**Object Interconnections: Dynamic CORBA, Part 2 - Dynamic Any**

*Douglas C. Schmidt and Steve Vinoski*

Like their static counterparts, Dynamic CORBA applications manipulate real-world complex data types, but without having the types' compile-time information. This column shows how to use CORBA's Dynamic Any feature to create, examine, and modify data values of any IDL type in Dynamic CORBA applications.

**Introduction**

Welcome to the second part of our series covering Dynamic CORBA. In Part 1 [[1]](http://www.ddj.com/cpp/184403840#1), we explained the basics of the CORBA DII (Dynamic Invocation Interface). The DII allows CORBA clients to invoke operations on target objects without having compile-time knowledge of the targets' interfaces. The DII is not widely used in C++ or Java applications, but it's indispensable for those applications that do use it, such as scripting applications or application gateways.

Our presentation in Part 1 showed two examples of using the DII to invoke simple operations. The first example explained how to invoke an operation with no arguments or return value. The second example was slightly more complicated than the first and showed how to use the DII to invoke a stock quote operation. We kept the arguments and return value of the stock quote operation simple so we could focus on the fundamentals of using the DII. That wasn't the only reason, however — we also kept the arguments and return value simple out of necessity. Without the compile-time type information normally provided by C++ stubs, our DII example was unable to create or examine values of constructed IDL types, at least by conventional C++ means.

Real-world IDL operations often include arguments of constructed types, such as **struct**s, **union**s, and **sequence**s. In this column, we explain the Dynamic Any, which is the standard facility for manipulating values of such types within Dynamic CORBA applications. The CORBA Dynamic Any facility has undergone several revisions — this column describes the latest version.

**Why We Need the Dynamic Any**

Let's change our stock quote example slightly to introduce more features into its 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);

 };

};

Rather than simply returning the current value of the requested stock, **get\_value** now returns a **struct** containing the stock's highest price, lowest price, and current price for the day. The following code shows how we can use the DII to invoke this new version of **get\_quote**.

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

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

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

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

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

req->invoke ();

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

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

 Stock::Info retval;

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

 ...

}

This code works, but it isn't fully dynamic. It uses the static **TypeCode** constants **Stock::\_tc\_Info** and **Stock::\_tc\_Invalid\_Stock** to populate the DII request, and it uses the static **Stock::Info** type to extract the return value. If we have this much compile-time information about the **Stock** module, we might as well forego using the DII and just use the conventional CORBA SII (static invocation interface) since it's more efficient and less error prone.

As we noted in our previous column, we can replace the use of the static **TypeCode** constants with information retrieved from the CORBA IFR (Interface Repository). We're not going to show that in this column, however. We'll cover it later in this series of columns when we explore the IFR.

While the IFR can help us avoid the need for static **TypeCode** constants, it can't help us entirely with the **get\_quote** return value. We can get the *type* of the return value from the IFR, but we can't examine the return *value* itself using IFR facilities. Moreover, without having the type of the return value compiled into our program, we can't make use of **Any** extraction operators either.

The Dynamic Any facility provides the solution to allow us to examine the **get\_quote** return value. It allows a CORBA application to create, examine, or modify a value of any IDL type without having compile-time information for that type. This capability is essential for truly dynamic CORBA applications.

**Dynamic Any Basics**

The operation of the Dynamic Any hinges on a single fundamental characteristic of all constructed IDL types: they are all ultimately composed of "atomic" basic IDL types, such as **long**, **short**, **boolean**, and **char**. The Dynamic Any allows an application to create or modify a value of a constructed type by individually setting each of the basic values that comprise it. Similarly, an application can examine a value of a constructed type by examining each of its basic "atomic" values. As we'll show below, when creating a Dynamic Any, you pass in a **TypeCode**, either directly or indirectly as part of an **Any**. The Dynamic Any uses this **TypeCode** as the source of structural information about the value it stores. Unlike a regular **Any**, where you can change both its value and its **TypeCode** over the course of its lifetime, once you create a Dynamic Any, its **TypeCode** remains invariant.

For example, our **Stock::Info** **struct** is composed of three basic elements, each of type **long**. A sequence of **Stock::Info** **struct**s would be composed of multiple **struct** elements, each of them composed of three **long** values. Suppose you defined another **struct** that had a **Stock::Info** as one of its members, like this:

// IDL

module Stock {

 struct Report {

 string analyst\_brief;

CORBA::StringSeq inside\_trading\_cronies;

 Info price\_info;

 };

};

A **Stock::Report** has as its basic elements a **string**, a **sequence** of **string**s, as well as the three **long** members of the **Stock::Info** type.

**Creating a Dynamic Any**

To create a Dynamic Any, you use a **DynamicAny::DynAnyFactory**. You first retrieve a reference to such a factory from the ORB using **ORB::resolve\_initial\_references**, as shown below.

CORBA::Object\_var obj =

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

DynamicAny::DynAnyFactory\_var dynfactory =

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

Using a **DynAnyFactory**, you can create a Dynamic Any from either a **TypeCode** or from an **Any**. These two approaches for creating Dynamic Anys correspond to the two modes of using them: you use the **TypeCode** creation approach when using a Dynamic Any to *compose* a value, and you use the **Any** creation approach when using a Dynamic Any to *decompose* a value. You compose or create a value in a Dynamic Any by initializing each of its basic elements. If on the other hand you already have a value, you can use Dynamic Any to decompose it into its basic elements, each of which you can examine or change. Our **get\_quote** operation's return value is easily accessible in the form of an **Any**, so we'll create a **DynamicAny::DynAny** using that **Any**, as shown below.

// ...same code as in previous get\_quote example...

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 ());

 ...

}

Once we've created the Dynamic Any, we can choose to deal with the value either through the base **DynamicAny::DynAny** interface or through the derived interface corresponding to the value's type, in this case **DynamicAny::DynStruct**. We'll show both approaches. Our first example uses the **DynAny** interface to examine each of the three **long** values in the **Stock::Info** **struct**.

long high = dynany->get\_long ();

dynany->next ();

long low = dynany->get\_long ();

dynany->next ();

long current = dynany->get\_long ();

This example clearly illustrates the nature of a Dynamic Any as a sequence of basic types that can be accessed via an internal iterator. We examine all the basic values making up the **Stock::Info** value by repeatedly retrieving a **long** value and then advancing the internal iterator of the **DynAny** to "point" to the next **long** value.

The fact that our **DynAny** holds a value of a **struct** type is not explicit in this code, but nevertheless this example still contains implicit static assumptions about the contained value's type. Specifically, it assumes that the value in the **DynAny** is composed of three basic elements of type **long**. There are two ways to solve this issue: we can either write more general code based on the **DynAny** interface, or we can use the **DynStruct** interface as we mentioned above. To use **DynStruct**, we have to narrow to it from **DynAny**:

DynamicAny::DynStruct\_var dynstruct =

DynamicAny::DynStruct::\_narrow (dynany);

if (!CORBA::is\_nil (dynstruct)) {

 // narrow succeeded

 ...

}

We can ensure that the **narrow** will succeed by checking the value's type using the **DynAny::type** operation:

CORBA::TypeCode\_var typecode = dynany->type ();

if (typecode->kind () == CORBA::tk\_struct) {

 DynamicAny::DynStruct\_var dynstruct =

 DynamicAny::DynStruct::\_narrow (dynany);

 assert (!CORBA::is\_nil (dynstruct));

 ...

}

Once we have a **DynStruct**, we can use its specialized interface to access the **struct** members. We can either access the members via accessor methods similar to those in the **DynAny** interface, or we can take a "batch" approach by asking for a sequence of name/value pairs representing all member values of the **struct**. In the following example, we determine the name and type of each member using the **DynStruct** accessor approach:

DynamicAny::DynStruct\_var dynstruct =

 DynamicAny::DynStruct::\_narrow (dynany);

cout << "This is a struct with "

<< dynstruct->component\_count ()

<< " members:\n";

do {

 CORBA::String\_var name =

 dynstruct->current\_member\_name ();

 cout << "member name: " << name;

 switch (dynstruct->current\_member\_kind ()) {

 case CORBA::tk\_long:

 cout << ", type: long, value: " << dynstruct->get\_long ();

 break;

 // similar for other type cases

}

 cout << '\n';

} while (dynstruct->next ());

The form of the loop above will work for any **struct** type, regardless of how many members it has or what types they are. The **next** operation returns false once all members have been accessed. Note that **current\_member\_name** and **current\_member\_kind** are **DynStruct** operations, whereas **component\_count**, **get\_long**, and **next** are operations on the base **DynAny** interface. A similar iterative approach can be accomplished using the **DynAny** interface, but it can provide only values and not member names.

Alternatively, by calling the **DynStruct::get\_members** method, you can use the batch approach to obtain a sequence of name/value pairs representing all **struct** members. This method returns a **NameValuePairSeq**, which is a sequence of the following structure:

// IDL

module DynAny {

 typedef string FieldName;

 struct NameValuePair {

 FieldName id;

 any value;

 };

 ...

};

Using this function, you can write a loop similar to the one in the example above as follows:

DynamicAny::DynStruct\_var dynstruct =

 DynamicAny::DynStruct::\_narrow (dynany);

NameValuePairSeq\_var nvseq = dynstruct->get\_members ();

const CORBA::ULong count = nvseq->length ();

cout << "This is a struct with "

<< count

<< " members:\n";

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

 cout << "member name: " << nvseq[i].id;

 switch (nvseq[i].value.kind ()) {

 case CORBA::tk\_long:

 {

 CORBA::Long val;

 nvseq[i].value >>= val;

cout << ", type: long, value: " << val;

 }

 break;

 // similar for other type cases

 }

 cout << '\n';

}

As you can see, the batch approach and the accessor approach are quite similar to each other.

**Creating Dynamic Any Values**

You can also use the Dynamic Any to create or modify values, using either the modifier functions for basic types as provided by the base **DynAny** interface, or by using specialized modifier functions provided by derived interfaces, such as **DynStruct**. For example, to set each of the basic values in a **DynAny** holding a **Stock::Info** **struct** value, you can use the **insert\_long** operation as shown below.

dynany->insert\_long (12.40); // set the stock's high value

dynany->next ();

dynany->insert\_long (10.80); // set the stock's low value

dynany->next ();

dynany->insert\_long (11.20); // set the stock's current value

Naturally, all the iterative and recursive capabilities described above for examining values in a Dynamic Any can be applied to modify its values as well.

**Concluding Remarks**

In this column, we introduced the Dynamic Any, which is useful for creating and manipulating values of constructed IDL types in Dynamic CORBA programs. We scratched the surface by focusing on the **DynAny** and **DynStruct** interfaces. However, CORBA specifies a specialized Dynamic Any interface derived from **DynAny** for each different type, such as **DynEnum**, **DynUnion**, **DynSequence**, and **DynArray**. Each specialized Dynamic Any interface has its own way to deal with values of its particular type. Rather than try to cover all of these interfaces in this column, we suggest downloading the freely available Dynamic Any chapter from the book *Advanced CORBA Programming with C++* [[2]](http://www.ddj.com/cpp/184403840#2) from <www.triodia.com/staff/michi/advanced\_corba/chapter\_17.pdf>, since it provides full and detailed coverage of the entire **DynAny** interface hierarchy.

Our next column will focus on the DSI (Dynamic Skeleton Interface), which is the server-side counterpart to the DII. The DSI allows you to write [server applications](http://www.ddj.com/cpp/184403840) using servants that dynamically determine the interface definitions of the objects they're incarnating. If you have comments, questions, or suggestions regarding Dynamic CORBA or our column, please let us know at **mailto:object\_connect@cs.wustl.edu**.

**References**

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

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

***Steve 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*** *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****.*