# *Mapping: OLE Automation and CORBA* 17

This chapter describes the bidirectional data type and interface mapping between OLE Automation and CORBA.

Microsoft's Object Description Language (ODL) is used to describe Automation object model constructs. However, many constructs supported by ODL are not supported by Automation. Therefore, this specification is confined to the Automation-compatible ODL constructs.

As described in the Interworking Architecture chapter, many implementation choices are open to the vendor in building these mappings. One valid approach is to generate and compile mapping code, an essentially static approach. Another is to map objects dynamically.

Although some features of the CORBA-Automation mappings address the issue of inverting a mapping back to its original platform, this specification does not assume the requirement for a totally invertible mapping between Automation and CORBA.

## *Contents*

This chapter contains the following sections.

## 17.1 Mapping CORBA Objects to OLE Automation

### 17.1.1 Architectural Overview

There are seven main pieces involved in the invocation of a method on a remote CORBA object: the OLE Automation Controller; the COM Communication Infrastructure; the OLE system registry; the client-side Automation View; the operation's type information; the Object Request Broker; and the CORBA object's implementation. These are illustrated in Figure 17-1 (the call to the Automation View could be a call in the same process).

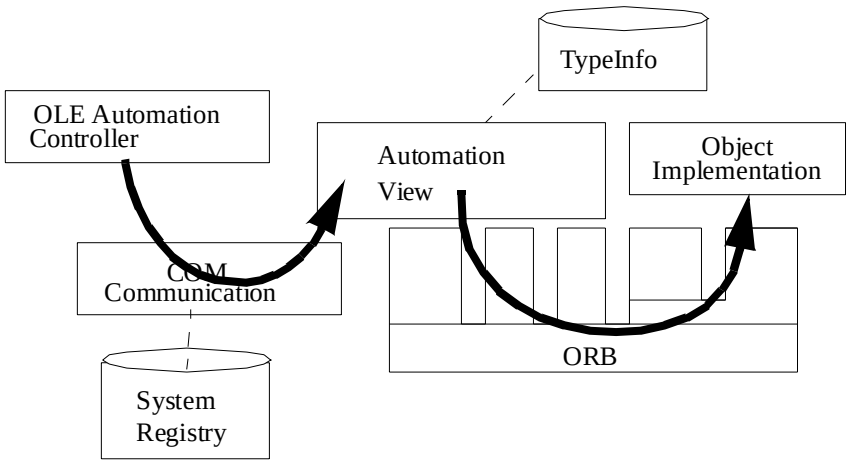


*Figure 17-1* CORBA Object Architectural Overview

The Automation View is an OLE Automation server with a dispatch interface that is isomorphic to the mapped OMG IDL interface. We call this dispatch interface an Automation View Interface. The Automation server encapsulates a CORBA object reference and maps incoming OLE Automation invocations into CORBA invocations on the encapsulated reference. The creation and storage of the type information is not specified.

There is a one-to-one correspondence between the methods of the Automation View Interface and operations in the CORBA interface. The Automation View Interface's methods translate parameters bidirectionally between a CORBA reference and an OLE reference.

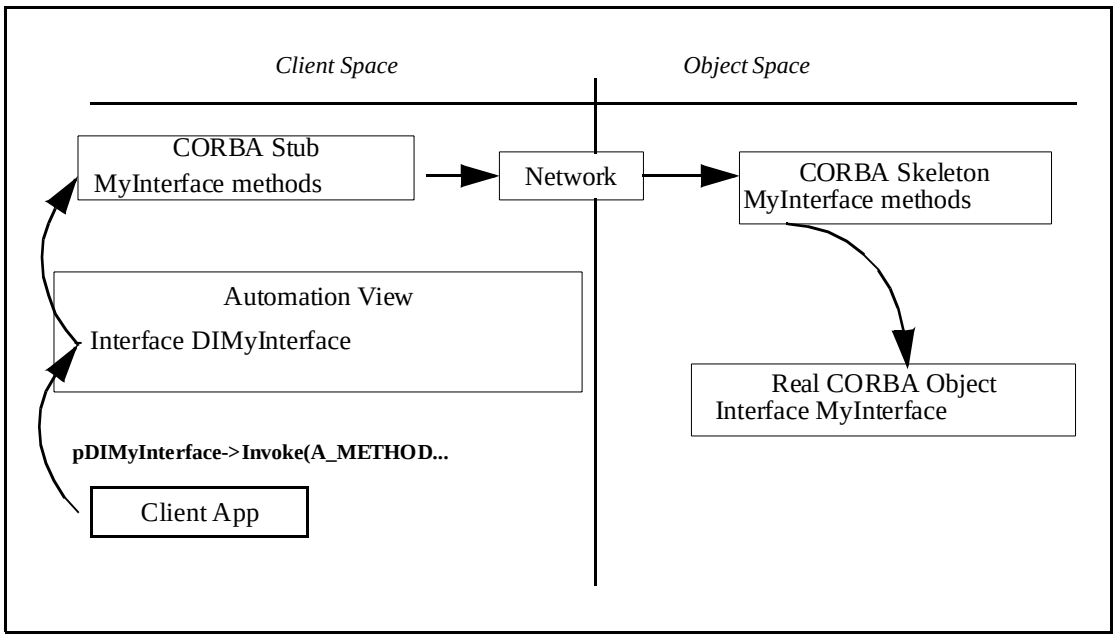


*Figure 17-2* Methods of the Automation View Interface delegate to the CORBA Stub

### *17.1.2 Main Features of the Mapping*

- OMG IDL attributes and operations map to Automation properties and methods respectively.
- OMG IDL interfaces map to Automation interfaces.
- The OMG IDL basic types map to corresponding basic types in Automation where possible. Since Automation supports a limited set of data types, some OMG IDL types cannot be mapped directly. Specifically:
  - OMG IDL constructed types such as structs and unions map to Automation interfaces with appropriate attributes and operations. User exceptions are mapped in the same way.
  - OMG IDL unsigned types map as closely as possible to Automation types, and overflow conditions are identified.
- OMG IDL sequences and arrays map to Automation Safearrays.

### *17.1.3 Mapping for Interfaces*

A CORBA interface maps in a straightforward fashion to an Automation View Interface. For example, the following CORBA interface

```
module MyModule // OMG IDL
{
interface MyInterface
{
// Attributes and operations;
...
};
};
```

maps to the following Automation View Interface:

```
[odl, dual, uuid(...)]
interface DIMyModule_MyInterface: IDispatch
{
// Properties and methods;
...
};
```

The interface **`IMyModule_account`** is an OLE Automation Dual Interface. A Dual Interface is a COM vtable-based interface which derives from IDispatch, meaning that its methods can be late-bound via **`IDispatch::Invoke`** or early-bound through the vtable portion of the interface. Thus, **`IMyModule_account`** contains the methods of IDispatch as well as separate vtable-entries for its operations and property get/set methods.

## *Mapping for Attributes and Operations*

An OMG IDL operation maps to an isomorphic Automation operation. An OMG IDL attribute maps to an ODL property, which has one method to *get* and one to *set* the value of the property. An OMG IDL readonly attribute maps to an OLE property, which has a single method to get the value of the property.

The order of the property and method declarations in the mapped Automation interface follows the rules described in “Ordering Rules for the CORBA->OLE Automation Transformation” part of “Detailed Mapping Rules” on page 15-13.

For example, given the following CORBA interface,

```
interface account // OMG IDL
{
attribute float balance;
readonly attribute string owner;
void makeLodgement(in float amount, out float balance);
void makeWithdrawal(in float amount, out float balance);
};
```

the corresponding Automation View Interface is:

```
[odl, dual, uuid(...)]
interface DIaccount: IDispatch
{                  // ODL
   HRESULT makeLodgement ([in] float amount,
                     [out] float * balance,
                     [optional, out] VARIANT * excep_OBJ);
   HRESULT makeWithdrawal ([in] float amount,
                     [out] float * balance,
                     [optional, out] VARIANT * excep_OBJ);
   [propget] HRESULT balance ([retval,out] float *
                [IT_retval];
   [propput] HRESULT balance ([in] float balance);
   [propget] HRESULT owner ([retval,out] BSTR * IT_retval);
}
```

OMG IDL **`in`**, **`out`**, and **`inout`** parameters map to ODL **`[in]`**, **`[out]`**, and **`[in,out]`** parameters, respectively. “Mapping for Basic Data Types” on page 17-9, explains the mapping for basic data types. The mapping for CORBA oneway operations is the same as for normal operations.

An operation of a Dual Interface always returns HRESULT, but the last argument in the operation’s signature may be tagged **`[retval,out]`**. An argument tagged in this fashion is considered syntactically to be a return value. Automation controller macro languages map this special argument to a return value in their language syntax. Thus, a CORBA operation’s return value is mapped to the last argument in the corresponding operation of the Automation View Interface.

#### *Additional, Optional Parameter*

All operations on the Automation View Interface have an optional **`out`** parameter of type VARIANT. The optional parameter returns explicit exception information in the context of each property set/get or method invocation. See “Mapping CORBA Exceptions to Automation Exceptions” on page 17-29 for a detailed discussion of how this mechanism works.

If the CORBA operation has no return value, then the optional parameter is the last parameter in the corresponding Automation operation. If the CORBA operation does have a return value, then the optional parameter appears directly before the return value in the corresponding Automation operation, since the return value must always be the last parameter.

### *Mapping for OMG IDL Single Inheritance*

A hierarchy of singly-inherited OMG IDL interfaces maps to an identical hierarchy of Automation View Interfaces.

For example, given the interface account and its derived interface **`checkingAccount`** defined as follows,

```
module MyModule {// OMG IDL
        interface account {
        attribute       float balance;
        readonly attributestring owner;
        void            makeLodgement (in float amount, out float
                        balance);
        void            makeWithdrawal (in float amount, out float
                        theBalance);
    };
    interface checkingAccount: account {
        readonly attribute float overdraftLimit;
        boolean         orderChequeBook ();
    };
};
```

the corresponding Automation View Interfaces are as follows

```
// ODL
[odl, dual, uuid(20c31e22-dcb2-aa79-1dc4-34a4ad297579)]
interface DIMyModule_account: IDispatch {
   HRESULT makeLodgement ([in] float amount,
            [out] float * balance,
            [optional, out] VARIANT * excep_OBJ);
   HRESULT makeWithdrawal ([in] float amount,
            [out] float * balance,
            [optional, out] VARIANT * excep_OBJ);
   [propget] HRESULT balance ([retval,out] float *
                [IT_retval];
   [propput] HRESULT balance ([in] float balance);
   [propget] HRESULT owner ([retval,out] BSTR * IT_retval);
};

[odl, dual, uuid(ffe752b2-a73f-2a28-1de4-21754778ab4b)]
interface DIMyModule_checkingAccount: IMyModule_account {
      HRESULT orderChequeBook(
               [optional, out] VARIANT * excep_OBJ,
               [retval,out] short * IT_retval);
      [propget] HRESULT overdraftLimit (
               [retval,out] short * IT_retval);
};
```

## *Mapping of OMG IDL Multiple Inheritance*

Automation does not support multiple inheritance; therefore, a direct mapping of a CORBA inheritance hierarchy using multiple inheritance is not possible. This mapping splits such a hierarchy, at the points of multiple inheritance, into multiple singly-inherited strands.

The mechanism for determining which interfaces appear on which strands is based on a left branch traversal of the inheritance tree. At points of multiple inheritance, the interface that is first in an ordering of the parent interfaces is included in what we call

the main strand, and other interfaces are assigned to other, secondary strands. (The ordering of parent interfaces is explained later in this section.) For example, consider the CORBA interface hierarchy, shown in Figure 17-3.

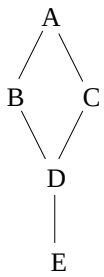


*Figure 17-3* A CORBA Interface Hierarchy Using Multiple Inheritance

We read this hierarchy as follows:

- B and C derive from A
- D derives from B and C
- E derives from D

This CORBA hierarchy maps to the following two Automation single inheritance hierarchies, shown in Figure 17-4.

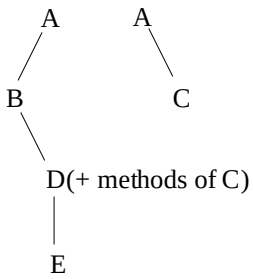


*Figure 17-4* The Mapped Automation Hierarchy Splits at the Point of Multiple Inheritance

Consider the multiple inheritance point D, which inherits from B and C. Following the left strand B at this point, our main strand is A-B-D and our secondary strand is A-C. However, to access all of the object's methods, a controller would have to navigate among these disjoint strands via QueryInterface. While such navigation is expected of COM clients and might be an acceptable requirement of C++ automation controllers, many Automation controller environments do not support such navigation.

To accommodate such controllers, at points of multiple inheritance we aggregate the operations of the secondary strands into the interface of the main strand. In our example, we add the operations of C to D (A's operations are not added because they already exist in the main strand). Thus, D has all the methods of the hierarchy and, more important, an Automation controller holding a reference to D can access all of the methods of the hierarchy without calling QueryInterface.

In order to have a reliable, deterministic, portable way to determine the inheritance chain at points of multiple inheritance, an explicit ordering model must be used. Furthermore, to achieve interoperability of virtual function tables for dual interfaces, a precise model for ordering operations and attributes within an interface must be specified.

Within an interface, attributes should appear before operations and both should be ordered lexicographically by bytes in machine-collating sequence. For non-readonly attributes, the **[propget]** method immediately precedes the **[propput]** method. This ordering determines the position of the vtable portion of a Dual Interface. At points of multiple inheritance, the base interfaces should be ordered from left to right lexicographically by bytes in machine-collating order. (In all cases, the ordering is based on ISO Latin-1.) Thus, the leftmost branch at a point of multiple inheritance is the one ordered first among the base classes, not necessarily the one listed first in the inheritance declaration.

Continuing with the example, the following OMG IDL code expresses a hierarchy conforming to Figure 17-3 on page 17-7.

```
// OMG IDL
module MyModule {
    interface A {
        void    aOp1();
        void    zOp1();
};
interface B: A{
        void    aOp2();
        void    zOp2();
};
interface C: A {
        void    aOp3();
        void    zOp3();
};
interface D: C, B{
        void    aOp4();
        void    zOp4();
    };
};
```

The OMG IDL maps to the following two Automation View hierarchies. Note that the ordering of the base interfaces for D has been changed based on our ISO Latin-1 alphabetic ordering model and that operations from C are added to interface D.

```
// ODL
// strand 1: A-B-D
[odl, dual, uuid(8db15b54-c647-553b-1dc9-6d098ec49328)]
interface DIMyModule_A: IDispatch {
   HRESULT  aOp1([optional,out] VARIANT * excep_OBJ);
   HRESULT  zOp1([optional,out] VARIANT * excep_OBJ);
}
[odl, dual, uuid(ef8943b0-cef8-21a5-1dc0-37261e082e51)]
interface DIMyModule_B: DIMyModule_A {
   HRESULT  aOp2([optional,out] VARIANT * excep_OBJ);
   HRESULT  zOp2([optional,out] VARIANT * excep_OBJ);
}
[odl, dual, uuid(67528a67-2cfd-e5e3-1de2-d59a444fe593)]
interface DIMyModule_D: DIMyModule_B {
   // C's aggregated operations
   HRESULT  aOp3([optional,out] VARIANT * excep_OBJ);
   HRESULT  zOp3([optional,out] VARIANT * excep_OBJ);
   // D's normal operations
   HRESULT  aOp4([optional,out] VARIANT * excep_OBJ);
   HRESULT  zOp4([optional,out] VARIANT * excep_OBJ);
}

// strand 2: A-C
[odl, dual, uuid(327885f8-ae9e-19c0-1dd5-d1ea05bcaae5)]
interface DIMyModule_C: DIMyModule_A {
   HRESULT  aOp3([optional,out] VARIANT * excep_OBJ);
   HRESULT  zOp3([optional,out] VARIANT * excep_OBJ);
}
```

Also note that the repeated operations of the aggregated strands are listed before D's operations. The ordering of these operations obeys the rules for operations within C and is independent of the ordering within D.

## 17.1.4 Mapping for Basic Data Types

### Basic Automation Types

Table 9 lists the basic data types supported by OLE Automation. The table contains fewer data types than those allowed by ODL because not all types recognized by ODL can be handled by the marshaling of IDispatch interfaces and by the implementation of **`ITypeInfo::Invoke`**. Arguments and return values of operations and properties are restricted to these basic types.

*Table 17-9* OLE Automation Basic Types

| Type | Description |
|---|---|
| boolean | True = -1, False = 0. |
| double | 64-bit IEEE floating-point number. |

| Type | Description |
|---|---|
| float | 32-bit IEEE floating-point number. |
| long | 32-bit signed integer. |
| short | 16-bit signed integer. |
| void | Allowed only as return type for a function, or in a function parameter list to indicate no parameters. |
| BSTR | Length-prefixed string. Prefix is an integer. |
| CURRENCY | 8-byte fixed-point number. |
| DATE | 64-bit floating-point fractional number of days since December 30, 1899. |
| SCODE | Built-in error type. In Win16, does not include additional data contained in an HRESULT. In Win32, identical to HRESULT. |
| IDispatch * | Pointer to IDispatch interface. From the viewpoint of the mapping, an IDispatch pointer parameter is an object reference. |
| IUnknown * | Pointer to IUnknown interface. (Any OLE interface can be represented by its IUnknown interface.) |

The formal mapping of CORBA types to Automation types is shown in Table 17-9.

*Table 17-9* OMG CORBA to OLE Automation Data Type Mappings

| CORBA Type | OLE Automation Type |
|---|---|
| boolean | boolean |
| char | short |
| double | double |
| float | float |
| long | long |
| octet | short |
| short | short |
| unsigned long | long |
| unsigned short | long |

### 17.1.5 Special Cases of Basic Data Type Mapping

An operation of an Automation View Interface must perform bidirectional translation of the Automation and CORBA parameters and return types. It must map from Automation to CORBA for **`in`** parameters and from CORBA to Automation for **`out`** parameters. The translation logic must handle the special conditions described in the following sections.

#### *Translating Automation long to CORBA unsigned long*

If the Automation long parameter is a negative number, then the View operation should return the HRESULT DISP_E_OVERFLOW.

#### *Translating CORBA unsigned long to Automation long*

If the **`CORBA::ULong`** parameter is greater than the maximum value of an Automation long, then the View operation should return the HRESULT DISP_E_OVERFLOW.

#### *Translating Automation long to CORBA unsigned short*

If the Automation long parameter is negative or is greater than the maximum value of a **`CORBA::UShort`**, then the View operation should return the HRESULT DISP_E_OVERFLOW.

#### *Translating Automation boolean to CORBA boolean and CORBA boolean to Automation boolean*

True and false values for CORBA boolean are, respectively, one (1) and zero (0). True and false values for Automation boolean are, respectively, negative one (-1) and zero (0). Therefore, true values need to be adjusted accordingly.

### 17.1.6 Mapping for Strings

An OMG IDL bounded or unbounded string maps to an OLE BSTR. For example, given the OMG IDL definitions,

```
// OMG IDL
string sortCode<20>;
string name;
```

the corresponding ODL code is

```
// ODL
BSTRsortCode;
BSTRname;
```

On Win32 platforms, a BSTR maps to a Unicode string. The use of BSTR is the only support for internationalization of strings defined at this time.

### 17.1.7 A Complete IDL to ODL Mapping for the Basic Data Types

There is no requirement that the OMG IDL code expressing the mapped CORBA interface actually exists. Other equivalent expressions of CORBA interfaces, such as the contents of an Interface Repository, may be used. Moreover, there is no requirement that ODL code corresponding to the CORBA interface be generated.

However, OMG IDL is the appropriate medium for describing a CORBA interface and ODL is the appropriate medium for describing an Automation View Interface. Therefore, the following OMG IDL code describes a CORBA interface that exercises all of the CORBA base data types in the roles of attribute, operation **`in`** parameter, operation **`out`** parameter, operation **`inout`** parameter, and return value. The OMG IDL code is followed by ODL code describing the Automation View Interface that would result from a conformant mapping.

```
module MyModule // OMG IDL
{
interface TypesTest
{
        attribute boolean   boolTest;
        attribute char      charTest;
        attribute double    doubleTest;
        attribute float     floatTest;
        attribute long      longTest;
        attribute octet     octetTest;
        attribute short     shortTest;
        attribute string    stringTest;
        attribute string<10>stringnTest;
        attribute unsigned long ulongTest;
        attribute unsigned short ushortTest;


        readonly attribute short readonlyShortTest;

        // Sets all the attributes
        boolean setAll (
                in boolean      boolTest,
                in char         charTest,
                in double       doubleTest,
                in float        floatTest,
                in long         longTest,
                in octet        octetTest,
                in short        shortTest,
```

```
                in string           stringTest,
                in string<10>       stringnTest,
                in unsigned long    ulongTest,
                in unsigned short   ushortTest);

        // Gets all the attributes
        boolean getAll (
                out boolean         boolTest,
                out char            charTest,
                out double          doubleTest,
                out float           floatTest,
                out long            longTest,
                out octet           octetTest,
                out short           shortTest,
                out string          stringTest,
                out string<10>      stringnTest,
                out unsigned long   ulongTest,
                out unsigned short  ushortTest);

        boolean setAndIncrement (
                inout boolean       boolTest,
                inout char          charTest,
                inout double        doubleTest,
                inout float         floatTest,
                inout long          longTest,
                inout octet         octetTest,
                inout short         shortTest,
                inout string        stringTest,
                inout string<10>    stringnTest,
                inout unsigned longulongTest,
                inout unsigned shortushortTest);

        boolean         boolReturn ();
        char            charReturn ();
        double          doubleReturn();
        float           floatReturn();
        long            longReturn ();
        octet           octetReturn();
        short           shortReturn ();
        string          stringReturn();
        string<10>      stringnReturn();
        unsigned long   ulongReturn ();
        unsigned shortushortReturn();

};// End of Interface TypesTest

};// End of Module MyModule
```

The corresponding ODL code is as follows.

```
[odl, dual, uuid(180d4c5a-17d2-a1a8-1de1-82e7a9a4f93b)]
interface DIMyModule_TypesTest: IDispatch {
   HRESULT boolReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] short *IT_retval);
   HRESULT charReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] short *IT_retval);
   HRESULT doubleReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] double *IT_retval);
   HRESULT floatReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] float *IT_retval);
   HRESULT getAll ([out] short *boolTest,
               [out] short *charTest,
               [out] double *doubleTest,
               [out] float *floatTest,
               [out] long *longTest,
               [out] short *octetTest,
               [out] short *shortTest,
               [out] BSTR stringTest,
               [out] BSTR *stringnTest,
               [out] long *ulongTest,
               [out] long *ushortTest,
               [optional,out] VARIANT * excep_OBJ,
               [retval,out] short * IT_retval);
   HRESULT longReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] long *IT_retval);
   HRESULT octetReturn ([optional,out] VARIANT * excep_OBJ,
               [retval,out] short *IT_retval);
   HRESULT setAll ([in] short boolTest,
               [in] short charTest,
               [in] double doubleTest,
               [in] float floatTest,
               [in] long longTest,
               [in] short octetTest,
               [in] short shortTest,
               [in] BSTR stringTest,
               [in] BSTR stringnTest,
               [in] long ulongTest,
               [in] long ushortTest,
               [optional,out] VARIANT * excep_OBJ,
               [retval,out] short * IT_retval);
   HRESULT setAndIncrement ([in,out] short *boolTest,
               [in,out] short *charTest,
               [in,out] double *doubleTest,
               [in,out] float *floatTest,
               [in,out] long *longTest,
               [in,out] short *octetTest,
               [in,out] short *shortTest,
               [in,out] BSTR *stringTest,
               [in,out] BSTR *stringnTest,
               [in,out] long *ulongTest,
               [in,out] long *ushortTest,
```

```
                [optional,out] VARIANT * excep_OBJ,
                [retval,out] short *IT_retval);
    HRESULT shortReturn ([optional,out] VARIANT * excep_OBJ,
                [retval,out] short *IT_retval);
    HRESULT stringReturn ([optional,out] VARIANT * excep_OBJ,
                [retval,out] BSTR *IT_retval);
    HRESULT stringnReturn ([optional,out] VARIANT *
                            exep_OBJ,
                [retval,out] BSTR *IT_retval);
    HRESULT ulongReturn ([optional,out] VARIANT * excep_OBJ,
                [retval,out] long *IT_retval);
    HRESULT ushortReturn ([optional,out] VARIANT * excep_OBJ,
                [retval,out] long *IT_retval);
    [propget] HRESULT boolTest([retval,out] short *IT_retval);
    [propput] HRESULT boolTest([in] short boolTest);
    [propget] HRESULT charTest([retval,out] short *IT_retval);
    [propput] HRESULT charTest([in] short charTest);
    [propget] HRESULT doubleTest([retval,out] double
       *IT_retval);
    [propput] HRESULT doubleTest([in] double doubleTest);
    [propget] HRESULT floatTest([retval,out] float
       *IT_retval);
    [propput] HRESULT floatTest([in] float floatTest);
    [propget] HRESULT longTest([retval,out] long *IT_retval);
    [propput] HRESULT longTest([in] long longTest);
    [propget] HRESULT octetTest([retval,out] short
       *IT_retval);
    [propput] HRESULT octetTest([in] short octetTest);
    [propget] HRESULT readonlyShortTest([retval,out] short
       *IT_retval);
    [propget] HRESULT shortTest([retval,out] short
       *IT_retval);
    [propput] HRESULT shortTest([in] short shortTest);
    [propget] HRESULT stringTest([retval,out] BSTR
       *IT_retval);
    [propput] HRESULT stringTest([in] BSTR stringTest);
    [propget] HRESULT stringnTest([retval,out] BSTR
       *IT_retval);
    [propput] HRESULT stringnTest([in] BSTR stringnTest);
    [propget] HRESULT ulongTest([retval,out] long *IT_retval);
    [propput] HRESULT ulongTest([in] long ulongTest);
    [propget] HRESULT ushortTest([retval,out] long
       *IT_retval);
    [propput] HRESULT ushortTest([in] long ushortTest);
}
```

### 17.1.8 Mapping for Object References

#### Type Mapping

The mapping of an object reference as a parameter or return value can be fully expressed by the following OMG IDL and ODL code. The OMG IDL code defines an interface Simple and another interface that references Simple as an **in** parameter, as an **out** parameter, as an **inout** parameter, and as a return value. The ODL code describes the Automation View Interface that results from an accurate mapping.

```
module MyModule // OMG IDL
{
// A simple object we can use for testing object references
interface Simple
{
        attribute short shortTest;
};

interface ObjRefTest
{
        attribute Simple simpleTest;
        Simple simpleOp(in Simple inTest,
             out Simple outTest,
             inout Simple inoutTest);
        };

}; // End of Module MyModule
```

The ODL code for the Automation View Dispatch Interface follows.

```
[odl, dual, uuid(c166a426-89d4-f515-1dfe-87b88727b4ea)]
interface DIMyModule_Simple: IDispatch
{
   [propget] HRESULT shortTest([retval, out] short *
                IT_retval);
   [propput] HRESULT shortTest([in] short shortTest);
}

[odl, dual, uuid(04843769-120e-e003-1dfd-6b75107d01dd)]
interface DIMyModule_ObjRefTest: IDispatch
{
   HRESULT simpleOp([in]DIMyModule_Simple *inTest,
            [out] DIMyModule_Simple **outTest,
            [in,out] DIMyModule_Simple **inoutTest,
            [optional, out] VARIANT * excep_OBJ,
            [retval, out] DIMyModule_Simple ** IT_retval);

   [propget] HRESULT simpleTest([retval, out]
               DIMyModule_Simple **
               IT_retval);
```

```
    [propput] HRESULT simpleTest([in] DIMyModule_Simple
                                 *simpleTest);

}
```

## *Object Reference Parameters and IForeignObject*

As described in the Interworking Architecture chapter, Automation and COM Views must expose the IForeignObject interface in addition to the interface that is isomorphic to the mapped CORBA interface. IForeignObject provides a mechanism to extract a valid CORBA object reference from a View object.

Consider an Automation View object B, which is passed as an **`in`** parameter to an operation M in View A. Operation M must somehow convert View B to a valid CORBA object reference. In Figure 17-1, Automation Views expose IForeignObject, as required of all Views.

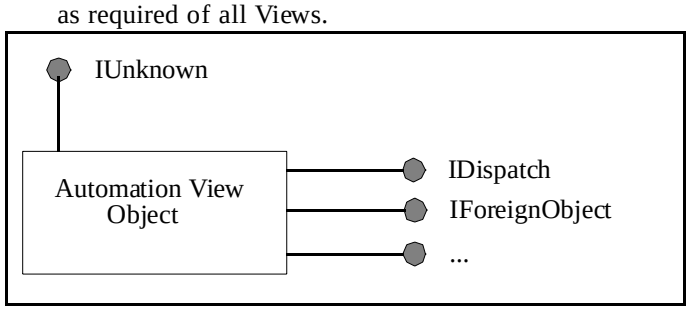


*Figure 17-1* Partial Picture of the Automation View

The sequence of events involving **`IForeignObject::GetForeignReference`** is as follows:

- The client calls **`Automation-View-A::M`**, passing an IDispatch-derived pointer to Automation-View-B.
- Automation-View-A::M calls **`IDispatch::QueryInterface`** for IForeignObject.
- Automation-View-A::M calls **`IForeignObject::GetForeignReference`** to get the reference to the CORBA object of type B.
- Automation-View-A::M calls **`CORBA-Stub-A::M`** with the reference, narrowed to interface type B, as the object reference **`in`** parameter.

### 17.1.9 Mapping for Enumerated Types

CORBA enums map to Automation enums. Consider the following example

```
// OMG IDL
module MyModule {
    enum color {red, green, blue};
    interface foo {
        void op1(in color col);
    };
};
```

which maps to the following ODL

```
// ODL
typedef enum {red, green, blue} MyModule_color;

[odl,dual,uuid(7d1951f2-b5d3-8b7c-1dc3-aa0d5b3d6a2b)]
interface DIMyModule_foo: IDispatch {
   HRESULT op1([in] MyModule_color col, [optional,out]
               VARIANT * excep_OBJ);
}
```

Internally, OLE Automation maps enum parameters to the platform's integer type. (For Win32, the integer type is equivalent to a long.) If the number of elements in the CORBA enum exceeds the maximum value of an integer, the condition should be trapped at some point during static or dynamic construction of the Automation View Interface corresponding to the CORBA interface in which the enum type appears as a parameter. If the overflow is detected at run-time, the Automation View operation should return the HRESULT DISP_E_OVERFLOW.

If an actual parameter applied to the mapped parameter in the Automation View Interface exceeds the maximum value of the enum, the View operation should return the HRESULT DISP_E_OVERFLOW.

Since all Automation controllers do not promote the ODL definition of enums into the controller scripting language context, vendors may wish to generate a header file containing an appropriate enum declaration or a set of constant declarations for the client language. Since the method for doing so is an implementation detail, it is not specified here. However, it should be noted that some languages type enums other than as longs, introducing the possibility of conversion errors or faults. If such problems arise, it is best to use a series of constant declarations rather than an enumerated type declaration in the client header file.

For example, the following **enum** declaration

**enum color {red, green, blue, yellow, white};// OMG IDL**

could be translated to the following Visual Basic code:

```
' Visual Basic
Global const color_red = 0
```

```
Global const color_green = 1
Global const color_blue = 2
Global const color_yellow = 3
Global const color_white = 4
```

In this case the default naming rules for the enum values should follow those for interfaces. That is, the name should be fully scoped with the names of enclosing modules or interfaces. (See “Naming Conventions for View Components” on page 15-29.)

If the enum is declared at global OMG IDL scope, as in the previous example, then the name of the enum should also be included in the constant name.

### *17.1.10 Mapping for Arrays and Sequences*

OLE Automation methods may have array parameters called Safearrays. Safearrays are one or multidimensional arrays whose elements are of any of the basic Automation types. The following ODL syntax describes an array parameter:

```
SAFEARRAY (elementtype) arrayname
```

A Safearray may be passed by reference, using the following syntax:

```
SAFEARRAY (elementtype) *arrayname
```

Safearrays have a header which describes certain characteristics of the array including bounding information, and are thus relatively safe for marshaling. Note that the ODL declaration of Safearrays does not include bound specifiers. OLE provides an API for allocating and manipulating Safearrays, which includes a procedure for resizing the array.

IDL arrays and sequences, both bounded and unbounded, are mapped to Safearrays. Bounded sequences are mapped to Safearrays with the same boundaries; they do not grow dynamically up to the bounded size but are statically allocated to the bounded size. Unbounded sequences are mapped to Safearrays with some default bound. Attempts to access past the boundary result in a resizing of the Safearray.

Since ODL Safearray declarations contain no boundary specifiers, the bounding knowledge is contained in the Automation View. A method of the Automation View Interface, which has a Safearray as a parameter, has the intelligence to handle the parameter properly. When Safearrays are submitted as **`in`** parameters, the View method uses the Safearray API to dynamically repackage the Safearray as a CORBA array, bounded sequence, or unbounded sequence. When Safearrays are **`out`** parameters, the View method uses the Safearray API to dynamically repackage the CORBA array or sequence as a Safearray. When an unbounded sequence grows beyond the current boundary of the corresponding Safearray, the View’s method uses the Safearray API to increase the size of the array by one allocation unit. The size of an allocation unit is unspecified. If a Safearray is mapped from a bounded sequence and a client of the View attempts to write to the Safearray past the maximum element of the bounded sequence, the View operation considers this a run-time error and returns the HRESULT DISP_E_OVERFLOW.

Multidimensional OMG IDL arrays map to multidimensional Safearrays. The order of dimensions in the OMG IDL array from left to right corresponds to ascending order of dimensions in the Safearray.

### 17.1.11 Mapping for CORBA Complex Types

CORBA constructed types—Structs, Unions and Exceptions—cannot be mapped directly to ODL constructed types, as Automation does not support them as valid parameter types. Instead, constructed types are mapped to Pseudo-Automation Interfaces. The objects that implement Pseudo-Automation Interfaces are called pseudo-objects. Pseudo-objects do not expose the IForeignObject interface.

Pseudo-Automation Interfaces are Dual Interfaces, but do not derive directly from IDispatch as do Automation View Interfaces. Instead, they derive from DIForeignComplexType:

```
// ODL
[odl, dual, uuid(...)]
interface DIForeignComplexType: IDispatch
{
[propget] HRESULT INSTANCE_repositoryId([retval,out]
BSTR *IT_retval);
HRESULT INSTANCE_clone([in] IDispatch *pDispatch,
[retval, out] IDispatch **IT_retval);
}
```

The UUID for DIForeignComplexType is:

**{A8B553C0-3B72-11cf-BBFC-444553540000}**

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DForeignComplexType and its UUID is:

**{E977F900-3B75-11cf-BBFC-444553540000}**

The purpose of the **DIForeignComplexType::INSTANCE_clone** method is to provide the client programmer a way to duplicate a complex type. **INSTANCE_clone** creates a new instance of the type with values identical to the input instance. Therefore, **INSTANCE_clone** does not simply duplicate a reference to a complex type.

The purpose of the **INSTANCE_repositoryId** readonly property is to support the ability of DICORBAAny (see “Mapping for anys” on page 17-24), when it wraps an instance of a complex type, to produce a type code for the instance when asked to do so via DICORBAAny’s readonly typeCode property.

## *Mapping for Structure Types*

CORBA structures are mapped to a Pseudo-Struct, which is an Pseudo-Automation Interface containing properties corresponding to the members of the struct. The names of a Pseudo-Struct's properties are identical to the names of the corresponding CORBA struct members.

A Pseudo-Struct derives from DICORBAStruct which, in turn, derives from DIForeignComplexType:

```
// ODL
[odl, dual, uuid(...)]
interface DICORBAStruct: DIForeignComplexType
{
}
```

The GUID for DICORBAStruct is:

```
{A8B553C1-3B72-11cf-BBFC-444553540000}
```

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBAStruct and its UUID is:

```
{E977F901-3B75-11cf-BBFC-444553540000}
```

The purpose of the methodless DICORBAStruct interface is to mark the interface as having its origin in the mapping of a CORBA struct. This information, which can be stored in a type library, is essential for the task of mapping the type back to CORBA in the event of an inverse mapping.

An example of mapping a CORBA struct to a Pseudo-Struct follows. The struct

```
struct S// IDL
{
    long l;
    double d;
    float f;
};
```

maps to Automation as follows, except that the mapped Automation Dual Interface derives from DICORBAStruct.

```
// IDL
interface S
{
    attribute long l;
    attribute double d;
    attribute float f;
};
```

## *Mapping for Union Types*

CORBA unions are mapped to a Pseudo-Automation Interface called a Pseudo-Union. A Pseudo-Union contains properties that correspond to the members of the union, with the addition of a discriminator property. The discriminator property's name is **UNION_d**, and its type is the Automation type that corresponds to the OMG IDL union discriminant.

If a union element is accessed from the Pseudo-Union, and the current value of the discriminant does not match the property being requested, then the operation of the Pseudo-Union returns **DISP_E_TYPEMISMATCH**. Whenever an element is set, the discriminant's value is set to the value that corresponds to that element.

A Pseudo-Union derives from the methodless interface DICORBAUnion which, in turn, derives from DIForeignComplexType:

```
// ODL
[odl, dual, uuid(...)]
interface DICORBAUnion: DIForeignComplexType // ODL
{
}
```

The UUID for DICORBAUnion is:

**{A8B553C2-3B72-11cf-BBFC-444553540000}**

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBAUnion and its UUID is:

**{E977F902-3B75-11cf-BBFC-444553540000}**

An example of mapping a CORBA union to a Pseudo-Union follows. The union

```
interface A;                // IDL

union U switch(long)
{
    case 1:  long l;
    case 2:  float f;
    default: A obj;
};
```

maps to Automation as if it were defined as follows, except that the mapped Automation Dual Interface derives from DICORBAUnion.

```
interface A;              // IDL

interface U
{
// Switch discriminant
readonly attribute long UNION_d;

    attribute long l;
    attribute float f;
    attribute A obj;
};
```

### 17.1.12 Mapping for TypeCodes

The OMG IDL TypeCode data type maps to the DICORBATypeCode interface. The DICORBATypeCode interface is defined as follows.

```
// ODL
typedef enum {
   tk_null = 0, tk_void, tk_short, tk_long, tk_ushort,
   tk_ulong, tk_float, tk_double, tk_octet,
   tk_any, tk_typeCode, tk_principal, tk_objref,
   tk_struct, tk_union, tk_enum, tk_string,
   tk_sequence, tk_array, tk_alias, tk_except
   } CORBATCKind;

[odl, dual, uuid(...)]
interface DICORBATypeCode: DIForeignComplexType {
   [propget] HRESULT kind([retval,out] TCKind * IT_retval);

   // for tk_objref, tk_struct, tk_union, tk_alias,
      tk_except
   [propget] HRESULT id([retval,out] BSTR *IT_retval);
   [propget] HRESULT name([retval,out] BSTR * IT_retval);

//tk_struct,tk_union,tk_enum,tk_except
   [propget] HRESULT member_count([retval,out]
      long * IT_retval);
   HRESULT member_name([in] long index,[retval,out]
      BSTR * IT_retval);
   HRESULT member_type([in] long index,
            [retval,out] IDispatch ** IT_retval),

// tk_union
   HRESULT member_label([in] long index,[retval,out]
         VARIANT * IT_retval);
   [propget] HRESULT discriminator_type([retval,out]
         IDispatch ** IT_retval);
   [propget] HRESULT default_index([retval,out]
```

```
            long * IT_retval);

   // tk_string, tk_array, tk_sequence
      [propget] HRESULT length([retval,out] long * IT_retval);

   // tk_sequence, tk_array, tk_alias
      [propget] HRESULT content_type([retval,out]
            IDispatch ** IT_retval);
   }
```

The UUID for DICORBATypeCode is:

**{A8B553C3-3B72-11cf-BBFC-444553540000}**

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBATypeCode and its UUID is:

**{E977F903-3B75-11cf-BBFC-444553540000}**

When generating Visual Basic constants corresponding to the values of the CORBATCKind enumeration, the constants should be declared as follows.

```
Global const CORBATCKind_tk_null =0
Global const CORBATCKind_tk_void = 1
. . .
```

Since DICORBATypeCode derives from DIForeignComplexType, objects which implement it are, in effect, pseudo-objects.

### 17.1.13 Mapping for ***any***s

The OMG IDL **any** data type maps to the DICORBAAny interface, which is declared as:

```
//ODL
[odl, dual, uuid(...)]
interface DICORBAAny: DIForeignComplexType
{
   [propget] HRESULT value([retval,out]
      VARIANT * IT_retval);
   [propput] HRESULT value([in] VARIANT val);
   [propget] HRESULT typeCode([retval,out]
      DICORBATypeCode ** IT_retval);
}
```

The UUID for DICORBAAny is:

**{A8B553C4-3B72-11cf-BBFC-444553540000}**

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBAAny and its UUID is:

**{E977F904-3B75-11cf-BBFC-444553540000}**

Since DICORBAAny derives from DIForeignComplexType, objects that implement it are, in effect, pseudo-objects. See Section 13.1.11, Mapping for CORBA Complex Types, for a description of the DIForeignComplexType interface.

Note that the VARIANT value property of DICORBAAny can represent a Safearray or can represent a pointer to a DICORBAStruct or DICORBAUnion interface. Therefore, the mapping for **any** is valid for an **any** that represents a CORBA array, sequence, structure, or union.

### *17.1.14 Mapping for Typedefs*

The mapping of OMG IDL **typedef** definitions to OLE depends on the OMG IDL type for which the **typedef** is defined. No mapping is provided for **typedef** definitions for the basic types: **float**, **double**, **long**, **short**, **unsigned long**, **unsigned short**, **char**, **boolean**, and **octet**. Hence, a Visual Basic programmer cannot make use of these **typedef** definitions.

```
// OMG IDL
module MyModule {
    module Module2 {
        module Module3 {
            interface foo {};
        };
    };
};
typedef MyModule::Module2::Module3::foo bar;
```

For complex types, the mapping creates an alias for the pseudo-object. For interfaces, the mapping creates an alias for the Automation View object. A conforming implementation may register these aliases in the Windows System Registry.

Creating a View for this interface would require something like the following:

```
‘ in Visual Basic
Dim a as Object
Set a = theOrb.GetObject(“MyModule.Module2.Module3.foo”)
‘ Release the object
Set a = Nothing
‘ Create the object using a typedef alias
Set a = theOrb.GetObject(“bar”)
```

### *17.1.15 Mapping for Constants*

The notion of a constant does not exist in OLE Automation; therefore, no mapping is prescribed for a CORBA constant.

As with the mapping for enums, some vendors may wish to generate a header file containing an appropriate constant declaration for the client language. For example, the following OMG IDL declaration

```
// OMG IDL
const long Max = 1000;
```

could be translated to the following in Visual Basic:

```
' Visual Basic
Global Const Max = 1000
```

The naming rules for these constants should follow that of enums.

### 17.1.16 Getting Initial CORBA Object References

The DICORBAFactory interface, described in “ICORBAFactory Interface” on page 15-24, provides a mechanism that is more suitable for the typical programmer in an Automation controller environment such as Visual Basic.

The implementation of the DICORBAFactory interface is not prescribed, but possible options include delegating to the OMG Naming Service and using the Windows System Registry[1].

The use of this interface from Visual Basic would appear as:

```
Dim theORBfactory as Object
Dim Target as Object
Set theORBfactory=CreateObject(“CORBA.Factory”)
Set Target=theORBfactory.GetObject
   (“software.sales.accounts”)
```

In Visual Basic 4.0 projects that have preloaded the standard CORBA Type Library, the code could appear as follows:

```
Dim Target as Object
Set Target=theORBfactory.GetObject(“soft-
ware.sales.accounts”)
```

The stringified name used to identify the desired target object should follow the rules for arguments to **`DICORBAFactory::GetObject`** described in “ICORBAFactory Interface” on page 15-24.

A special name space for names with a period in the first position can be used to resolve an initial reference to the OMG Object Services (for example, the Naming Service, the Life Cycle Service, and so forth). For example, a reference for the Naming Service can be found using:

---

1. It is always permissible to directly register a CORBA/OLE Automation bridging object directly with the Windows Registry. The administration and assignment of ProgIds for direct registration should follow the naming rules described in the Interworking Architecture chapter.

```
Dim NameContext as Object
Set NameContext=theORBfactory.GetObject(".NameService")
```

Generally the GetObject method will be used to retrieve object references from the Registry/Naming Service. The CreateObject **method** is really just a shorthand notation for GetObject("someName").create. It is intended to be used for object references to objects supporting a CORBAServices Factory interface.

### *17.1.17 Creating Initial in Parameters for Complex Types*

Although CORBA complex types are represented by Automation Dual Interfaces, creating an instance of a mapped CORBA complex type is not the same as creating an instance of a mapped CORBA interface. The main difference lies in the fact that the name space for CORBA complex types differs fundamentally from the CORBA object and factory name spaces.

To support creation of instances of Automation objects exposing Pseudo-Automation Interfaces, we define a new interface, derived from DICORBAFactory (see "ICORBAFactory Interface" on page 15-24 for a description of DICORBAFactory).

```
// ODL
[odl, dual, uuid(...)]
interface DICORBAFactoryEx: DICORBAFactory
{
   HRESULT CreateType([in] IDispatch *scopingObject,
      [in] BSTR typeName,
                [retval,out] VARIANT *val);
   HRESULT CreateTypeById([in] IDispatch *scopingObject,
      [in] BSTR repositoryId,
               [retval,out] VARIANT *val);
}
```

The UUID for DICORBAFactoryEx is:

**{A8B553C5-3B72-11cf-BBFC-444553540000}**

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBAFactoryEx and its UUID is:

**{E977F905-3B75-11cf-BBFC-444553540000}**

The Automation object having the ProgId "CORBA.Factory" shown next actually exposes DICORBAFactoryEx.

The CreateType method creates an Automation object that has been mapped from a CORBA complex type. The parameters are used to determine the specific type of object returned.

The first parameter, scopingObject, is a pointer to an Automation View Interface. The most derived interface type of the CORBA object bound to the View identifies the scope within which the second parameter, typeName, is interpreted. For example, assume the following CORBA interface exists:

```
// OMG IDL
module A {
    module B {
        interface C {
            struct S {
                // ...
            }
            void op(in S s);
        // ....
        }
    }
}
```

The following Visual Basic example illustrates the primary use of CreateType:

```
‘ Visual Basic
Dim myC as Object
Dim myS as Object
Dim myCORBAFactory as Object
Set myCORBAFactory = CreateObject(“CORBA.Factory”)
Set myC = myCORBAFactory.CreateObject( “...” )

‘ creates Automation View of the CORBA object
supporting interface ‘ A::B::C
Set myS = myCORBAFactory.CreateType(myC, “S”)
myC.op(myS)
```

The following rules apply to CreateType:

- The typeName parameter can contain a fully-scoped name (i.e., the name begins with a double colon “::”). If so, then the first parameter defines the type name space within which the fully scoped name will be resolved.
- If the scopingObject parameter does not point to a valid Automation View Interface, then CreateObject returns the HRESULT DISP_E_UNKNOWNINTERFACE.
- If the typeName parameter does not identify a valid type in the name space associated with the scopingObject parameter, then CreateObject returns the HRESULT TYPE_E_UNDEFINEDTYPE.

The CreateTypeByID method accomplishes the same general goal of CreateType, the creation of Automation objects that are mapped from CORBA constructed types. The second parameter, repositoryID, is a string containing the CORBA Interface Repository ID of the CORBA type whose mapped Automation Object is to be created. The Interface Repository associated with the CORBA object identified by the scopingObject parameter defines the repository within which the ID will be resolved.

The following rules apply to CreateTypeById:

- If the scopingObject parameter does not point to a valid Automation View Interface, then CreateObject returns the HRESULT DISP_E_UNKNOWNINTERFACE.

- If the repositoryID parameter does not identify a valid type in the Interface Repository associated with the scopingObject parameter, then CreateObject returns the HRESULT TYPE_E_UNDEFINEDTYPE.

### *ITypeFactory Interface*

The DICORBAFactory interface delegates its CreateType and CreateTypeByID methods to an ITypeFactory interface on the scoping object. ITypeFactory is defined as a COM interface because it is not intended to be exposed to Automation controllers. Every Automation View object must support the ITypeFactory interface:

```
//MIDL
interface ITypeFactory: IUnknown
{
   HRESULT CreateType([in] LPSTR typeName, [out] VARIANT
            *IT_retval);
   HRESULT CreateTypeById([in] RepositoryId repositoryID,
            [out] VARIANT *IT_retval);
}
```

The UUID for ITypeFactory is:

```
{A8B553C6-3B72-11cf-BBFC-444553540000}
```

The methods on ITypeFactory provide the behaviors previously described for the corresponding DICORBAFactory methods.

## *17.1.18 Mapping CORBA Exceptions to Automation Exceptions*

### *Overview of Automation Exception Handling*

Automation's notion of exceptions does not resemble true exception handling as defined in C++ and CORBA. Automation methods are invoked with a call to **`IDispatch::Invoke`** or to a vtable method on a Dual Interface. These methods return a 32-bit HRESULT, as do almost all COM methods. HRESULT values, which have the *severity* bit (bit 31 being the high bit) set, indicate that an error occurred during the call, and thus are considered to be error codes. (In Win16, an SCODE was defined as the lower 31 bits of an HRESULT, whereas in Win32 and for our purposes HRESULT and SCODE are identical.) HRESULTs also have a multibit field called the facility. One of the predefined values for this field is FACILITY_DISPATCH. Visual Basic 4.0 examines the return HRESULT. If the severity bit is set and the facility field has the value FACILITY_DISPATCH, then Visual Basic executes a built-in error handling routine, which pops up a message box and describes the error.

Invoke has among its parameters one of type EXCEPINFO*. The caller can choose to pass a pointer to an EXCEPINFO structure in this parameter or to pass NULL. If a non-NULL pointer is passed, the callee can choose to handle an error condition by returning the HRESULT DISP_E_EXCEPTION and by filling in the EXCEPINFO structure.

OLE also provides Error Objects, which are task local objects containing similar information to that contained in the EXCEPINFO structure. Error objects provide a way for Dual Interfaces to set detailed exception information.

Visual Basic allows the programmer to set up error traps, which are automatically fired when an invocation returns an HRESULT with the severity bit set. If the HRESULT is DISP_E_EXCEPTION, or if a Dual Interface has filled an Error Object, the data in the EXCEPINFO structure or in the Error Object can be extracted in the error handling routine.

### *CORBA Exceptions*

CORBA exceptions provide data not directly supported by the Automation error handling model. Therefore, all methods of Automation View Interfaces have an additional, optional **`out`** parameter of type VARIANT which is filled in by the View when a CORBA exception is detected.

Both CORBA System exceptions and User exceptions map to Pseudo-Automation Interfaces called pseudo-exceptions. Pseudo-exceptions derive from IForeignException which, in turn, derives from IForeignComplexType:

```
//ODL
[odl, dual, uuid(...)]
interface DIForeignException: DIForeignComplexType
{
   [propget] HRESULT EX_majorCode([retval,out] long
             *IT_retval);
   [propget] HRESULT EX_repositoryID([retval,out] BSTR
             *IT_retval);
};
```

The UUID for DIForeignException is:

**`{A8B553C7-3B72-11cf-BBFC-444553540000}`**

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named **`DForeignException`** and its UUID is:

**`{E977F907-3B75-11cf-BBFC-444553540000}`**

The attribute EX_majorCode defines the broad category of exception raised, and has one of the following numeric values:

```
NO_EXCEPTION = 0
SYSTEM_EXCEPTION = 1
USER_EXCEPTION = 2
```

These values may be specified as an enum in the typelibrary information:

```
typedef enum {NO_EXCEPTION,
SYSTEM_EXCEPTION,
USER_EXCEPTION } CORBA_ExceptionType;
```

The attribute **EX_repositoryID** is a unique string that identifies the exception. It is the exception type's repository ID from the CORBA Interface Repository.

## *CORBA User Exceptions*

A CORBA user exception is mapped to a properties-only pseudo-exception whose properties correspond one-to-one with the attributes of the CORBA user exception, and which derives from the methodless interface DICORBAUserException:

```
//ODL
[odl, dual, uuid(...)]
interface DICORBAUserException: DIForeignException
{
}
```

The UUID for DICORBAUserException is:

```
{A8B553C8-3B72-11cf-BBFC-444553540000}
```

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBAUserException and its UUID is:

```
{E977F908-3B75-11cf-BBFC-444553540000}
```

Thus, an OMG IDL exception declaration is mapped to an OLE definition as though it were defined as an interface. The declaration

```
// OMG IDL
exception reject
{
    string reason;
};
```

maps to the following ODL:

```
//ODL
[odl, dual, uuid(6bfaf02d-9f3b-1658-1dfb-7f056665a6bd)]
interface DIreject: DICORBAUserException
{
   [propget] HRESULT reason([retval,out] BSTR reason);
}
```

## *Operations that Raise User Exceptions*

If the optional exception parameter is supplied by the caller and a User Exception occurs, the parameter is filled in with an IDispatch pointer to an exception Pseudo-Automation Interface, and the operation on the Pseudo-Interface returns the HRESULT

S_FALSE. S_FALSE does not have the severity bit set, so that returning it from the operation prevents an active Visual Basic Error Trap from being fired, allowing the caller to retrieve the exception parameter in the context of the invoked method. The View fills in the VARIANT by setting its *vt* field to VT_DISPATCH and setting the **`pdispval`** field to point to the pseudo-exception. If no exception occurs, the optional parameter is filled with an IForeignException pointer on a pseudo-exception object whose **`EX_majorCode`** property is set to NO_EXCEPTION.

If the optional parameter is not supplied and an exception occurs, and

- If the operation was invoked via **`IDispatch::Invoke`**, then
  - The operation returns DISP_E_EXCEPTION.
  - If the caller provided an EXCEPINFO, then it is filled by the View.
- If the method was called via the vtable portion of a Dual Interface, then the OLE Error Object is filled by the View.

Note that in order to support Error Objects, Automation Views must implement the standard OLE interface ISupportErrorInfo.

*Table 17-1* EXCEPINFO Usage for CORBA User Exceptions

| **Field** | **Description** |
|---|---|
| wCode | Must be zero. |
| bstrSource | <interface name>.<operation name><br>*where the interface and operation names are those of the CORBA interface, which this Automation View is representing.* |
| bstrDescription | CORBA User Exception [<exception repository id>]<br>*where the repository id is that of the CORBA user exception.* |
| bstrHelpFile | Unspecified |
| dwHelpContext | Unspecified |
| pfnDeferredFillIn | NULL |
| scode | DISP_E_EXCEPTION |

*Table 17-2* ErrorObject Usage for CORBA User Exceptions

| **Property** | **Description** |
|---|---|
| bstrSource | <interface name>.<operation name><br>*where the interface and operation names are those of the CORBA interface, which this Automation View is representing.* |
| bstrDescription | CORBA User Exception: [<exception repository id>]<br>*where the repository id is that of the CORBA user exception.* |

*Table 17-2* ErrorObject Usage for CORBA User Exceptions *(Continued)*

| Property | Description |
|---|---|
| bstrHelpFile | Unspecified |
| dwHelpContext | Unspecified |
| GUID | The IID of the Automation View Interface. |

## *CORBA System Exceptions*

A CORBA System Exception is mapped to the Pseudo-Exception DICORBASystemException, which derives from DIForeignException:

```
// ODL
[odl, dual, uuid(...)]
interface DICORBASystemException: DIForeignException
{
   [propget] HRESULT EX_minorCode([retval,out] long
      *IT_retval);
   [propget] HRESULT EX_completionStatus([retval,out] long
      *IT_retval);
}
```

The UUID for DICORBASystemException is:

```
{1E5FFCA0-563B-11cf-B8FD-444553540000}
```

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCORBASystemException and its UUID is:

```
{1E5FFCA1-563B-11cf-B8FD-444553540000}
```

The attribute **`EX_minorCode`** defines the type of system exception raised, while **`EX_completionStatus`** has one of the following numeric values:

```
COMPLETION_YES = 0
COMPLETION_NO = 1
COMPLETION_MAYBE =
```

These values may be specified as an enum in the typelibrary information:

```
typedef enum {COMPLETION_YES,
COMPLETION_NO,
COMPLETION_MAYBE } CORBA_ExceptionType;
```

## *Operations that Raise System Exceptions*

As is the case for UserExceptions, system exceptions can be returned to the caller using the optional last parameter, which is present on all mapped methods.

If the optional parameter is supplied and a system exception occurs, the optional parameter is filled in with an IForeignException pointer to the pseudo-exception, and the automation return value is S_FALSE. If no exception occurs, the optional parameter is filled with an IForeignException pointer whose **`EX_majorCode`** property is set to NO_EXCEPTION.

If the optional parameter is not supplied and a system exception occurs, the exception is looked up in Table 17-3. This table maps a subset of the CORBA system exceptions to semantically equivalent FACILITY_DISPATCH HRESULT values. If the exception is on the table, the equivalent HRESULT is returned. If the exception is not on the table, that is, if there is no semantically equivalent FACILITY_DISPATCH HRESULT, then the exception is mapped to an HRESULT according to Table 16-3 on page 16-12. This new HRESULT is used as follows.

- If the operation was invoked via **`IDispatch::Invoke`**:
  - The operation returns DISP_E_EXCEPTION.
  - If the caller provided an EXCEPINFO, then it is filled with the scode field set to the new HRESULT value.
- If the method was called via the vtable portion of a Dual Interface:
  - The OLE Error Object is filled.
  - The method returns the new HRESULT.

*Table 17-3* CORBA Exception to COM Error Codes

| CORBA Exception | COM Error Codes |
|---|---|
| BAD_OPERATION | DISP_E_MEMBERNOTFOUND |
| NO_RESPONSE | DISP_E_PARAMNOTFOUND |
| BAD_INV_ORDER | DISP_E_BADINDEX |
| INV_IDENT | DISP_E_UNKNOWNNAME |
| INV_FLAG | DISP_E_PARAMNOTFOUND |
| DATA_CONVERSION | DISP_E_OVERFLOW |

*Table 17-4* EXCEPINFO Usage for CORBA System Exceptions

| Field | Description |
|---|---|
| wCode | Must be zero. |
| bstrSource | <interface name>.<operation name><br>*where the interface and operation names are those of the CORBA interface, which this Automation View is representing.* |
| bstrDescription | CORBA System Exception: [<exception repository id>] minor code [<minor code>][<completion status>]<br>*where the <exception repository id> and <minor code> are those of the CORBA system exception. <completion status> is "YES," "NO," or "MAYBE" based upon the value of the system exceptions's CORBA completion status. Spaces and square brackets are literals and must be included in the string.* |
| bstrHelpFile | Unspecified |
| dwHelpContext | Unspecified |
| pfnDeferredFillIn | NULL |
| scode | Mapped COM error code from Table 13-3 in Chapter 13B. |

*Table 17-5* ErrorObject Usage for CORBA System Exceptions

| Property | Description |
|---|---|
| bstrSource | <interface name>.<operation name><br>*where the interface and operation names are those of the CORBA interface, which this Automation View is representing.* |
| bstrDescription | CORBA System Exception: [<exception repository id>] minor code [<minor code>][<completion status>]<br>*where the <exception repository id> and <minor code> are those of the CORBA system exception. <completion status> is "YES," "NO," or "MAYBE" based upon the value of the system exceptions's CORBA completion status. Spaces and square brackets are literals and must be included in the string.* |
| bstrHelpFile | Unspecified |
| dwHelpContext | Unspecified |
| GUID | The IID of the Automation View Interface. |

### 17.1.19 Conventions for Naming Components of the Automation View

The conventions for naming components of the Automation View are detailed in“Naming Conventions for View Components” on page 15-29.

### 17.1.20 Naming Conventions for Pseudo-Structs, Pseudo-Unions, and Pseudo-Exceptions

The formulas used to name components of the Automation View (see “Naming Conventions for View Components” on page 15-29) are also used to name components Pseudo-Structs, Pseudo-Unions, and Pseudo-Exceptions. The CORBA type name is used as input to the formulas, just as the CORBA interface name is used as input to the formulas when mapping interfaces.

These formulas apply to the name and IID of the Pseudo-Automation Interface, and to the Program Id and Class Id of an object implementing the Pseudo-Automation Interface if it is registered in the Windows System Registry.

### 17.1.21 Automation View Interface as a Dispatch Interface (Nondual)

In addition to implementing the Automation View Interface as an OLE Automation Dual Interface, it is also acceptable to map it as a generic Dispatch Interface.

In this case, the normal methods and attribute accessor/assign methods are not required to have HRESULT return values. Instead, an additional “dispinterface” is defined, which can use the standard OLE dispatcher to dispatch invocations.

For example, a method declared in a dual interface in ODL as follows:

```
HRESULT aMethod([in] <type> arg1, [out] <type> arg2,
   [retval, out] <return type> IT_retval)
```

would be declared in ODL in a dispatch interface in the following form:

```
<return type> aMethod([in] <type> arg1, [out] <type> arg2)
```

Using the example from “Mapping for Interfaces” on page 17-3:

```
interface account
{           // OMG IDL
        attribute float balance;
        readonly attribute string owner;
        void makeLodgement (in float amount, out float
        balance);
        void makeWithdrawal (in float amount, out float
        balance);
};
```

the corresponding Iaccount interfaces are defined as follows.

```
[odl, uuid(e268443e-43d9-3dab-1d7e-f303bbe9642f)]
interface Iaccount: IUnknown {// ODL
      void makeLodgement ([in] float amount,
         [out] float balance,[out,optional]
            VARIANT *excep_OBJ);
      void makeWithdrawal([in] float amount,
         [out] float balance,[out,optional]
         VARIANT *excep_OBJ);
      [propget] float balance ([retval,out] *IT_retval);
      [propput] void balance ([in] float balance)
      [propget] BSTR owner ([retval,out] *IT_retval);
}
[uuid(e268443e-43d9-3dab-1dbe-f303bbe9642f)]
dispinterface Daccount {
   interface Iaccount;
};
```

A separate “dispinterface” declaration is required because Iaccount derives from IUnknown. The dispatch interface is DIaccount. Thus, in the example used for mapping object references in “Mapping for Object References” on page 17-16, the reference to the Simple interface in the OMG IDL would map to a reference to **`IMyModule_Simple`** rather than **`DIMyModule_Simple`**. The naming conventions for Dispatch Interfaces (and for their IIDs) exposed by the View are slightly different from Dual Interfaces. See “Naming Conventions for View Components” on page 15-29 for details.

The Automation View Interface must correctly respond to a QueryInterface for the specific Dispatch Interface Id (DIID) for that View. By conforming to this requirement, the Automation View can be strongly type-checked. For example, **`ITypeInfo::Invoke`**, when handling a parameter that is typed as a pointer to a specific DIID, calls QueryInterface on the object for that DIID to make sure the object is of the required type.

Pseudo-Automation Interfaces representing CORBA complex types such as structs, unions, exceptions and the other noninterface constructs mapped to dispatch interfaces can also be exposed as nondual dispatch interfaces.

### *17.1.22 Aggregation of Automation Views*

COM’s implementation reuse mechanism is aggregation. Automation View objects must either be capable of being aggregated in the standard COM fashion or must follow COM rules to indicate their inability or unwillingness to be aggregated.

The same rule applies to pseudo-objects.

### *17.1.23 DII and DSI*

OLE Automation interfaces are inherently self-describing and may be invoked dynamically. There is no utility in providing a mapping of the DII interfaces and related pseudo-objects into OLE Automation interfaces.

## 17.2 Automation Objects as CORBA Objects

This problem is the reverse of exposing CORBA objects as Automation objects. It is best to solve this problem in a manner similar to the approach for exposing CORBA objects as Automation objects.

### 17.2.1 Architectural Overview

We begin with ODL or type information for an Automation object, which implements one or more dispatch interfaces and whose server application exposes a class factory for its COM class.

We then create a CORBA View object, which provides skeletal implementations of the operations of each of those interfaces. The CORBA View object is in every way a legal CORBA object. It is not an Automation object. The skeleton is placed on the machine where the real Automation object lives.

The CORBA View is not fully analogous to the Automation View which, as previously explained, is used to represent a CORBA object as an Automation object. The Automation View has to reside on the client side because COM is not distributable. A copy of the Automation View needs to be available on every client machine.

The CORBA View, however, can live in the real CORBA object's space and can be represented on the client side by the CORBA system's stub because CORBA is distributable. Thus, only one copy of this View is required.

---

**Note –** Throughout this section, the term *CORBA View* is distinct from CORBA stubs and skeletons, from COM proxies and stubs, and from Automation Views.

---

The CORBA View is an Automation client. Its implementations of the CORBA operations translate parameter types and delegate to the corresponding methods of the real Automation object. When a CORBA client wishes to instantiate the real Automation object, it instantiates the CORBA View.

Thus, from the point of view of the client, it is interacting with a CORBA object which may be a remote object. CORBA handles all of the interprocess communication and marshaling. No COM proxies or stubs are created.

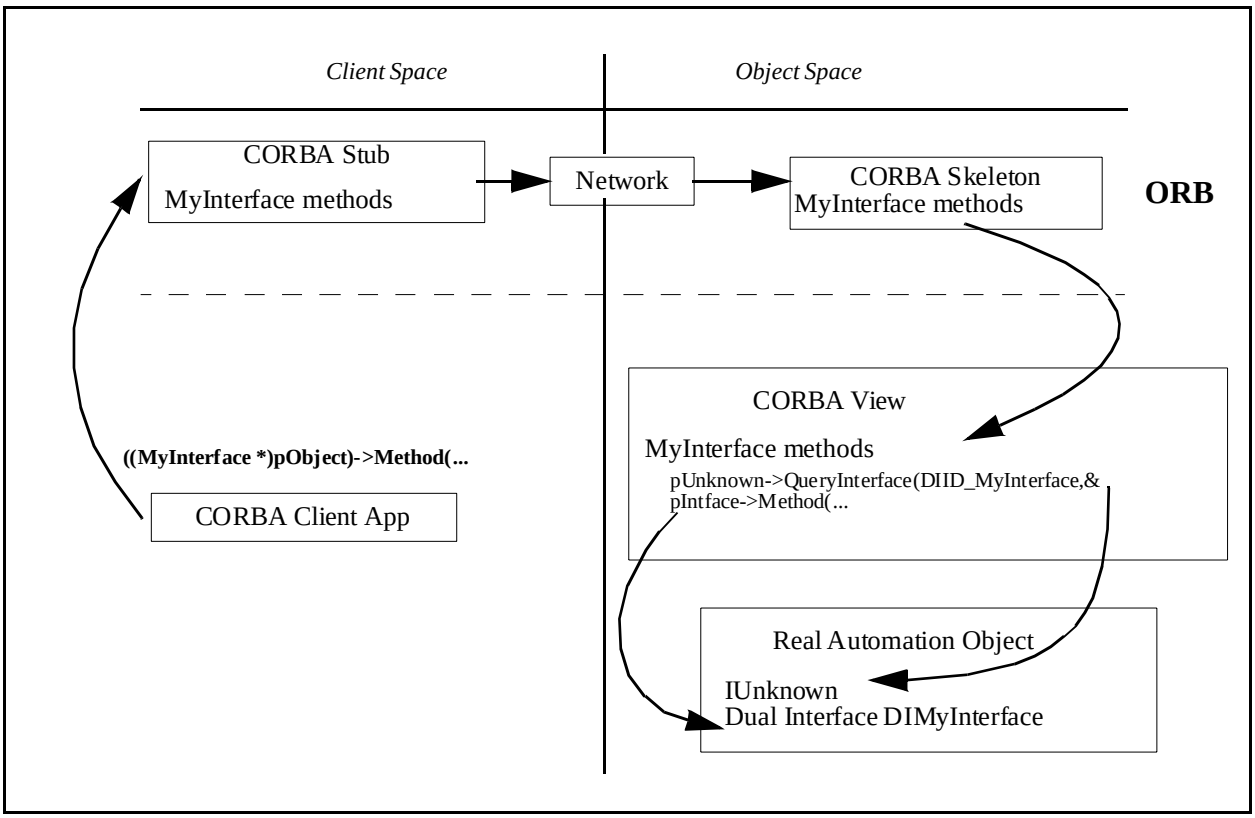


*Figure 17-2* The CORBA View: a CORBA Object, which is a Client of a COM Object

### *17.2.2 Main Features of the Mapping*

- ODL or type library information can form the input for the mapping.
- Automation properties and methods map to OMG IDL attributes and operations, respectively.
- Automation interfaces map to OMG IDL interfaces.
- Automation basic types map to corresponding OMG IDL basic types where possible.
- Automation errors are mapped similarly to COM errors.

### *17.2.3 Getting Initial Object References*

The OMG Naming Service can be used to get initial references to the CORBA View Interfaces. These interfaces may be registered as normal CORBA objects on the remote machine.

### 17.2.4 Mapping for Interfaces

The mapping for an ODL interface to a CORBA View interface is straightforward. Each interface maps to an OMG IDL interface. In general, we map all methods and properties with the exception of the IUnknown and IDispatch methods.

For example, given the ODL interface **IMyModule_account**,

```
[odl, dual, uuid(...)]
interface DIMyModule_account: IDispatch
{
   [propget] HRESULT balance([retval,out] float * ret);
};
```

the following is the OMG IDL equivalent:

```
// OMG IDL
interface MyModule_account
{
   readonly attribute float balance;
};
```

If the ODL interface does not have a parameter with the **[retval,out]** attributes, its return type is mapped to long. This allows COM SCODE values to be passed through to the CORBA client.

### 17.2.5 Mapping for Inheritance

A hierarchy of Automation interfaces is mapped to an identical hierarchy of CORBA View Interfaces.

For example, given the interface “account” and its derived interface “checkingAccount” defined next,

```
// ODL
[odl, dual, uuid(...)]
interface DIMyModule_account: IDispatch {
   [propput] HRESULT balance([in] float balance);
   [propget] HRESULT balance([retval,out] float * ret);
   [propget] HRESULT owner([retval,out] BSTR * ret);
   HRESULT makeLodgement([in] float amount,
         [out] float * balance);
   HRESULT makeWithdrawal([in] float amount,
         [out] float * balance);
};
interface DIMyModule_checkingAccount: DIMyModule_account {
   [propget] HRESULT overdraftLimit ([retval,out]
      short * ret);
   HRESULT orderChequeBook([retval,out] short * ret);
};
```

the corresponding CORBA View Interfaces are:

```
// OMG IDL
interface MyModule_account {
    attribute       float balance;
    readonly attribute string owner;
    long            makeLodgement (in float amount, out float
                    balance);
    long            makeWithdrawal (in float amount, out float
                    theBalance);
};
interface MyModule_checkingAccount: MyModule_account {
    readonly attributeshort overdraftLimit;
    short           orderChequeBook ();
};
```

### *17.2.6 Mapping for ODL Properties and Methods*

An ODL property which has either a get/set pair or just a set method is mapped to an OMG IDL attribute. An ODL property with just a get accessor is mapped to an OMG IDL **`readonly`** attribute.

Given the ODL interface definition

```
// ODL
[odl, dual, uuid(...)]
interface DIaccount: IDispatch {
   [propput] HRESULT balance ([in] float balance,
   [propget] HRESULT balance ([retval,out] float * ret);
   [propget] HRESULT owner ([retval,out] BSTR * ret);
   HRESULT makeLodgement ([in] float amount,
            [out] float * balance,
            [optional, out] VARIANT * excep_OBJ);
   HRESULT makeWithdrawal([in] float amount,
            [out] float * balance,
            [optional, out] VARIANT * excep_OBJ);
}
```

the corresponding OMG IDL interface is:

```
// OMG IDL
interface account {
    attribute float balance;
    readonly attribute string owner;
    long makeLodgement(in float amount, out float balance);
    long makeWithdrawal(in float amount, out float balance);
};
```

ODL **`[in]`**, **`[out]`**, and **`[in,out]`** parameters map to OMG IDL **`in`**, **`out`**, and **`inout`** parameters, respectively. “Mapping for Basic Data Types” on page 17-9 explains the mapping for basic types.

### 17.2.7 Mapping for Automation Basic Data Types

#### Basic Automation Types

The basic data types allowed by OLE Automation as parameters and return values are detailed in “Mapping for Basic Data Types” on page 17-9.

The formal mapping of CORBA types to Automation types is shown in Table 17-6.

*Table 17-6* Mapping of Automation Types to OMG IDL Types

| OLE Automation Type | OMG IDL Type |
|---|---|
| boolean | boolean |
| short | short |
| double | double |
| float | float |
| long | long |
| BSTR | string |
| CURRENCY | COM::Currency |
| DATE | double |
| SCODE | long |

The Automation CURRENCY type is a 64-bit integer scaled by 10,000, giving a fixed point number with 15 digits left of the decimal point and 4 digits to the right. The **`COM::Currency`** type is thus defined as follows:

```
module COM
{
    struct Currency
    {
        unsigned long lower;
        long upper;
    }
}
```

This mapping of the CURRENCY type is transitional and should be revised when the extended data types revisions to OMG IDL are adopted. These revisions are slated to include a 64-bit integer.

The Automation DATE type is an IEEE 64-bit floating-point number representing the number of days since December 30, 1899.

### 17.2.8 Conversion Errors

An operation of a CORBA View Interface must perform bidirectional translation of the Automation and CORBA parameters and return types. It must map from CORBA to Automation for **`in`** parameters and from Automation to CORBA for **`out`** parameters.

When the CORBA View encounters an error condition while translating between CORBA and Automation data types, it raises the CORBA system exception DATA_CONVERSION.

### 17.2.9 Special Cases of Data Type Conversion

#### Translating COM::Currency to Automation CURRENCY

If the supplied **`COM::Currency`** value does not translate to a meaningful Automation CURRENCY value, then the CORBA View should raise the CORBA System Exception DATA_CONVERSION.

#### Translating CORBA double to Automation DATE

If the CORBA double value is negative or converts to an impossible date, then the CORBA View should raise the CORBA System Exception DATA_CONVERSION.

#### Translating CORBA boolean to Automation boolean and Automation boolean to CORBA boolean

True and false values for CORBA boolean are, respectively, one and zero. True and false values for Automation boolean are, respectively, negative one (-1) and zero. Therefore, true values need to be adjusted accordingly.

### 17.2.10 A Complete OMG IDL to ODL Mapping for the Basic Data Types

As previously stated, there is no requirement that the ODL code expressing the mapped Automation interface actually exist. Other equivalent expressions of Automation interfaces, such as the contents of a Type Library, may be used. Moreover, there is no requirement that OMG IDL code corresponding to the CORBA View Interface be generated.

However, ODL is the appropriate medium for describing an Automation interface, and OMG IDL is the appropriate medium for describing a CORBA View Interface. Therefore, we provide the following ODL code to describe an Automation interface, which exercises all of the Automation base data types in the roles of properties, method **`[in]`** parameter, method **`[out]`** parameter, method **`[inout]`** parameter, and return value. The ODL code is followed by OMG IDL code describing the CORBA View Interface, which would result from a conformant mapping.

```
// ODL
[odl, dual, uuid(...)]
interface DIMyModule_TypesTest: IForeignObject {
   [propput] HRESULT boolTest([in] short boolTest);
   [propget] HRESULT boolTest([retval,out] short
*IT_retval);
   [propput] HRESULT doubleTest([in] double doubleTest);
   [propget] HRESULT doubleTest([retval,out] double
*IT_retval);
      [propput] HRESULT floatTest([in] float floatTest);
   [propget] HRESULT floatTest([retval,out] float
*IT_retval);
   [propput] HRESULT longTest([in] long longTest);
   [propget] HRESULT longTest([retval,out] long *IT_retval);
   [propput] HRESULT shortTest([in] short shortTest);
   [propget] HRESULT shortTest([retval,out] short
*IT_retval);
   [propput] HRESULT stringTest([in] BSTR stringTest);
   [propget] HRESULT stringTest([retval,out] BSTR
*IT_retval);
   [propput] HRESULT dateTest([in] DATE stringTest);
   [propget] HRESULT dateTest([retval,out] DATE *IT_retval);
   [propput] HRESULT currencyTest([in] CURRENCY stringTest);
   [propget] HRESULT currencyTest([retval,out] CURRENCY
*IT_retval);
   [propget] HRESULT readonlyShortTest([retval,out] short
*IT_retval);
   HRESULT setAll ([in] short boolTest,
            [in] double doubleTest,
            [in] float floatTest,
            [in] long longTest,
            [in] short shortTest,
            [in] BSTR stringTest,
            [in] DATE dateTest,
            [in] CURRENCY currencyTest,
            [retval,out] short * IT_retval);
   HRESULT getAll ([out] short *boolTest,
            [out] double *doubleTest,
            [out] float *floatTest,
            [out] long *longTest,
            [out] short *shortTest,
            [out] BSTR stringTest,
            [out] DATE * dateTest,
            [out] CURRENCY *currencyTest,
            [retval,out] short * IT_retval);
   HRESULT setAndIncrement ([in,out] short *boolTest,
            [in,out] double *doubleTest,
            [in,out] float *floatTest,
            [in,out] long *longTest,
            [in,out] short *shortTest,
            [in,out] BSTR *stringTest,
```

```
            [in,out] DATE * dateTest,
            [in,out] CURRENCY * currencyTest,
            [retval,out] short *IT_retval);
   HRESULT boolReturn ([retval,out] short *IT_retval);
   HRESULT doubleReturn ([retval,out] double *IT_retval);
   HRESULT floatReturn ([retval,out] float *IT_retval);
   HRESULT longReturn ([retval,out] long *IT_retval);
   HRESULT shortReturn ([retval,out] short *IT_retval);
   HRESULT stringReturn ([retval,out] BSTR *IT_retval);
   HRESULT octetReturn ([retval,out] DATE *IT_retval);
   HRESULT currencyReturn ([retval,out] CURRENCY
*IT_retval);
}
```

The corresponding OMG IDL is as follows.

```
// OMG IDL
interface MyModule_TypesTest

{
    attribute boolean   boolTest;
    attribute double    doubleTest;
    attribute float     floatTest;
    attribute long      longTest;
    attribute short     shortTest;
    attribute string    stringTest;
    attribute double    dateTest;
    attribute COM::Currency currencyTest;

    readonly attribute short readonlyShortTest;

    // Sets all the attributes
    boolean setAll (in boolean      boolTest,
                    in double       doubleTest,
                    in float        floatTest,
                    in long         longTest,
                    in short        shortTest,
                    in string       stringTest,
                    in double       dateTest,
                    in COM::Currency currencyTest);

    // Gets all the attributes
    boolean getAll (out boolean     boolTest,
                    out double      doubleTest,
                    out float       floatTest,
                    out long        longTest,
                    out short       shortTest,
                    out string      stringTest,
                    out double      dateTest,
                    out COM::Currency currencyTest);
```

```
boolean setAndIncrement (
                inout boolean       boolTest,
                inout double        doubleTest,
                inout float         floatTest,
                inout long          longTest,
                inout short         shortTest,
                inout string        stringTest,
                inout double        dateTest,
                inout COM::Currency currencyTest);
boolean         boolReturn ();
double          doubleReturn();
float           floatReturn();
long            longReturn ();
short           shortReturn ();
string          stringReturn();
double          dateReturn ();
COM::CurrencycurrencyReturn();

}; // End of Interface TypesTest
```

### 17.2.11 Mapping for Object References

The mapping of an object reference as a parameter or return value can be fully expressed by the following OMG IDL and ODL code. The ODL code defines an interface “Simple” and another interface that references Simple as an **in** parameter, an **out** parameter, an **inout** parameter, and as a return value. The OMG IDL code describes the CORBA View Interface that results from a proper mapping.

```
// ODL
[odl, dual, uuid(...)]
interface DIMyModule_Simple: IDispatch
{
   [propget] HRESULT shortTest([retval, out]
      short * IT_retval);
   [propput] HRESULT shortTest([in] short sshortTest);
}

[odl, dual, uuid(...)]
interface DIMyModule_ObjRefTest: IDispatch
{
   [propget] HRESULT simpleTest([retval, out]
      DIMyModule_Simple ** IT_retval);
   [propput] HRESULT simpleTest([in] DIMyModule_Simple
      *pSimpleTest);

   HRESULT simpleOp([in] DIMyModule_Simple *inTest,
               [out] DIMyModule_Simple **outTest,
               [in,out]DIMyModule_Simple **inoutTest,
               [retval, out] DIMyModule_Simple **IT_retval);
}
```

The OMG IDL code for the CORBA View Dispatch Interface is as follows.

```
// OMG IDL
// A simple object we can use for testing object references
interface MyModule_Simple
{
    attribute short shortTest;
};

interface MyModule_ObjRefTest
{
    attribute MyModule_Simple simpleTest;
    MyModule_Simple simpleOp(in MyModule_Simple inTest,
                out MyModule_Simple outTest,
                inout MyModule_Simple inoutTest);
};
```

### 17.2.12 Mapping for Enumerated Types

ODL enumerated types are mapped to OMG IDL enums; for example:

```
// ODL
typedef enum MyModule_color {red, green, blue};

[odl,dual,uuid(...)]
interface DIMyModule_foo: IDispatch {
   HRESULT op1([in] MyModule_color col);
}

// OMG IDL
module COM {
    enum MyModule_color {red, green, blue};
    interfacefoo: COM::CORBA_View {
        long op1(in MyModule_color col);
    };
};
```

### 17.2.13 Mapping for SafeArrays

Automation SafeArrays should be mapped to CORBA unbounded sequences.

A method of the CORBA View Interface, which has a SafeArray as a parameter, will have the knowledge to handle the parameter properly.

When SafeArrays are **`in`** parameters, the View method uses the Safearray API to dynamically repackage the SafeArray as a CORBA sequence. When arrays are **`out`** parameters, the View method uses the Safearray API to dynamically repackage the CORBA sequence as a SafeArray.

### *Multidimensional SafeArrays*

SafeArrays are allowed to have more than one dimension. However, the bounding information for each dimension, and indeed the number of dimensions, is not available in the static typelibrary information or ODL definition. It is only available at run-time.

For this reason, SafeArrays, which have more than one dimension, are mapped to an identical linear format and then to a sequence in the normal way.

This linearization of the multidimensional SafeArray should be carried out as follows:

- The number of elements in the linear sequence is the product of the dimensions.
- The position of each element is deterministic; for a SafeArray with dimensions d0, d1, d2, the location of an element [p0][p1][p2] is defined as:

```
pos[p0][p1][p2] = p0*d1*d2 + p1*d2 + p2
```

Consider the following example: SafeArray with dimensions 5, 8, 9.

This maps to a linear sequence with a run-time bound of 5 * 8 * 9 = 360. This gives us valid offsets 0-359. In this example, the real offset to the element at location [4][5][1] is 4*8*9 + 5*9 + 1 = 334.

## *17.2.14 Mapping for Typedefs*

ODL typedefs map directly to OMG IDL typedefs. The only exception to this is the case of an ODL enum, which is required to be a typedef. In this case the mapping is as per “Mapping for Enumerated Types” on page 17-18.

## *17.2.15 Mapping for VARIANTs*

The VARIANT data type maps to a CORBA **any**. If the VARIANT contains a DATE or CURRENCY element, these are mapped as per “Mapping for Automation Basic Data Types” on page 17-42.

## *17.2.16 Mapping Automation Exceptions to CORBA*

There are several ways in which an HRESULT (or SCODE) can be obtained by an Automation client such as the CORBA View. These ways differ based on the signature of the method and the behavior of the server. For example, for vtable invocations on dual interfaces, the HRESULT is the return value of the method. For **`IDispatch::Invoke`** invocations, the significant HRESULT may be the return value from Invoke, or may be in the EXCEPINFO parameter’s SCODE field.

Regardless of how the HRESULT is obtained by the CORBA View, the mapping of the HRESULT is the exactly the same as for COM to CORBA (see Mapping for COM Errors under “Interface Mapping” on page 16-11. The View raises either a standard CORBA system exception or the COM_HRESULT user exception.

CORBA Views must supply an EXCEPINFO parameter when making **IDispatch::Invoke** invocations to take advantage of servers using EXCEPINFO. Servers do not use the EXCEPINFO parameter if it is passed to Invoke as NULL.

An Automation method with an HRESULT return value and an argument marked as a **[retval]** maps to an IDL method whose return value is mapped from the **[retval]** argument. This situation is common in dual interfaces and means that there is no HRESULT available to the CORBA client. It would seem on the face of it that there is a problem mapping S_FALSE scodes in this case because the fact that no system exception was generated means that the HRESULT on the vtable method could have been either S_OK or S_FALSE. However, this should not truly be a problem. A method in a dual interface should never attach semantic meaning to the distinction between S_OK and S_FALSE because a Visual Basic program acting as a client would never be able to determine whether the return value from the actual method was S_OK or S_FALSE.

An Automation method with an HRESULT return value and no argument marked as **[retval]** maps to a CORBA interface with a long return value. The long HRESULT returned by the original Automation operation is passed back as the long return value from the CORBA operation.

## 17.3 Older OLE Automation Controllers

This section provides some solutions that vendors might implement to support existing and older OLE Automation controllers. These solutions are suggestions; they are strictly optional.

### 17.3.1 Mapping for OMG IDL Arrays and Sequences to Collections

Some OLE Automation controllers do not support the use of SAFEARRAYs. For this reason, arrays and sequences can also be mapped to OLE collection objects.

A collection object allows generic iteration over its elements. While there is no explicit ICollection type interface, OLE does specify guidelines on the properties and methods a collection interface should export.

```
// ODL
[odl, dual, uuid(...)]
interface DICollection: IDispatch {
   [propget] HRESULT Count([retval,out] long * count);
   [propget, id(DISPID_VALUE)] HRESULT Item([in] long index,
                [retval,out] VARIANT * retval);
   [propput, id(DISPID_VALUE)] HRESULT Item([in] long index,
                [in] VARIANT val);
   [propget, id(NEW_ENUM)] HRESULT _NewEnum(
                [retval, out] IEnumVARIANT * newEnum);
}
```

The UUID for DICollection is:

**{A8B553C9-3B72-11cf-BBFC-444553540000}**

This interface can also be implemented as generic (nondual) Automation Interface, in which case it is named DCollection and its UUID is:

**{E977F909-3B75-11cf-BBFC-444553540000}**

In controller scripting languages such as VBA in MS-Excel, the FOR...EACH language construct can automatically iterate over a collection object such as that previously described.

```
‘ Visual Basic:
Dim doc as Object
For Each doc in DocumentCollection
doc.Visible = False
Next doc
```

The specification of DISPID_VALUE as the id() for the Item property means that access code like the following is possible.

```
‘ Visual Basic:
Dim docs as Object
Set docs = SomeCollection

docs(4).Visible = False
```

Multidimensional arrays can be mapped to collections of collections with access code similar to the following.

```
‘ Visual Basic
Set docs = SomeCollection

docs.Item(4).Item(5).Visible = False
```

If the Collection mapping for OMG IDL Arrays and Sequences is chosen, then the signatures for operations accepting SAFEARRAYs should be modified to accept a VARIANT instead. In addition, the implementation code for the View wrapper method should detect the kind of object being passed.

## 17.4 Example Mappings

### 17.4.1 Mapping the OMG Naming Service to OLE Automation

This section provides an example of how a standard OMG Object Service, the Naming Service, would be mapped according to the Interworking specification.

The Naming Service provides a standard service for CORBA applications to obtain object references. The reference for the Naming Service is found by using the **`resolve_initial_references()`** method provided on the ORB pseudo-interface:

```
CORBA::ORB_ptr theORB = CORBA::ORB_init(argc, argv,
CORBA::ORBid, ev)
CORBA::Object_var obj =
    theORB->resolve_initial_references("NameService", ev);
CosNaming::NamingContext_var inital_nc_ref =
    CosNaming::NamingContext::_narrow(obj,ev);
CosNaming::Name factory_name;
factory_name.length(1);
factory_name[0].id = "myFactory";
factory_name[0].kind = "";
CORBA::Object_var objref = initial_nc_ref->resolve(factory_name, ev);
```

The Naming Service interface can be directly mapped to an equivalent OLE Automation interface using the mapping rules contained in the rest of this section. A direct mapping would result in code from VisualBasic that appears as follows.

```
Dim CORBA as Object
Dim ORB as Object
Dim NamingContext as Object
Dim NameSequence as Object
Dim Target as Object

Set CORBA=GetObject("CORBA.ORB")
Set ORB=CORBA.init("default")
Set NamingContext = ORB.resolve_initial_reference("Naming-
Service")
Set NameSequence=NamingContext.create_type("Name")
ReDim NameSequence as Object(1)
NameSequence[0].name = "myFactory"
NameSequence[0].kind = ""
Set Target=NamingContext.resolve(NameSequence)
```

### *17.4.2 Mapping a COM Service to OMG IDL*

This section provides an example of mapping a Microsoft IDL-described set of interfaces to an equivalent set of OMG IDL-described interfaces. The interface is mapped according to the rules provided in "COM to CORBA Data Type Mapping" on page 16-32 in the Mapping Com and CORBA chapter. The example chosen is the COM ConnectionPoint set of interfaces. The ConnectionPoint service is commonly used for supporting event notification in OLE custom controls (OCXs). The service is a more general version of the IDataObject/IAdviseSink interfaces.

The ConnectionPoint service is defined by four interfaces, described in Table 17-9 on page 17-52.

*Table 17-9* Interfaces of the ConnectionPoint Service

| | |
|---|---|
| IConnectionPointContainer | Used by a client to acquire a reference to one or more of an object's notification interfaces |
| IConnectionPoint | Used to establish and maintain notification connections |
| IEnumConnectionPoints | An iterator over a set of IConnectionPoint references |
| IEnumConnections | Used to iterate over the connections currently associated with a ConnectionPoint |

For purposes of this example, we describe these interfaces in Microsoft IDL. The IConnectionPointContainer interface is shown next.

```
// Microsoft IDL
   interface IConnectionPoint;
   interface IEnumConnectionPoints;
   typedef struct {
       unsigned long Data1;
       unsigned short Data2;
       unsigned short Data3;
       unsigned char Data4[8];
   } REFIID;
   [object, uuid(B196B284-BAB4-101A-B69C-00AA00241D07),
      pointer_default(unique)]
   interface IConnectionPointContainer: IUnknown
{
   HRESULT EnumConnectionPoints ([out] IEnumConnectionPoints
      **pEnum);
   HRESULT FindConnectionPoint([in] REFIID iid, [out]
      IConnectionPoint **cp);
};
MIDL definition for IConnectionPointContainer
```

This IConnectionPointContainer interface would correspond to the OMG IDL interface shown next.

```
// OMG IDL
    interface IConnectionPoint;
    interface IEnumConnectionPoints;
    struct REFIID {
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[8];
    };
    interface IConnectionPointContainer: CORBA::Composite,
    CosLifeCycle::LifeCycleObject
{
    HRESULT EnumConnectionPoints (out IEnumConnectionPoints
                pEnum) raises (COM_HRESULT);
    HRESULT FindConnectionPoint(in REFIID iid, out
                IConnectionPoint cp) raises (COM_HRESULT);
#pragma ID IConnectionPointContainer ="DCE:B196B284-BAB4-
    101A-B69C-00AA00241D07";
};
```

Similarly, the forward declared ConnectionPoint interface shown next is remapped to the OMG IDL definition shown in the second following example.

```
// Microsoft IDL
   interface IEnumConnections;
   [object, uuid(B196B286-BAB4-101A-B69C-00AA00241D07),
pointer_default(unique)]
   interface IConnectionPoint: IUnknown
{
   HRESULT GetConnectionInterface([out] IID *pIID);
   HRESULT GetConnectionPointContainer([out]
      IConnectionPointContainer **ppCPC);
   HRESULT Advise([in] IUnknown *pUnkSink, [out] DWORD
      *pdwCookie);
   HRESULT Unadvise(in DWORD dwCookie);
   HRESULT EnumConnections([out] IEnumConnections **ppEnum);
};

// OMG IDL
interface IEnumConnections;
interface IConnectionPoint:: CORBA::Composite,
    CosLifeCycle::LifeCycleObject
{
```

```
    HRESULT GetConnectionInterface(out IID pIID)
            raises (COM_HRESULT);
    HRESULT GetConnectionPointContainer
            (out IConnectionPointContainer pCPC)
            raises (COM_HRESULT);
    HRESULT Advise(in IUnknown pUnkSink, out DWORD pdwCookie)
            raises (COM_HRESULT);
    HRESULT Unadvise(in DWORD dwCookie)
            raises (COM_HRESULT);
    HRESULT EnumConnections(out IEnumConnections ppEnum)
            raises (COM_HRESULT);
#pragma ID IConnectionPoint = "DCE:B196B286-BAB4-101A-B69C-
00AA00241D07";
};
```

Finally, the MIDL definition for IEnumConnectionPoints and IEnum Connections interfaces are shown next.

```
typedef struct tagCONNECTDATA {
   IUnknown * pUnk;
   DWORD dwCookie;
} CONNECTDATA;

[object, uuid(B196B285-BAB4-101A-B69C-00AA00241D07),
   pointer_default(unique)]
interface IEnumConnectionPoints: IUnknown
{
      HRESULT Next([in] unsigned long cConnections,
         [out] IConnectionPoint **rcpcn,
         [out] unsigned long *lpcFetched);
      HRESULT Skip([in] unsigned long cConnections);
      HRESULT Reset();
      HRESULT Clone([out] IEnumConnectionPoints **pEnumval);
};
[object, uuid(B196B287-BAB4-101A-B69C-00AA00241D07),
   pointer_default(unique)]
interface IEnumConnections: IUnknown
{
      HRESULT Next([in] unsigned long cConnections,
         [out] IConnectionData **rcpcn,
         [out] unsigned long *lpcFetched);
      HRESULT Skip([in] unsigned long cConnections);
      HRESULT Reset();
      HRESULT Clone([out] IEnumConnections **pEnumval);
};
```

The corresponding OMG IDL definition for EnumConnectionPoints and EnumConnections is shown next:

```
struct CONNECTDATA {
    IUnknown * pUnk;
    DWORD dwCookie;
};
interface IEnumConnectionPoints: CORBA::Composite,
CosLifeCycle::LifeCycleObject
{
    HRESULT Next(in unsigned long cConnections,
        out IConnectionPoint rcpcn,
        out unsigned long lpcFetched) raises (COM_HRESULT);
    HRESULT Skip(in unsigned long cConnections) raises
        (COM_HRESULT);
    HRESULT Reset() raises (COM_HRESULT);
    HRESULT Clone(out IEnumConnectionPoints pEnumval)
        raises(COM_HRESULT)
#pragma ID IEnumConnectionPoints =
    "DCE:B196B285-BAB4-101A-B69C-00AA00241D07";

};

interface IEnumConnections: CORBA::Composite,
    CosLifeCycle::LifeCycleObject
{
    HRESULT Next(in unsigned long cConnections,
        out IConnectData rgcd,
        out unsigned long lpcFetched) raises (COM_HRESULT);
    HRESULT Skip(in unsigned long cConnections) raises
        (COM_HRESULT);
    HRESULT Reset() raises (COM_HRESULT);
    HRESULT Clone(out IEnumConnectionPoints pEnumVal) raises
        (COM_HRESULT);
#pragma ID IEnumConnections =
    "DCE:B196B287-BAB4-101A-B69C-00AA00241D07";
};
```

### *17.4.3 Mapping an OMG Object Service to OLE Automation*

This section provides an example of mapping an OMG-defined interface to an equivalent OLE Automation interface. This example is based on the OMG Naming Service and follows the mapping rules from the Mapping: OLE Automation and CORBA chapter. The Naming Service is defined by two interfaces and some associated

types, which are scoped in the OMG IDL CosNaming module.

*Table 17-10* Interfaces of the OMG Naming Service

| Interface | Description |
|---|---|
| CosNaming::NamingContext | Used by a client to establish the name space in which new associations between names and object references can be created, and to retrieve an object reference that has been associated with a given name. |
| CosNaming::BindingIterator | Used by a client to walk a list of registered names that exist within a naming context. |

Microsoft ODL does not explicitly support the notions of modules or scoping domains. To avoid name conflicts, all types defined in the scoping space of *CosNaming* are expanded to global names.

The data type portion (interfaces excluded) of the *CosNaming* interface is shown next.

```
// OMG IDL
module CosNaming{
    typedef string Istring;
    struct NameComponent {
    Istring id;
    Istring kind;
};
    typedef sequence <NameComponent> Name;
    enum BindingType { nobject, ncontext };
    struct Binding {
            Name binding_name;
            BindingType binding_type;
};
    typedef sequence <Binding> BindingList;
    interface BindingIterator;
    interface NamingContext;
    // ...
}
```

The corresponding portion (interfaces excluded) of the Microsoft ODL interface is shown next.

```
[uuid(a1789c86-1b2c-11cf-9884-08000970dac7)] // from COMID
association
 library CosNaming
 {
   importlib("stdole32.tlb");
   importlib("corba.tlb"); / for standard CORBA types
   typedef CORBA_string CosNaming_Istring;
   [uuid((04b8a791-338c-afcf-1dec-cf2733995279), help-
string("struct NameComponent"),
         oleautomation, dual]
   interface CosNaming_NameComponent: ICORBAStruct {
   [propget] HRESULT id([out, retval]CosNaming_Istring
**val);
   [propput] HRESULT id([in]CosNaming_IString* val);
   [propget] HRESULT kind([out, retval]CosNaming_Istring
   ** val);
   [propget] HRESULT kind([in]CosNaming_Istring *val);
};
# define Name SAFEARRAY(CosNaming_NameComponent *)
   // typedef doesn't work
   typedef enum { [helpstring("nobject")]nobject,
      [helpstring("ncontext")]ncontext
} CosNaming_BindingType;
#define CosNaming_BindingList SAFEARRAY(CosNaming_Binding *)
   [uuid(58fbe618-2d20-d19f-1dc2-560cc6195add),
      helpstring("struct Binding"),
      oleautomation, dual]
   interface DICosNaming_Binding: ICORBAStruct {
   [propget] HRESULT binding_name([retval, out]
      CosNaming_IString ** val);
   [propput] HRESULT binding_name([in]
      CosNaming_IString * vall);
   [propget] HRESULT binding_type([retval, out]
      CosNaming_BindingType *val);
   [propset] HRESULT binding_type([in]
      CosNaming_BindingType val);
};
#define CosNaming_BindingList SAFEARRAY(CosNaming_Binding)
   interface DICosNaming_BindingIterator;
   interface DICosNaming_NamingContext;
   // ...
};
```

The types scoped in an OMG IDL interface are also expanded using the notation [<modulename>_]*[<interfacename>_]typename. Thus the types defined within the *CosNaming::NamingContext* interface (shown next) are expanded in Microsoft ODL as shown in the second following example.

```
module CosNaming{
// ...
   interface NamingContext
```

```
    {
        enum NotFoundReason { missing_node, not_context,
        not_object };
        exception NotFound {
            NotFoundReason why;
            Name rest_of_name;
    };
    exception CannotProceed {
        NamingContext cxt;
        Name rest_of_name;
    };
    exception InvalidName {};
    exception AlreadyBound {};
    exception NotEmpty {};
    void bind(in Name n, in Object obj)
        raises( NotFound, CannotProceed, InvalidName,
        AlreadyBound );
    void rebind(in Name n, in Object obj)
        raises( NotFound, CannotProceed, InvalidName );
    void bind_context(in Name n, in NamingContext nc)
        raises( NotFound, CannotProceed, InvalidName,
        AlreadyBound );
    void rebind_context(in Name n, in NamingContext nc)
        raises( NotFound, CannotProceed, InvalidName );
    Object resolve(in Name n)
        raises( NotFound, CannotProceed, InvalidName );
    void unbind(in Name n)
        raises( NotFound, CannotProceed, InvalidName );
    NamingContext new_context();
    NamingContext bind_new_context(in Name n)
        raises( NotFound, AlreadyBound, CannotProceed, InvalidName );
    void destroy()
        raises( NotEmpty );
    void list(in unsigned long how_many,
        out BindingList bl, out BindingIterator bi );
    };
// ...
};

[uuid(d5991293-3e9f-0e16-1d72-7858c85798d1)]
library CosNaming
 {// ...
   interface DICosNaming_NamingContext;
   [uuid(311089b4-8f88-30f6-1dfb-9ae72ca5b337),
         helpstring("exception NotFound"),
         oleautomation, dual]
 interface DICosNaming_NamingContext_NotFound:
   ICORBAException {
   [propget] HRESULT why([out, retval] long* _val);
   [propput] HRESULT why([in] long _val);
   [propget] HRESULT rest_of_name([out, retval]
```

```
                    CosNaming_Name ** _val);
   [propput] HRESULT rest_of_name([in] CosNaming_Name
      * _val);
};
   [uuid(d2fc8748-3650-cedd-1df6-026237b92940),
         helpstring("exception CannotProceed"),
         oleautomation, dual]
   interface DICosNaming_NamingContext_CannotProceed:
         DICORBAException{
   [propget] HRESULT cxt([out, retval]
         DICosNaming_NamingContext ** _val);
   [propput] HRESULT cxt([in] DICosNaming_NamingContext
      * _val);
   [propget] HRESULT rest_of_name([out, retval]
         CosNaming_Name ** _val);
   [propput] HRESULT rest_of_name([in] CosNaming_Name *
_val);
};
   [uuid(7edaca7a-c123-42a1-1dca-a7e317aafe69),
         helpstring("exception InvalidName"),
         oleautomation, dual]
   interface DICosNaming_NamingContext_InvalidName:
         DICORBAException {};
   [uuid(fee85a90-1f6b-c47a-1dd0-f1a2fc1ab67f),
         helpstring("exception AlreadyBound"),
         oleautomation, dual]
interface DICosNaming_NamingContext_AlreadyBound:
      DICORBAException {};
   [uuid(8129b3e1-16cf-86fc-1de4-b3080e6184c3),
         helpstring("exception NotEmpty"),
         oleautomation, dual]
interface CosNaming_NamingContext_NotEmpty:
      DICORBAException {};
typedef enum {[helpstring("missing_node")]
   NamingContext_missing_node,
      [helpstring("not_context") NamingContext_not_context,
      [helpstring("not_object") NamingContext_not_object
} CosNaming_NamingContext_NotFoundReason;
   [uuid(4bc122ed-f9a8-60d4-1dfb-0ff1dc65b39a),
         helpstring("NamingContext"),
         oleautomation,dual]
   interface DICosNaming_NamingContext {
   HRESULT bind([in] CosNaming_Name * n, [in] IDispatch *
obj,
      [out, optional] VARIANT * _user_exception);
   HRESULT rebind([in] CosNaming_Name * n, in] IDispatch *
obj,
      [out, optional] VARIANT * _user_exception);
   HRESULT bind_context([in] CosNaming_Name * n,
      [in] DICosNaming_NamingContext * nc,
      [out, optional] VARIANT * _user_exception);
```

```
   HRESULT rebind_context([in] CosNaming_Name * n,
      [in] DICosNaming_NamingContext * nc,
      [out, optional ] VARIANT * _user_exception);
   HRESULT resolve([in] CosNaming_Name * n,
      [out, retval] IDispatch** pResult,
      [out, optional] VARIANT * _user_exception)
   HRESULT unbind([in] CosNaming_Name * n,
      [out, optional] VARIANT * _user_exception);
   HRESULT new_context([out, retval]
DICosNaming_NamingContext ** pResult);
   HRESULT bind_new_context([in] CosNaming_Name * n,
      [out, retval] DICosNaming_NamingContext ** pResult,
      [out, optional] VARIANT * _user_exception);
   HRESULT destroy([out, optional] VARIANT*
_user_exception);
   HRESULT list([in] unsigned long how_many, [out]
CosNaming_BindingList ** bl,
      [out] DICosNaming_BindingIterator ** bi);
   };
};
```

The *BindingIterator* interface is mapped in a similar manner, as shown in the next two examples.

```
module CosNaming {
   //...
   interface BindingIterator {
   boolean next_one(out Binding b);
   boolean next_n(in unsigned long how_many,
   out BindingList bl);
   void destroy();
   };
};
```

```
[uuid(a1789c86-1b2c-11cf-9884-08000970dac7)]
library CosNaming
 {// ...
   [uuid(5fb41e3b-652b-0b24-1dcc-a05c95edf9d3),
   help  string("BindingIterator"),
   helpcontext(1), oleautomation, dual]
   interface DICosNaming_IBindingIterator: IDispatch {
      HRESULT next_one([out] DICosNaming_Binding ** b,
         [out, retval] boolean* pResult);
      HRESULT next_n([in] unsigned long how_many,
         [out] CosNaming_BindingList ** bl,
         [out, retval] boolean* pResult);
      HRESULT destroy();
   };
}
```