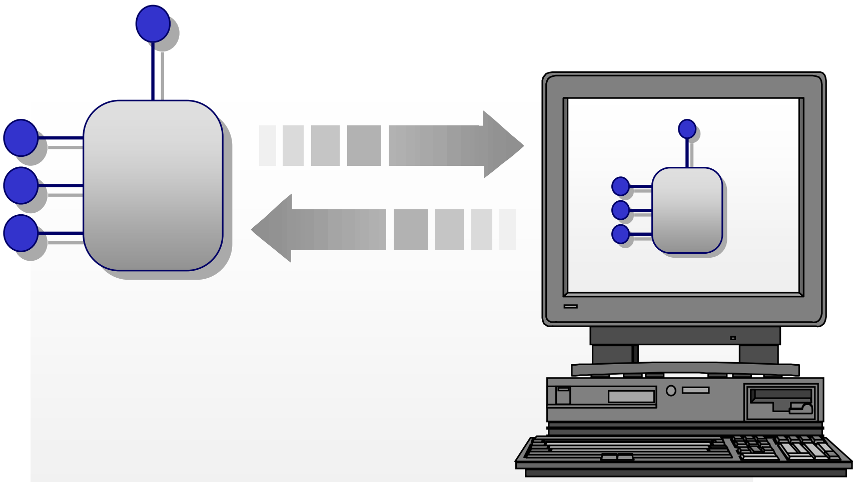
Automation



COM Programming

133

Chapter Overview

QA-IQ

- **Objectives**
  - **Explain the technology behind Automation**
  - **Build a simple Automation object using ATL**
- **Chapter content**
  - **IDispatch**
  - **Dual interfaces**
  - **Error handling**
- **Practical content**
  - **Use a scripting client**
  - **Experiment with dual interfaces**
- **Summary**

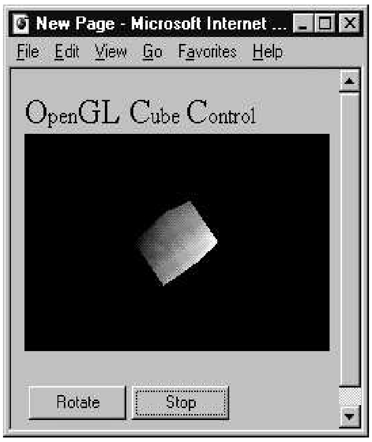
134

What is Automation?

QA-IQ

- **Allows one application to drive another**
  - **Server can expose one or more objects**
  - **Controller can create and manipulate these objects**
  - **Controller can be written in a scripting-language such as VBScript**

```
<SCRIPT LANGUAGE="VBSCRIPT">
Sub RotateBtn_OnClick
    VTOpenGL1.xDegrees=5
    VTOpenGL1.yDegrees=5
    VTOpenGL1.zDegrees=5
    VTOpenGL1.Rotate 1
End Sub
</SCRIPT>
```



135

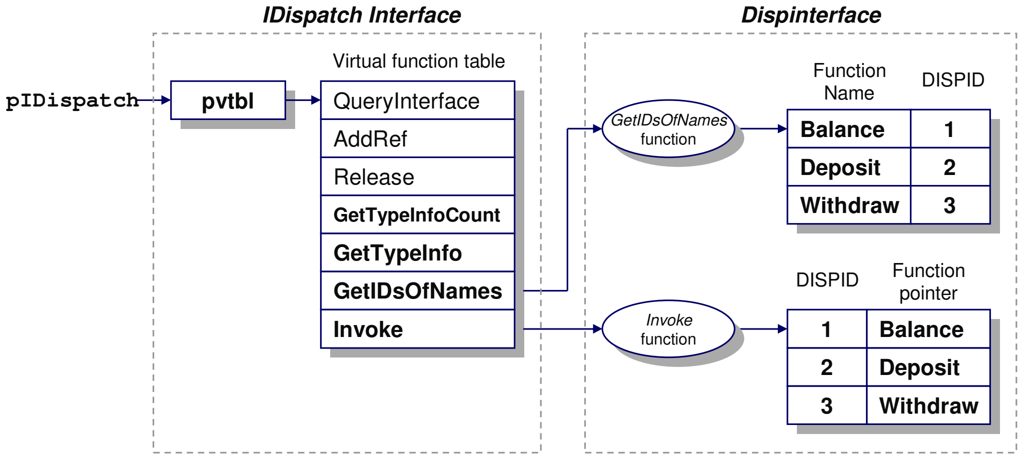
Automation is the technology that allows an application to expose a set of programmable objects to the outside world. Typically this manifests itself in the form of one application driving another, such as Microsoft Word driving Microsoft Excel.

In a Web browser, automation is used by scripting languages such as VBScript of JavaScript to drive ActiveX Controls on a web page or the browser itself.

Automation is supported through a single standard COM interface, `IDispatch`. Amazingly enough, given the sheer number of automatable COM objects in the world, and the number of programmers writing them, `IDispatch` has only *seven* methods, and we know the first three of those.

How Does it Work?

- **A COM object can offer services through a single standard interface called IDispatch**
  - **A "gateway" to a collection of methods called a dispinterface**
  - **Allows a client to execute a function by name**
  - **Allows a scripting-language client to use object**



136

It is important to distinguish between an automation interface (also called a dispatch interface or *dispinterface*) and a COM interface.

Previously, we discussed how COM interfaces are implemented as an array of function pointers, also known as a *vtable*. The interface pointer stored by client code is actually a pointer to a pointer to a vtable. Client code calls a COM interface method by calling the function referenced by the pointer at a given offset in the vtable. No run-time type checking is carried out as it is performed by the compiler.

All automation interfaces are implemented with the `Invoke()` method of the `IDispatch` interface, which is, of course, a COM interface method like any other. The parameters of `IDispatch::Invoke()` include a *dispatch ID* (DISPID), which identifies which property or method is being invoked and an array of parameters to that property or method.

`IDispatch::Invoke()` can be implemented in various ways, but the mapping of dispatch IDs to handler functions is commonly called a *dispatch table*. Notice that while offsets into a vtable must necessarily be consecutive, this is not true of dispatch IDs; we'll see later that certain values are reserved for standard properties.

Dispatch IDs

- **Each method, property and method argument is identified by a locale-independent DISPID**
  - **A LONG number, not a GUID**
  - **Unique only to a specific implementation of IDispatch**
  - **Specified in IDL using id() attribute**
  - **Method-argument DISPIDs are assigned automatically**

```
import "oaidl.idl";
import "ocidl.idl";
[
  uuid(xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx),
]
dispinterface IAccount
{
properties:
  [id(1), readonly] float Balance;
methods:
  [id(2)] HRESULT Deposit([in] float amount);
  [id(3)] HRESULT Withdraw([in] float amount);
};
```

Defines Automation types, etc.

Defines a pure Automation interface - implicitly inherits from `IDispatch`

137

This slide shows how to define a pure Automation interface in IDL. The particular syntax has its roots in ODL.

You will notice that the format of the `dispinterface` is very different from that of a standard COM interface. In particular, the syntax for properties resembles that of normal member variables in C++.

The key attribute for methods and properties are the dispatch identifiers, which are used in the `Invoke()` method to determine which function or property is being called. These numbers need to be positive (negative numbers are reserved by Microsoft), but only need to be unique within the scope of a single `dispinterface`.

IDispatch

QA-IQ

```
interface IDispatch : IUnknown {
  HRESULT GetTypeInfoCount([out] UINT *pctinfo);

  HRESULT GetTypeInfo([in] UINT iTinfo,
                      [in] LCID lcid,
                      [out] ITypeInfo **ppTinfo);

  HRESULT GetIDsOfNames([in] REFIID riid,
                        [in, size_is(cNames)] LPOLESTR *rgszNames,
                        [in] UINT cNames,
                        [in] LCID lcid,
                        [out, size_is(cNames)] DISPID *rgDispid);

  HRESULT Invoke([in] DISPID dispIdMember,
                 [in] REFIID riid,
                 [in] LCID lcid,
                 [in] WORD wFlags,
                 [in, out] DISPPARAMS *pDispParams,
                 [out] VARIANT *pVarResult,
                 [out] EXCEPINFO *pExcepInfo,
                 [out] UINT *puArgErr);
};
```

Is there any type information?

Give me the type information

Give me the DISPIDs of these methods or properties

Execute the function or property identified by this DISPID

138

`GetTypeInfoCount()` is the standard way in which an automation controller can query an automation object to find out if there is any type information available. If there is, the controller can get hold of this type information via `GetTypeInfo()`. As can be imagined, `GetTypeInfo()` is normally implemented using the type library.

The controller may have an automation method that it wants to execute on the object. To do this the controller needs to pass a `dispID` to `Invoke()`, so it calls `GetIDsOfNames()`, passing amongst other things, the method name and a pointer to a `dispID` variable that will be filled out by the object.

Having retrieved the `dispID`, the controller can then call `Invoke()` passing any parameters necessary.

GetIDsOfNames() Arguments

QA-IQ

Reserved. Must be IID_NULL

Array of function names

Number of names

```
GetIDsOfNames([in] REFIID riid,
              [in, size_is(cNames)] LPOLESTR *rgszNames,
              [in] UINT cNames,
              [in] LCID lcid,
              [out, size_is(cNames)] DISPID *rgDispid);
```

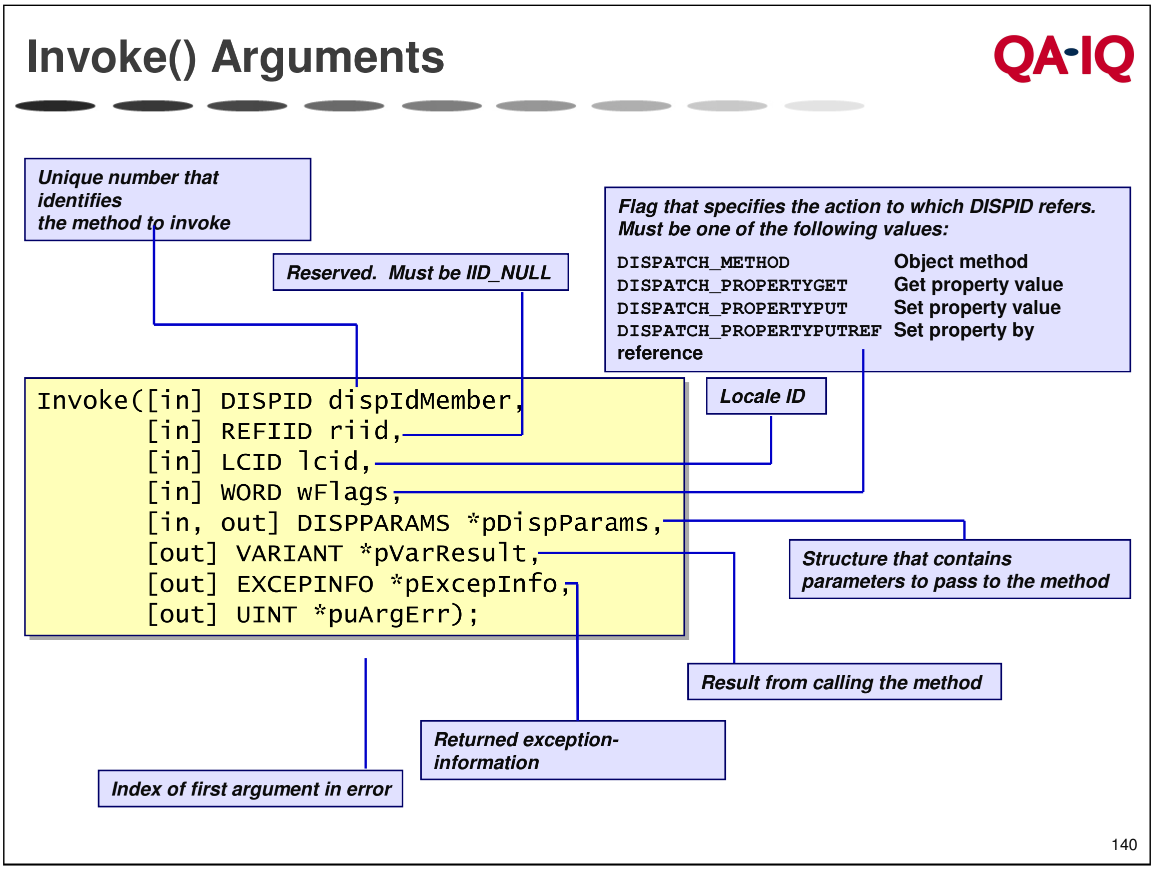
Identifies a locale for national language support

Array of dispatch IDs

139

Prior to calling the `Invoke()` method, `GetIDsOfNames()` is used to retrieve one or more of the required dispatch IDs. An array of names is passed in, with a matching array of `DISPID`s (Note the use of the `size_is()` attribute on both arrays).

Given that we are asking for methods or properties by name, we might be dealing with different lookup tables for different national languages. Therefore, a *locale* identifies a language and optionally a country, e.g. US English, French-speaking Canada. In many cases, this locale ID will be `LOCALE_USER_DEFAULT`.



Typically, an automation object will support a combination of methods and properties. The controller identifies which method to invoke or which property to access via the `dispID` passed as the first parameter to `IDispatch::Invoke()`. The automation object's implementation of `IDispatch::Invoke()` will determine what happens next. All negative dispIDs are reserved for Microsoft use. The following negative dispIDs, amongst others, are defined;

`DISPID_VALUE (0)` - default member
`DISPID_UNKNOWN (-1)` - Unknown member or parameter name
`DISPID_PROPERTYPUT (-3)` - Indicates parameter in put operation
`DISPID_NEWENUM (-4)` - Used in collections
`DISPID_EVALUATE (-5)` - Used when controller finds `[]`

The `dispID` identifies what method or what property, but we also need to identify whether this is a method operation or a property operation, and this is the purpose of the `wFlags` parameter.

The method parameters are passed in a `DISPARAMS` structure, which is defined as follows:

```
typedef struct tagDISPPARAMS {
  // array of arguments
  [size_is(cArgs)] VARIANTARG *rgvarg;

  // DISPIDs of named arguments
  [size_is(cNamedArgs)] DISPID *rgdispidNamedArgs;

  UINT cArgs;        // number of arguments
  UINT cNamedArgs;   // number of named arguments
} DISPPARAMS;
```


A Simple C++ Client

```
// error checking deleted for clarity

// create the component
IDispatch *pDisp = NULL;
CoCreateInstance( CLSID_Account, NULL, CLSCTX_INPROC_SERVER,
                  IID_IDispatch, (void**)&pDisp );

// get DISPID for dispatch function Beep()
DISPID dispId;
LPOLESTR pNames = L"Beep";
hr = pDisp->GetIDsOfNames( IID_NULL, &pNames, 1,
                           LOCALE_USER_DEFAULT, &dispId );

// lucky no arguments!
DISPPARAMS dispParams = {NULL, NULL, 0, 0};

// invoke the function
pDisp->Invoke( dispId, IID_NULL, LOCALE_USER_DEFAULT,
               DISPATCH_METHOD, &dispParams, NULL, NULL,
               NULL );
```

141

In this example, we have assumed that the client knows about the function `Beep()` and therefore does need to obtain type information. Also, because the function `Beep()` has no arguments, it is a trivial exercise to initialise the *dispParams* parameter of `Invoke()`.

You can imagine that calling the `Beep()` method this way would be much slower than calling a corresponding function through a normal vtable mechanism.

A More Complex C++ Client

```
// allocate and initialise a VARIANT argument
VARIANTARG varg;
VariantInit( &varg );
varg.vt = VT_BSTR;
varg.bstrVal = SysAllocString( L"Test string from client" );

// fill in a DISPPARAMS structure and invoke the function
DISPPARAMS dispParams;
dispParams.cArgs = 1;        // one argument
dispParams.rgvarg = &varg;   // pointer to argument
dispParams.cNamedArgs = 0;   // no named arguments
dispParams.rgdispidNamedArgs = NULL;

pIDispatch->Invoke( dispID, IID_NULL, LOCALE_USER_DEFAULT,
                    DISPATCH_METHOD, &dispParams, NULL,
                    NULL, NULL );
// clean up
VariantClear( &varg );
```

142

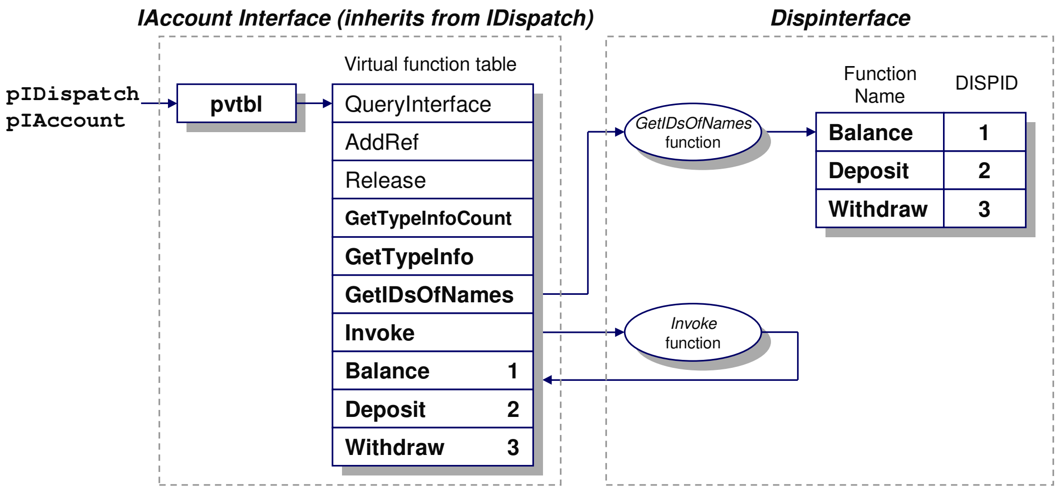
Note that the above code could be simplified by using the `CComVariant` wrapper class. `CComVariant` ensures that the `VARIANT` is initialised with `VariantInit()` before it is used, and cleared with `VariantClear()` when it is finally released. It also provides overloaded assignment operators for many types that `VARIANT` can handle. For more information on `CComVariant`, see the Visual C++ documentation.

One aspect of Visual Basic programming is its ability to handle *named* parameters. Consequently, a locale identifier has to be passed in to the `Invoke()` method, so that these named parameters can be translated!

We haven't shown the error handling code here. Suffice to say that an `EXCEPINFO` structure is declared on the stack, and is then passed by reference to the call. It will be populated by the COM object if an error occurs (the returned `HRESULT` is `DISP_E_EXCEPTION`). You can then check the contents of the `EXCEPINFO` structure for more information.

Dual Interfaces

- **A dual interface is one that inherits from IDispatch**
  - **Scripting-language clients can call methods through Invoke()**
  - **C++ and Visual Basic clients can call same methods through vtable (for best performance)**



143

A COM interface offers vtable-based, early-bound, strongly typed access to functionality in a COM object. A custom interface is merely a COM interface that you define, rather than one that Microsoft define.

`IDispatch` is a COM interface and is vtable-based, early-bound and strongly typed. However, through its methods, such as `Invoke()`, it offers a more "interpreted", late-bound access to a programmable interface using run-time type checking and/or coercion. A programmable interface is a set of methods and properties that you define.

A dual interface is derived from `IDispatch` and it extends `IDispatch` with the methods and properties of the programmable interface. Because it is derived from `IDispatch` it can still be used as a dispatch interface, but for any controller that is able to, it can be used as a custom interface.

A dual interface provides for both the speed of direct vtable binding and the flexibility of `IDispatch` binding. For this reason, dual interfaces are recommended whenever possible.

Dual interfaces do have some restrictions. All methods, including property put and get methods must return `HRESULT`s. Also all parameters must be Automation compatible. This basically means that they must be capable of being represented by a `VARIANT`. Finally, supporting multiple dual interfaces on an object requires additional work beyond that which is done for you by ATL.

Automation-Compatible Types

QA-IQ

Type	IDL Type Name	Representation
integer	`short` `long`	16-bit signed integer 32-bit signed integer
floating point	`float` `double`	32-bit IEEE floating point number 64-bit IEEE floating point number
boolean	`VARIANT_BOOL`	Unsigned char (TRUE or FALSE)
character	`unsigned char`	8-bit unsigned data item
string	`BSTR`	Length-prefixed string
array	`SAFEARRAY(type)`	Bounded array of type
status	`HRESULT`	Built-in error type
interface pointers	`IUnknown *` `IDispatch *`	Pointer to interface other than `IDispatch` Pointer to `IDispatch` interface

144

These are all types that can be put into a `VARIANT`. In addition, the following types are supported: `enum`, currency and DATE.

`VARIANT`s can also contain arrays of the above types.

It is worth noting the special `SAFEARRAY`(*type*) syntax. A requirement of the Automation marshalling code (more later!) is that it must be able to understand the types inside a SafeArray.

Example of a Dual Interface

Import Automation types

```
// SimpleAccount.idl ----------------------
import "oaidl.idl";
import "ocidl.idl";
[
  object,
  uuid(81B8FDDC-FB37-11D2-9C81-000000000000),
  dual,
  pointer_default(unique)
]
interface IAccount : IDispatch
{
  [id(1)] HRESULT Deposit([in] float amount);
  [id(2)] HRESULT Withdraw([in] float amount);
  [propget, id(3)]
  HRESULT Balance([out, retval] float *pVal);
};
```

dual keyword

Return **HRESULT**

DispID

Read-only property

Inherit from **IDispatch**

Automation-compatible type

Return value

145

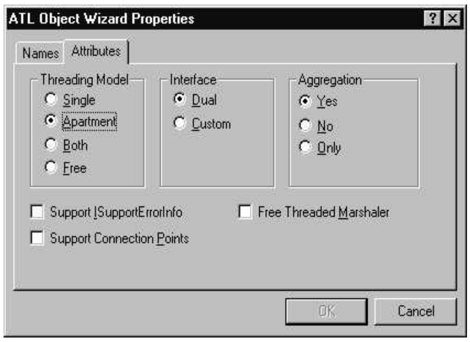
A dual interface has a syntax that is more akin to a normal, custom COM interface. The major differences are the fact that it is derived from `IDispatch`, and that all the methods and properties must have dispatch IDs.

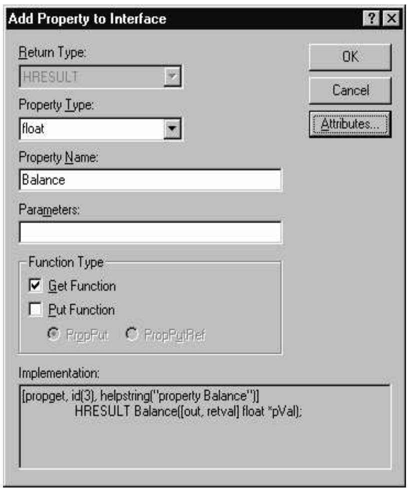
Additionally, the interface is marked with the attribute `dual`. This tells the MIDL compiler to check all of the parameters to ensure that they are all automation compatible.

Building an Automation Object

QA-IQ

- **Easy using ATL**
  - **By default, ATL Object Wizard defines an object with a dual interface**





DispIDs are assigned automatically when adding methods and properties through *ClassView*

146

By default, when you create a COM object with the ATL Object Wizard, its interface is set to be a dual interface. This is because it addresses the majority of COM clients. This can easily be changed from the *Attributes* tab of the wizard, if so desired.

Thereafter, all dispatch IDs are generated automatically for all new methods and properties.

Code Generated by Object Wizard

QA-IQ

Account.h

```
// Account.h : Declaration of the CAccount

...

// CAccount
class ATL_NO_VTABLE CAccount :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAccount, &CLSID_Account>,
    public IDispatchImpl<IAccount, &IID_IAccount,
                              &LIBID_SIMPLEACCOUNTLib> {

...


BEGIN_COM_MAP(CAccount)
    COM_INTERFACE_ENTRY(IAccount)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

...
```

147

The wondrous part of ATL's support for `IDispatch` and dual interfaces is that you don't have to do a thing! ATL provides an implementation class for `IDispatch` that uses the type library to support all of the methods.

You will also notice that the wizard adds `IDispatch` to the COM interface map (without which the VB Script and JavaScript clients couldn't talk to the object).

IDispatchImpl<>

QA·IQ

atlcom.h

```
template <class T,
          const IID* piid,
          const GUID* plibid = &CComModule::m_libid,
          WORD wMajor = 1,
          WORD wMinor = 0,
          class tihclass = CComTypeInfoHolder>
class ATL_NO_VTABLE IDispatchImpl : public T {
public:
  typedef tihclass _tihclass;

// IDispatch
  STDMETHOD(GetTypeInfoCount)(UINT* pctinfo) {
    *pctinfo = 1;
    return S_OK;
  }

  STDMETHOD(GetTypeInfo)(UINT itinfo, LCID lcid,
                         ITypeInfo** pptinfo) {
    return _tih.GetTypeInfo(itinfo, lcid, pptinfo);
  }
  ...
```

148

There are six template parameters to `IDispatchImpl<>`, but four of them have default values. The first is the abstract base class for the interface that will be exposed through this `IDispatchImpl`, while the second is the interface ID. The third parameter is the LIBID of the type library that contains the type information for the interface. The next two parameters are the major and minor version numbers, respectively, of the type library. The default value is 1 for major and 0 for minor. The final parameter is the class of a type info holder. This class defaults to `CComTypeInfoHolder`.

All of the `IDispatch` methods, with the exception of `GetTypeInfoCount()`, as implemented by `IDispatchImpl<>` are delegated to the class defined in the final template parameter.

`CComTypeInfoHolder` is used to manage type libraries at run time. The `IDispatchImpl<>` implementation delegates `Invoke()`, `GetTypeInfo()` and `GetIDsOfNames()` to `CComTypeInfoHolder` by default. `CComTypeInfoHolder` in turn delegates `Invoke()` and `GetIDsOfNames()` to the `ITypeInfo` methods of the same name.

Summary

QA-IQ

- An Automation object must implement IDispatch or provide a dual interface
- IDispatch is a standard interface
    - GetTypeInfoCount()
    - GetTypeInfo()
    - GetIDsOfNames()
    - Invoke()
- A dual interface is one that inherits from IDispatch
    - Scripting-language clients can call methods through Invoke()
    - C++ and Visual Basic clients can call same methods through vtable
- Only data that can be held in a VARIANT can be passed via a dispatch interface
- By default, ATL Object Wizard defines an object with a dual interface

149