**Dynamic CORBA, Part 4: The Interface Repository**

The Dynamic CORBA Interface Repository provides type information about CORBA interfaces and other entities defined in IDL. It helps ensure the type-safety and correctness of the DII and DSI mechanisms that defer an application's binding onto specific interface types until run time.

**Object Interconnections: Dynamic CORBA, Part 4: The Interface Repository**

**Douglas C. Schmidt and Steve Vinoski**

**Introduction**

Welcome to the next installment of our series covering Dynamic CORBA. In Part 1 [[1]](http://www.ddj.com/cpp/184403854#1), we discussed the DII (CORBA 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/184403854#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. In Part 3 [[3]](http://www.ddj.com/cpp/184403854#3), we covered the DSI (Dynamic Skeleton Interface), which is the server-side counterpart of the DII that enables CORBA developers to portably construct applications that cannot know *a priori* the types or identities of the objects they must serve.

In this column, we explain the IFR (Interface Repository), which is the next piece of the Dynamic CORBA puzzle. Our previous columns in this series have intentionally ignored the fact that Dynamic CORBA applications require a means of obtaining type information dynamically. To avoid cluttering our examples and explanations, we instead relied on hard-coded type information. Now that we've completed our coverage of the DII, DSI, and Dynamic Any, we'll show how CORBA developers can use the IFR to construct truly dynamic applications that discover all necessary type information at run time.

**IFR Basics**

CORBA developers are typically familiar with IDL compilers, which parse IDL definitions and generate stubs and skeletons to represent them. Such compilers are typically composed of:

* A *front end*, which performs lexical, syntactic, and semantic analysis of the IDL input while building a parse tree representing that input.
* One or more *back ends*, which walk the resulting parse tree and generate code based on its contents. For example, one back-end implementation might generate C++ stubs and skeletons, while another might generate Java stubs and skeletons.

The IFR is a distributed service that shares similarities with the inner workings of an IDL compiler. It allows applications to store and retrieve IDL definitions to and from an instance of the service, much like allowing them to modify and access an IDL parse tree at run time. For this column, we focus on the IFR operations that provide read access to IDL definitions, since the write operations are typically used only by tools used to populate and administer an IFR. ORBs that provide IFR implementations normally support tools that make use of these write interfaces to allow you to add definitions to the IFR (though most require you to remember to do so explicitly). Often, the tool that writes the IFR is a special back end to the IDL compiler that walks the internal parse tree and writes the definitions to the IFR.

There are several different ways to access the IFR's contents. We describe the two most useful approaches below:

1. *Initial reference*. If you pass the string **"InterfaceRepository"** to the **ORB::resolve\_initial\_references** method, it will return an object reference that you can narrow to the **CORBA::Repository** interface type. You can then use that interface to look up and navigate through all type definitions stored in the repository.
2. ***CORBA::Object*** *method*. The **get\_interface** methodof the **CORBA::Object** interface returns a **CORBA::InterfaceDef** object that provides access to the IDL definitions specific to the target object.

Dynamic CORBA applications typically use the **CORBA::Object::get\_interface** method because it provides more direct access to the type information needed for such applications. Since applications must be able to invoke **get\_interface** without knowing the specific type of the target object, this method appears on the **CORBA::Object** interface, meaning that it's available on all CORBA objects. The **InterfaceDef** returned from **get\_interface** describes the most derived interface supported by the target object. For example, if interface **B** derives from interface **A**, and a servant implements interface **B**, then **B** is the most derived interface supported by an object incarnated by the servant. By virtue of inheritance, the object also supports interface **A**, but **A** is not its most derived interface. If someone later adds an interface **C** derived from **B**, the most derived interface of the existing object is still **B**, because it does not implement **C**.

Since it inherits from several base interfaces, **InterfaceDef** is somewhat complicated. First we look at its **describe\_interface** method, which returns a **struct** that contains all the information about the target interface. To make our example concrete, this column will continue to use the Stock Quoter IDL shown below as our target interface:

// 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);

 };

};

Assuming we had an object reference to an instance of **Stock::Quoter**, **FullInterfaceDescription** **struct** returned from **describe\_interface** would contain the following information:

* Its simple name, e.g., **"Quoter"**.
* Its repository ID, e.g., **"IDL:Stock/Quoter:1.0"**. (We discuss repository IDs in more detail near the end of this column.)
* Its **TypeCode**, which for static programs is also available as the constant **Stock::\_tc\_Quoter**.
* Its version string.
* Its enclosing scope's repository ID.
* The repository IDs of its base interfaces.
* Sequences of full descriptions of its operations and attributes.

For example, let's assume we've retrieved a stringified object reference for a **Stock::Quoter** object from a file, so we can invoke its **get\_interface** method, as shown below:

CORBA::String\_var str = // read string from file

CORBA::Object\_var target =

 orb->string\_to\_object (str);

CORBA::InterfaceDef\_var intf\_def =

 target->\_get\_interface ();

Note that like other **CORBA::Object** operations, the **get\_interface** operations acquires a leading underscore when mapped to a C++ method to keep it from clashing with user-defined operations in derived classes. After we invoke **get\_interface**, we can access the full description of the **Stock::Quoter** interface by invoking **describe\_interface** on the returned **InterfaceDef** to obtain **FullInterfaceDescription**, as follows:

CORBA::InterfaceDef::FullInterfaceDescription\_var fid =

 intf\_def->describe\_interface ();

Continuing with the example above, we could print the names of all operations and attributes in the **Stock::Quoter** interface (or any interface for that matter) like this:

cout << "interface name: " << fid.name.in () << endl;

CORBA::ULong i;

for (i = 0; i < fid->operations.length (); ++i)

 cout << " operation " << i+1 << ": "

<< fid->operations[i].name.in () << endl;

for (i = 0; i < fid->attributes.length (); ++i)

 cout << " attribute " << i+1 << ": "

 << fid->attributes[i].name.in () << endl;

The **operations** member is a **sequence** of **OperationDescription**, and the **attributes** member is a **sequence** of **AttributeDescription**. The output of this program for **Stock::Quoter** is:

interface name: Quoter

 operation 1: get\_quote

**Invoking a DII Request**

The full descriptions of the operations and attributes that **InterfaceDef** contains are the key to obtaining the type information necessary to write true dynamic clients. In Parts 1 and 2 of this series, we used static type information in our examples since we hadn't discussed the IFR yet. We'll now revisit those examples to show how to get the necessary type information from the IFR. Our DII/**DynAny** example is based on the **Stock::Quoter** IDL definitions shown above. Our original DII/**DynAny** client looked like this:

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

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

req->add\_in\_arg () <<= "IONA"; // Note static string type.

// Note static TypeCode.

req->set\_return\_type (Stock::\_tc\_Info);

// Another static TypeCode.

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

req->invoke ();

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

if (CORBA::is\_nil (env) || env->exception () == 0) {

 DynamicAny::DynAny\_var dynany =

 dynfactory->create\_dyn\_any (req->return\_value ());

 ...

}

For a truly dynamic client, we must replace the static type information we use to set up the request arguments, return type, and exception types with type information dynamically retrieved from the IFR. One way to do this is to navigate explicitly through the full **OperationDescription** for the **get\_quote** operation. The following example builds on the **FullInterfaceDescription** retrieval code from the previous example.

 1 CORBA::Object\_var obj =

 2 // ...obtain object reference...

 3 CORBA::Request\_var req =

 4 obj->\_request ("get\_quote");

 5 for (CORBA::ULong i =

 6 0; i < fid->operations.length (); ++i) {

 7 if (strcmp (fid->operations[i].name,

 8 "get\_quote") == 0) {

 9 for (CORBA::ULong j = 0;

10 j <fid->operations[i].parameters.length ();

11 ++j) {

12 CORBA::Flags direction;

13 switch

14 (fid->operations[i].parameters[j].mode) {

15 case CORBA::PARAM\_IN:

16 direction = CORBA::ARG\_IN;

17 break;

18 case CORBA::PARAM\_OUT:

19 direction = CORBA::ARG\_OUT;

20 break;

21 case CORBA::PARAM\_INOUT:

22 direction = CORBA::ARG\_INOUT;

23 break;

24 }

25 CORBA::NamedValue\_ptr nv =

26 req->arguments ()->add (direction);

27 nv->value ()->replace

28 (fid->operations[i].parameters[j].type. in (),

29 0);

30 }

31 req->set\_return\_type

32 (fid->operations[i].result.in ());

33 for (CORBA::ULong j = 0;

34 j < fid->operations[i].exceptions.length ();

35 ++j)

36 req->exceptions ()->add

37 (fid->operations[i].exceptions[j].type.in ());

38}

39 }

Lines 5-8 look through the sequence of operation descriptors to find the element that describes the **get\_quote** operation. Using that element, lines 9-30 set the type and direction for each **Request** argument. Line 231 then sets the **Request** return type, and lines 33-38 set all the user-defined exceptions that the **Request** can return. After this code completes, all that's left to do is set the string value for the single argument to **get\_quote** using **DynAny**, which we'll show a little later.

Although the code above eliminates the dependency on static type information, it's complicated due to the deep nesting of the description **struct**s. Another approach is to use the **create\_operation\_list** method that the ORB provides to fill in the target operation's argument details:

 1 CORBA::Object\_var target =

 2 // ...obtain object reference...

 3 CORBA::InterfaceDef\_var intf\_def =

 4 target->\_get\_interface ();

 5 CORBA::Contained\_var ctd =

 6 intf\_def-> lookup ("get\_quote");

 7 CORBA::OperationDef\_var op\_def =

 8 CORBA::OperationDef::\_narrow (ctd.in ());

 9 CORBA::NVList\_var arglist;

10 orb->create\_operation\_list

11 (op\_def.in (), arglist.out ());

The **create\_operation\_list** method shown on lines 10-11 uses its **OperationDef** argument to fill in its **NVList** argument, which is an **out** parameter. (An actual implementation of **create\_operation\_list** inside the ORB might look a lot like the **FullInterfaceDescription** example, in fact). Lines 5-6 obtain **OperationDef** by looking up the entity named **"get\_quote"** in **InterfaceDef**. Note, however, that **create\_operation\_list** fills in only the types and directions of the arguments in the target operation's argument list and does not take care of the **Request** return type, context list, or exception list. A complete example using **create\_operation\_list** follows:

 1 CORBA::Object\_var target =

 2 // ...obtain object reference...

 3 CORBA::InterfaceDef\_var intf\_def =

 4 target->\_get\_interface ();

 5 CORBA::Contained\_var ctd =

 6 intf\_def-> lookup ("get\_quote");

 7 CORBA::OperationDef\_var op\_def =

 8 CORBA::OperationDef::\_narrow (ctd.in ());

 9 CORBA::NVList\_var arglist;

10 orb->create\_operation\_list (op\_def.in (),

11 arglist.out ());

12 CORBA::NamedValue\_var ret\_nv;

13 orb->create\_named\_value (ret\_nv.out ());

14 CORBA::TypeCode\_var ret\_type = op\_def->result ();

15 ret\_nv->value ()->replace (ret\_type.in (), 0);

16 CORBA::ExceptionList\_var exc\_list;

17 CORBA::ExceptionDefSeq\_var exc\_seq =

18 op\_def->exceptions ();

19 if (exc\_seq->length () > 0) {

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

21 for (CORBA::ULong i = 0; i < exc\_seq->length ();

22 ++i) {

23 CORBA::TypeCode\_var exc\_tc = exc\_seq[i]->type ();

24 exc\_list->add (exc\_tc.in ());

25 }

26 }

27 CORBA::Request\_var request;

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

29 "get\_quote", arglist.in (), ret\_nv.in (),

30 CORBA:ContextList::\_nil (),exc\_list.in (),

31 request.out (), 0);

This code is really no simpler than the previous example, but it reveals an important fact about the IFR: it supports multiple views of the information it contains. Specifically, the IFR allows navigation based on description data (as the previous example shows) or based on object references (as this example shows). Specifically, in this example, we first retrieve an **InterfaceDef** object reference on lines 3-4. From that we retrieve an **OperationDef** object reference via named-based lookup on lines 5-8. Lines 10-26 then invoke separate methods on **OperationDef** to retrieve each data item we need, which differs from the previous example that invokes one **OperationDef** method to return all the necessary data in one big data structure. Finally, lines 28-31 invoke the long form of the **create\_request** operation. We use the long form because we already have all the information necessary to create the request.

Note how lines 5-6 in this example use the **lookup** method to obtain **Contained**, which we then narrow to **OperationDef** in lines 7-8. The **Contained** interface and its counterpart **Container** interface serve as fundamental base interfaces in the IFR, allowing for navigation through the hierarchies of IDL definitions. In IDL, every entity is either a container or is something that can be put into a container. Some IDL entities are both. For example, **InterfaceDef** is a **Container** that holds definitions for operations, attributes, and types. It's also a **Contained** that can be stored within a **module** definition. Similarly, **ModuleDef** is also both a **Container** and a **Contained**, whereas **EnumDef** (used to describe an IDL **enum** type) is only a **Contained**. At the top level of the IFR hierarchy, the **Repository** interface (which is the type of interface returned by **ORB::resolve\_initial\_references("InterfaceRepository")**) does not correspond to an IDL construct but still derives from **Container** because it contains everything within the IFR.

Regardless of whether we use **create\_operation\_list** or explicitly construct the argument list ourselves using a full description **struct**, we must set the argument values. The C++ code below will work with either approach:

 1 CORBA::Object\_var obj =

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

 3 DynamicAny::DynAnyFactory\_var dynfactory =

 4 DynamicAny::DynAnyFactory::\_narrow (obj.in ());

 5 CORBA::ULong len = req->arguments ()->count ();

 6 for (CORBA::ULong i = 0; i < len; ++i) {

 7 CORBA::NamedValue\_ptr nv =

 8 req->arguments ()->item (i);

 9 if (nv->flags () == CORBA::ARG\_OUT)

10 continue;

11 CORBA::Any \*arg = nv->value ();

12 CORBA::TypeCode\_var tc = arg->type ();

13 DynamicAny::DynAny\_var dynany =

14 dynfactory->create\_dyn\_any\_from\_type\_code

15 (tc.in ());

16 switch (tc->kind ()) {

17 case CORBA::tk\_string:

18 dynany->insert\_string ("IONA");

19 break;

20

21 // handle other types

22 }

23 CORBA::Any\_var newany = dynany->to\_any ();

24 \*arg = newany.in ();

25 }

26 req->invoke ();

To set our arguments, lines 6-22 walk over the list of arguments created in the previous example, ignoring all **out** arguments. For each argument, lines 12-15 use **TypeCode** to create **DynAny**. Lines 16-24 then set the value of **DynAny** according to its type. Our example sets a hard-coded string value in **DynAny** for simplicity, though a real dynamic application would obtain its values from another application, a database or file, or a GUI. After setting all the arguments, line 26 finally invokes the request. For clarity, we separated the creation of the argument list from the setting of the argument values. In real application code, it would be more efficient and easier to set the argument values as you construct the argument list. Note also that our example shows only the handling of the string type because showing a complete solution would require much more code.

**The IFR and the DSI**

Now that we've illustrated how a dynamic client application can use the IFR to obtain and use type information, let's move the discussion to the server. As we explained in Part 3 of this column series [[3]](http://www.ddj.com/cpp/184403854#3), the DSI allows certain types of server applications, such as gateways or debugging intermediaries, to serve objects for which they have no compile-time type information. Such DSI applications thus require access to the IFR to handle requests properly.

When it is up-called by the POA, a DSI servant must determine the type and identity of the target object for which it's being invoked. Based on the target object's type, it can query the IFR to obtain type information about the arguments of the operation being invoked. It then passes this argument type information back to the ORB runtime through the **ServerRequest** interface to allow the runtime to demarshal the input arguments properly. The following example illustrates these steps within the DSI servant's **invoke** method:

 1 CORBA::Object\_var obj =

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

 3 PortableServer::Current\_var current =

 4 PortableServer::Current::\_narrow (obj.in ());

 5 CORBA::Object\_var target = current->get\_reference ();

 6 CORBA::InterfaceDef\_var intf\_def =

 7 target->\_get\_interface ();

 8 CORBA::Contained\_var ctd =

 9 intf\_def-> lookup (server\_request->operation ());

10 CORBA::OperationDef\_var op\_def =

11CORBA::OperationDef::\_narrow (ctd.in ());

12 CORBA::NVList\_var arglist;

13 orb->create\_operation\_list (op\_def.in (),

14 arglist.out ());

15 server\_request->arguments (arglist.inout ());

Lines 1-5 obtain an object reference for the target object via the POA **Current**. Lines 6-7 then use the object reference and the name of the invoked operation (obtained from **ServerRequest**) to obtain an **OperationDef** for the target operation. Lines 13-24 pass **OperationDef** to **create\_operation\_list** to create an argument list for the target operation, which we pass into the **ServerRequest::arguments** method to obtain the arguments passed from the caller on line 15. After our arguments are available, we process them using **DynAny** as shown in [[3]](http://www.ddj.com/cpp/184403854#3). As you can see, **create\_operation\_list** and **OperationDef** make it fairly straightforward to implement truly dynamic DSI servants.

**Repository IDs**

Ever wonder about those strange repository ID strings that appear in various parts of CORBA? The IFR is the main reason those repository IDs exist. As its name suggests, a repository ID identifies an entity in the IFR. There are multiple formats allowed for repository IDs, such as Java RMI hashed identifiers, DCE UUID identifiers, and support for custom application-specific formats, but the default is the familiar **"IDL:identifier:version"** format. For example, the default repository ID for our **Stock::Quoter** interface is:

IDL:Stock/Quoter:1.0

The entity identifier is formed by replacing all instances of **::** in the fully-scoped IDL name with the **/** separator, and setting the version defaults to **1.0**. You can change the entity identifier and the version portions of a repository ID in your IDL definition using **#pragma prefix** or **typeprefix** directives. For example, all definitions in OMG specifications have the **"omg.org"** prefix:

#pragma prefix "omg.org"

module CORBA {

 interface InterfaceDef { ... };

};

Without the prefix, the repository ID for the **InterfaceDef** interface would be:

IDL:CORBA/InterfaceDef:1.0

With the **#pragma prefix** directive in effect, the repository ID becomes:

IDL:omg.org/CORBA/InterfaceDef:1.0

Using such prefixes helps reduce the chances of name clashes among repository IDs for types defined by separate organizations. To modify the version number in a repository ID, we use the **#pragma version** directive:

module CORBA {

 typeprefix CORBA "omg.org"

 interface InterfaceDef { ... };

#pragma version InterfaceDef 3.0

};

The parameters to the **#pragma version** directive are the scoped name of the entity to be versioned, and the version number in **<major>.<minor>** format. (Caveats regarding the versioning of IDL definitions are explained elsewhere [[4]](http://www.ddj.com/cpp/184403854#4)). Note also that in this example we use the new **typeprefix** IDL keyword to attach the **"omg.org"** prefix to the **CORBA** module definition, rather than using **#pragma prefix**. This new keyword promotes the importance of prefixes and helps eliminate ambiguities regarding the scope and application of **#pragma prefix**, especially when one IDL file includes another. With the **#pragma version** directive and **typeprefix** keywordin effect, the repository ID for our example becomes:

IDL:omg.org/CORBA/InterfaceDef:3.0

Finally, you can use the **#pragma ID** directive or the new **typeid** keyword to set the whole repository ID, as follows:

module A {

 interface B { ... };

#pragma ID B "LOCAL:module(A)/interface(B)"

};

Both the **#pragma ID** directive and the **typeid** keyword must be followed by a scoped name and an identifier string. This example shows the use of a **LOCAL** repository ID format for the identifier. The CORBA specification includes the **LOCAL** format to allow applications to support their own custom repository ID strings. For this format, the identifier can be any string you want it to be as long as it follows the **"LOCAL:"** format identifier. Despite official CORBA support for it, the **LOCAL** format is rarely used in practice.

If we know the repository ID of an entity, we can easily look it up in an IFR. The **Repository** interface we mentioned above (i.e., the one that represents the outermost container in the IFR) provides the **lookup\_id** operation for searching the IFR based on repository ID. The following example illustrates how to use the repository ID for this purpose:

 1 CORBA::Object\_var obj =

 2 orb->resolve\_initial\_references

 3 ("InterfaceRepository");

 4 CORBA::Repository\_var repos =

 5 CORBA::Repository::\_narrow (obj.in ());

 6 CORBA::Contained\_var contained =

 7 repos->lookup\_id ("IDL:Stock/Quoter:1.0");

 8 CORBA::InterfaceDef\_var intf\_def =

 9 CORBA::InterfaceDef::\_narrow (contained.in ());

10 // intf\_def now refers to the interface definition

11 // for Stock::Quoter

Lines 1-5 obtain a **Repository** reference by passing **"InterfaceRepository"** to **resolve\_initial\_references** and narrowing the return value appropriately. Lines 5-6 then invoke **lookup\_id**, passing in the default repository ID for the **Stock::Quoter** interface. The **lookup\_id** operation returns **Contained**, which we narrow to **InterfaceDef** in lines 8-9. Note that this narrowing is based on static knowledge that **Stock::Quoter** is an interface, but we could avoid that by asking **Contained** what it is, as follows:

 // Lines 1-7 same as above.

 8 if (contained->def\_kind () == CORBA::dk\_Interface) {

 9 CORBA::InterfaceDef\_var intf\_def =

10 CORBA::InterfaceDef::\_narrow (contained.in ());

11 // intf\_def now refers to the interface

12 // definition for Stock::Quoter

13 }

The **def\_kind** operation call on line 8 returns **DefinitionKind**, which is an **enum** that indicates what type of definition **Contained** refers to. Having verified that **Contained** actually does refer to an **interface** type, on lines 9-10 we safely narrow **Contained** to **InterfaceDef**.

**Evaluating the IFR**

When used in conjunction with other Dynamic CORBA features, such as the DII and DSI, the CORBA IFR supports powerful and type-safe metaprogramming capabilities [[5]](http://www.ddj.com/cpp/184403854#5). CORBA applications can query an IFR to obtain metadata that describes IDL interface types, operation signatures, operation arguments and return types, and the definition of user-defined data types. DII clients and DSI servers can use the metadata provided in an IFR to construct "generic" applications whose behavior can be determined at run time (e.g., via an interpretive language like CorbaScript [[6]](http://www.ddj.com/cpp/184403854#6)). These generic applications can also be used to reduce the effort required to develop bridges that translate requests from an ORB in one domain to other ORBs in a different domain or other systems running different middleware technologies, such as COM or Java RMI.

For example, a generic CORBA bridge can use the DII, DSI, and IFR to process a two-way operation as follows:

* Upon receiving a request, use the DSI API and IFR to decompose the request into a name/value list (**NVList**), which stores each argument in a separate Any data structure.
* Use the resulting **NVList** to construct a DII request and use the DII API to send this request to the target object.
* After receiving the DII reply, **NVList** holds the **out** arguments, so transfer it and the return value, or any raised exception, back into the DSI **ServerRequest** and reply to the original DSI request.

Such a bridge could serve, for example, as a generic dynamic load balancer, forwarding requests to the least loaded server among a set of replicas [[7]](http://www.ddj.com/cpp/184403854#7). Achieving this sort of flexibility using statically compiled CORBA stubs and skeletons would be impractical.

Although the IFR specification is a powerful service that occupies a large portion of the CORBA standard, surprisingly few CORBA developers actually use the IFR in practice. The following are some of the key reasons:

* **Complexity**. It should be clear from the discussion above that a generic ORB bridge can incur significant overhead, both in terms of time/space utilization and programming effort. For example, Dynamic CORBA operations and the IFR may require multiple data copies (e.g., copying the data from the request buffer into each one of the **Any**s of the DSI argument list and then later copying the reply from the DSI argument list into the DII request buffer). Copies are also required from the DII reply buffer into the **Any**s used to extract each argument and finally into the DSI reply buffer. There's also the overhead of having to make one or more expensive remote calls to the IFR to obtain the type information needed just to invoke the original desired request. Likewise, the programming APIs for the IFR are large and complicated, which makes them hard to use correctly. Our examples in this column provide only a sample of the complexity involved in navigating the IFR data structures and interfaces.
* **Consistency management**. The IFR holds metadata about objects separately from the objects themselves. Although this design saves human and computer resources for applications that don't use an IFR, it also allows the IFR to easily get out of sync with the objects it is supposed to describe. It also requires you to remember to populate the IFR with your IDL definitions if you intend to use your objects in Dynamic CORBA applications. An alternative way of providing an IFR-like service would be to have the objects themselves provide their own metadata in a reflective manner. For example, rather than having **Object::get\_interface** retrieve information from a separate IFR, the object's skeletons could return the information directly based on their own compile-time type information. This would ensure that the metadata would never be inaccurate or missing — if you can get to the object, then you are assured that you can also get to its metadata. Naturally, this approach would have increased the space utilization for applications that don't use the IFR, which helps explain why the OMG didn't mandate the tighter coupling between objects and their metadata.

**Concluding Remarks**

CORBA applications based on static stubs and skeletons have been used successfully in domains such as telecommunications, aerospace, process automation, and e-commerce. Static stubs and skeletons are not well suited, however, to shielding developers from the effects of changes to requirements or environmental conditions that occur late in an application's lifecycle (i.e., during deployment and/or at run time). For example, application developers may need to interact with objects whose interfaces did not exist when a distributed application was deployed initially. With Static CORBA, applying these types of changes requires tedious and error-prone redesign and reimplementation of existing application software. For applications such as interactive test programs or intermediary bridges that must be independent of the objects they interact with, Static CORBA simply does not suffice.

To address the limitations of Static CORBA outlined above, this column described the Dynamic CORBA IFR, which is a service that provides type information about CORBA interfaces and other entities defined in IDL. As described in our previous columns on Dynamic CORBA, one purpose of the DII and DSI mechanisms is to defer an application's binding onto specific interface types until run time. The IFR helps ensure the type-safety and correctness of these run-time bindings.

Our next column will discuss Portable Interceptors, which are yet another kind of Dynamic CORBA feature that an ORB invokes in the path of an operation invocation to monitor or modify the behavior of the invocation without changing the client or server application. If you have comments, questions, or suggestions regarding Dynamic CORBA or our column in general, please let us know at object\_connect@cs.wustl.edu.

We'd like to thank Jeff Parsons for his helpful comments that improved the form and content of this column.

**References**

[1] Steve Vinoski and Douglas C. Schmidt. "Object Interconnections: Dynamic CORBA, Part 1: The Dynamic Invocation Interface," *C/C++ Users Journal C++ Experts Forum*, July 2002, [<www.cuj.com/experts/2007/vinoski.htm>](http://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 C++ Experts Forum*, September 2002, [<www.cuj.com/experts/2009/vinoski.htm>](http://www.cuj.com/experts/2009/vinoski.htm).

[3] Steve Vinoski and Douglas C. Schmidt. "Object Interconnections: Dynamic CORBA, Part 3: The Dynamic Skeleton Interface," *C/C++ Users Journal C++ Experts Forum*, November 2002, [<www.cuj.com/experts/2011/vinoski.htm>](http://www.cuj.com/experts/2011/vinoski.htm).

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

[5] Nanbor Wang, Douglas C. Schmidt, Ossama Othman, and Kirthika Parameswaran. "Evaluating Meta-Programming Mechanisms for ORB Middleware," *IEEE Communication Magazine*, October 2001.

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

[7] Ossama Othman, Carlos O'Ryan, and Douglas C. Schmidt. "An Efficient Adaptive Load Balancing Service for CORBA," *IEEE Distributed Systems Online*, March 2001.

**About the Author**

[**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++*](http://cseng.aw.com/book/0%2C%2C0201379279%2C00.html), 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**](http://www.cs.wustl.edu/~schmidt/) is a full professor at Vanderbilt University. His research focuses on patterns, optimization principles, model-based software development, 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*](http://www.cs.wustl.edu/~schmidt/POSA/), published by Wiley and Sons, and [*C++ Network Programming: Mastering Complexity with ACE and Patterns*](http://www.cs.wustl.edu/~schmidt/ACE/book1/) and [*C++ Network Programming: Systematic Reuse with ACE and Frameworks*](http://www.cs.wustl.edu/~schmidt/ACE/book2/) both published by Addison-Wesley. He can be contacted at schmidt@isis-server.isis.vanderbilt.edu.