interface Repository. In addition, the DII is required for applications that use CORBA's deferred synchronous model of operation invocation, which decouples a request from the response so that other client activities can occur while the server is processing the response.

Our next column will focus on Dynamic Any. We'll show how to use Dynamic Any to create and manipulate complex arguments and return types in Dynamic CORBA applications. If you have comments, questions, or suggestions regarding Dynamic CORBA or our column, please let us know at **object\_connect@cs.wustl.edu**.

**References**

[1] Object Management Group. "CORBA Scripting Language, v1.0," OMG Document formal/01-06-05, June 2001.

[2] Object Management Group. "Python Language Mapping Specification," OMG Document formal/01-02-66, February 2001.

[3] Douglas C. Schmidt and Steve Vinoski. "Object Interconnections: An Introduction to CORBA Messaging," *C++ Report*, November/December 1998.

***Steve Vinoski*** *(<http://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 is also IONA's primary representative to the W3C (World Wide Web Consortium) Web Services Architecture Working Group.*

***Doug Schmidt*** *(<http://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****.*