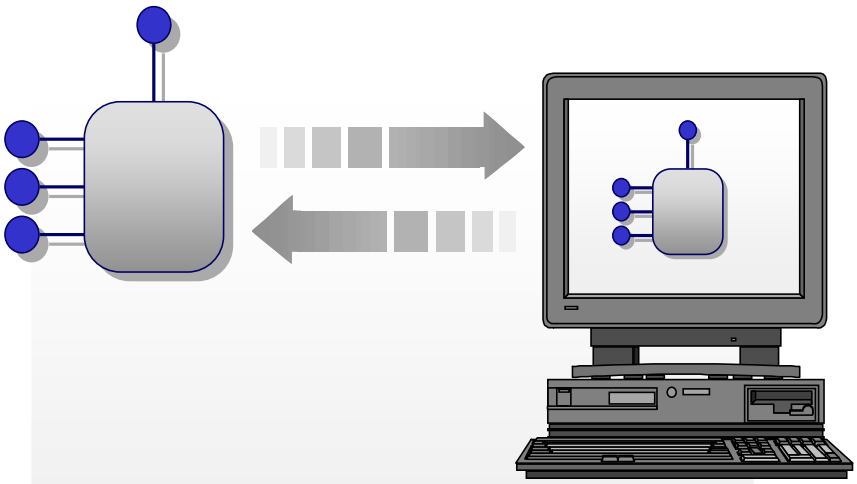
Events and Callbacks

QA-IQ



COM Programming

173

Chapter Overview

QA-IQ

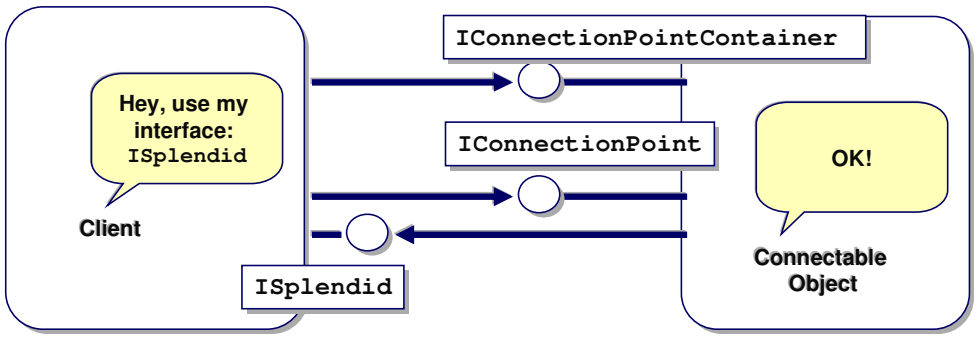
- Objectives
  - Explain the terminology of connectable objects
  - Implement a connectable object using ATL
- Chapter content
  - What is a connectable object?
  - Outgoing interfaces
  - Connection points
  - IConnectionPoint and IConnectionPointContainer interfaces
  - Connection identifiers
  - Enumeration
  - Using ATL
- Practical content
  - Write a COM object that supports Connection Points
- Summary

174

What are Connectable Objects For?

QA-IQ

- **The standard mechanism for a client object to ask an object server to use one of its interfaces**
- **Used by ActiveX controls to achieve event firing from control to container**
- **May also be used for any situation where two way communication is needed**



175

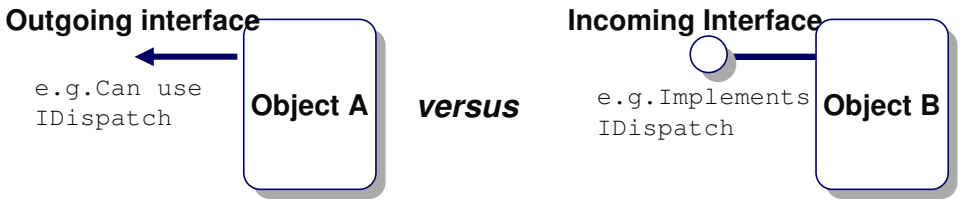
A client object will often want an object server to use one of its incoming interfaces. For example, this happens with a data object client and `IDataObject::DAdvise()`. In this example, the client supports the incoming interface `IAdviseSink` and wants the data object to use it. Once the data object has been given the client's `IAdviseSink` interface, it will call `IAdviseSink::OnDataChange()` when its data changes. Other interfaces such as `IOleObject` and `IViewObject2` have similar members that achieve the same goal.

So, Microsoft came up with connectable objects, a standard extensible technology that abstracts this kind functionality. A connectable object is one that supports not only incoming interfaces, but also outgoing interfaces. This chapter will guide you through the connectable object technology.

What is a Connectable Object?

QA-IQ

- One that provides outgoing interface(s)
  - Can use a given interface e.g. IDispatch
- Technically, one that:
  - Implements IConnectionPointContainer
  - Has created at least one lightweight object that implements IConnectionPoint



176

Interfaces that you are familiar with so far have been examples of *incoming interfaces*. An incoming interface is one that can be queried for by the client object using `QueryInterface()`. If the server object supports the interface being queried for, then the client can use the server object via the methods of that interface.

An *outgoing interface* is the opposite of an incoming interface: it is the capability of an object to make use of an interface on another object - it's like the client saying to the server object

"I have an `IDispatch` interface, if you have the capability to use it, then here it is…connect to it and make use of it"

A connectable object is therefore one that implements outgoing interfaces.

IConnectionPointContainer

QA-IQ

- Implemented by the connectable object
  - The way the client obtains an IConnectionPoint
- A successful query for this interface indicates the presence of outgoing interfaces on the object
- Provides the base mechanism for a client to query the connectable object about its outgoing interfaces

```
interface IConnectionPointContainer : IUnknown {
 HRESULT FindConnectionPoint(REFIID, IConnectionPoint **);
 HRESULT EnumConnectionPoints(IEnumConnectionPoints **);
};
```

- FindConnectionPoint() - a means of querying for a specific outgoing interface
- EnumConnectionPoints() - a means of finding out all outgoing interfaces

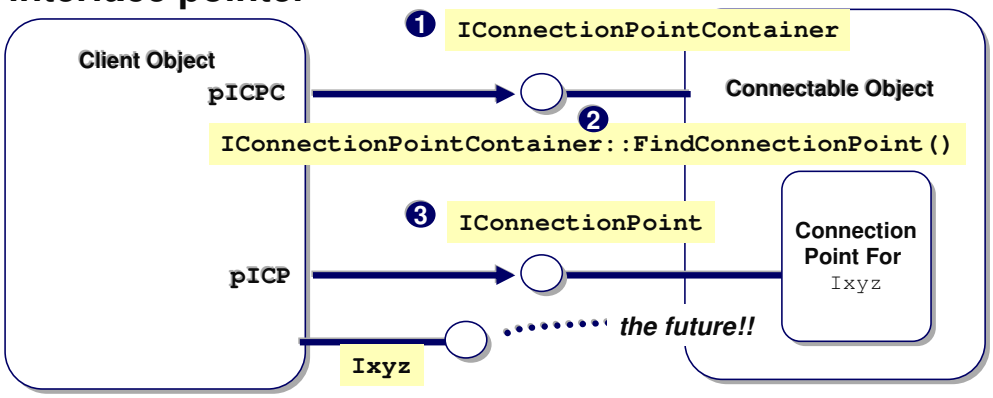
177

`IConnectionPointContainer` is the interface a connectable object implements to support outgoing interfaces. It provides the mechanism for a client to find out what outgoing interfaces the connectable object supports. Primarily, this interface provides a mechanism for the client object to obtain an `IConnectionPoint` interface.

Using IConnectionPointContainer

QA-IQ

- **Example of using an IConnectionPointContainer interface pointer**



1. Client queries for `IConnectionPointContainer`
   e.g. `pIUnknown->QueryInterface(IConnectionPointContainer, pICPC);`
2. Client queries for a specific outgoing interface
   e.g. `pICPC->FindConnectionPoint(IID_Ixyz, pICP);`
3. Client now has a valid `IConnectionPoint` pointer e.g. through `pICP`

178

A client wanting to hook up its `Ixyz` interface to one of the connectable objects outgoing interfaces must first query the connectable object for `IConnectionPointContainer`, using `QueryInterface()`. If successful, then the client can then take one of two routes. It can either:

a) ask for a specific connection point or

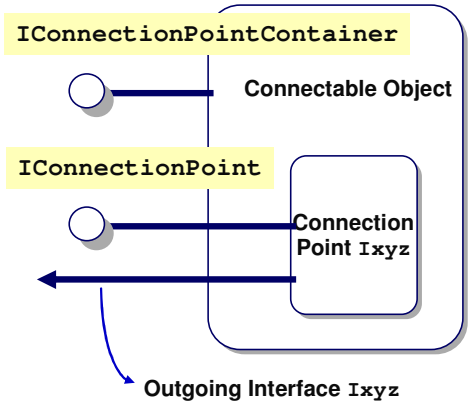
b) find out all the connection points

The example diagram shows the first option. The client in the diagram wants the connectable object to connect to its incoming interface `Ixyz`, so it queries the connectable object through `IConnectionPointContainer::FindConnectionPoint(IID_Ixyz, pICP)`. The stage is now set for the next phase: to make the connectable object connection point hook up to the client's `Ixyz` interface…the future!!

Question: What is a Connection Point?

QA-IQ

- **Answer: a lightweight component object:**
  - That implements IConnectionPoint
  - That has one outgoing interface e.g. Ixyz
  - That has no CLSID
  - That is created by the containing connectable object
  - Whose lifetime is nested in the connectable object's lifetime
  - Will have its outgoing interface connected to a sink's incoming interface in the future



179

A connection point is implemented as a component object without a CLSID. Each connection point implements the `IConnectionPoint` interface and an outgoing interface. The members of `IConnectionPoint` facilitate the mating of the outgoing interface of the connection point with the sink interface of the client.

IConnectionPoint

QA-IQ

- Implemented by connection point objects

```
interface IConnectionPoint: IUnknown {
    HRESULT GetConnectionInterface(IID *);
    HRESULT GetConnectionPointContainer(IConnectionPointContainer **);
    HRESULT Advise(IUnknown *, DWORD *);
    HRESULT Unadvise(DWORD);
    HRESULT EnumConnections(IEnumConnections **);
};
```

- GetConnectionInterface()
  - Return the outgoing interface identifier e.g. IID_QContinuum
- GetConnectionPointContainer()
  - A means of getting to the connectable object
- Advise()
  - Hooks the client's incoming interface to the connection points outgoing interface
- Unadvise()
  - Tells the connection point to stop using the client's incoming interface
- EnumConnections()
  - Allows client to find out about all other connections

180

The two key members of `IConnectionPoint` are `Advise()` and `Unadvise()`. Advise is the method through which a client passes the interface pointer it wants the connectable object to make use of. This establishes a connection. `Unadvise()` is the method that the client uses to tell the connectable object to stop using a given connection.

Sometimes, a client will want to get to another connection point. The connectable object manages this information through `IConnectionPointContainer`, so `IConnectionPoint::GetConnectionPointContainer()` returns the `IConnectionPointContainer` interface.

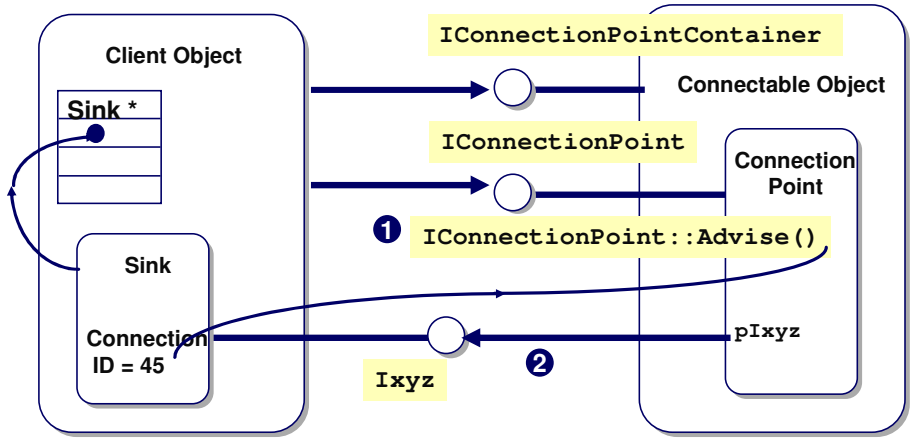
If a client has enumerated a connectable object with `IConnectionPointContainer::EnumConnectionPoints()` to get a list of all the connection points supported, it will have an array or linked list of `IConnectionPoint` interface pointers. The client will not know what those outgoing interfaces are, so `IConnectionPoint::GetConnectionInterface()` will return the outgoing interface identifier for a given connection point.

`IConnectionPoint::EnumConnections()` will be discussed later

Connecting Them Up

QA-IQ

- **Example of using IConnectionPoint interface**



1. Client passes `Ixyz` to connection point and obtains connection ID
   e.g. `pICP->Advise(pUnkSink, pdwCookie);`
2. Connection Point treats `Ixyz` as `IUnknown`, Querying for `Ixyz`
   e.g. `pUnkSink->QueryInterface(IID_Ixyz, pIxyz);`

181

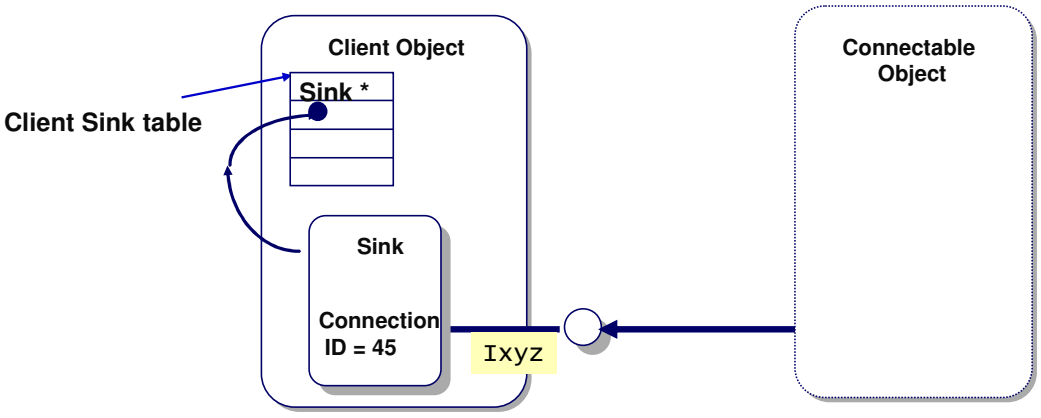
Mating the connection point with the client sink interface is done via `IConnectionPoint::Advise()` member. The client object calls this method, passing a pointer to the sink interface and a pointer to a `DWORD` as parameters. The `DWORD` will contain the connection identifier if `IConnectionPoint::Advise()` is successful.. The connection point could dangerously assume the sink pointer given to it by `IConnectionPoint::Advise()` was the correct interface pointer! Recall that the connection point represents an outgoing interface, i.e. it can only call methods on the interface it was designed to use. For this reason, when the connection point receives the sink interface pointer, it treats it like an `IUnknown` pointer and queries for the interface it knows how to use, e.g. `Ixyz`.

The connection identifier will be used by the client to disconnect the sink interface from the connection point with the `IConnectionPoint::Unadvise()` member. Remember that a given connection point may be connected to many sinks simultaneously.

Question: What is a Sink?

QA-IQ

- **Answer: a lightweight component object:**
  - **That implements the to-be-connected incoming interface on the client object e.g. Ixyz**
  - **That has no CLSID**
  - **That is created by the client object**
  - **Whose lifetime is nested in the client object's lifetime**



182

A client object that wants to use a connectable object will create a lightweight component object called a sink. A sink implements the interface a client wants a connectable object to use. The sink has no CLSID and is therefore something that is not manufactured by `CoGetClassObject()`. Rather, it is created as a simple C++ object by the client object.

Once a sink is connected, it attains a connection identifier. The connection identifier is used later in the disconnection process.

The Connection Identifier

QA-IQ

- A unique 'magic number' generated by the connection point in:
  IConnectionPoint::Advise(LPUNKNOWN pIUnknown, DWORD *pdwConnectID) {...}
- Given to the client by the connection point via:
  pCP->Advise(pIUnknown, &dwConnectID);
- Connection point stores the sink pointer and connection ID in a table entry
- Used later by the client to disconnect a connection point from one of its sinks:

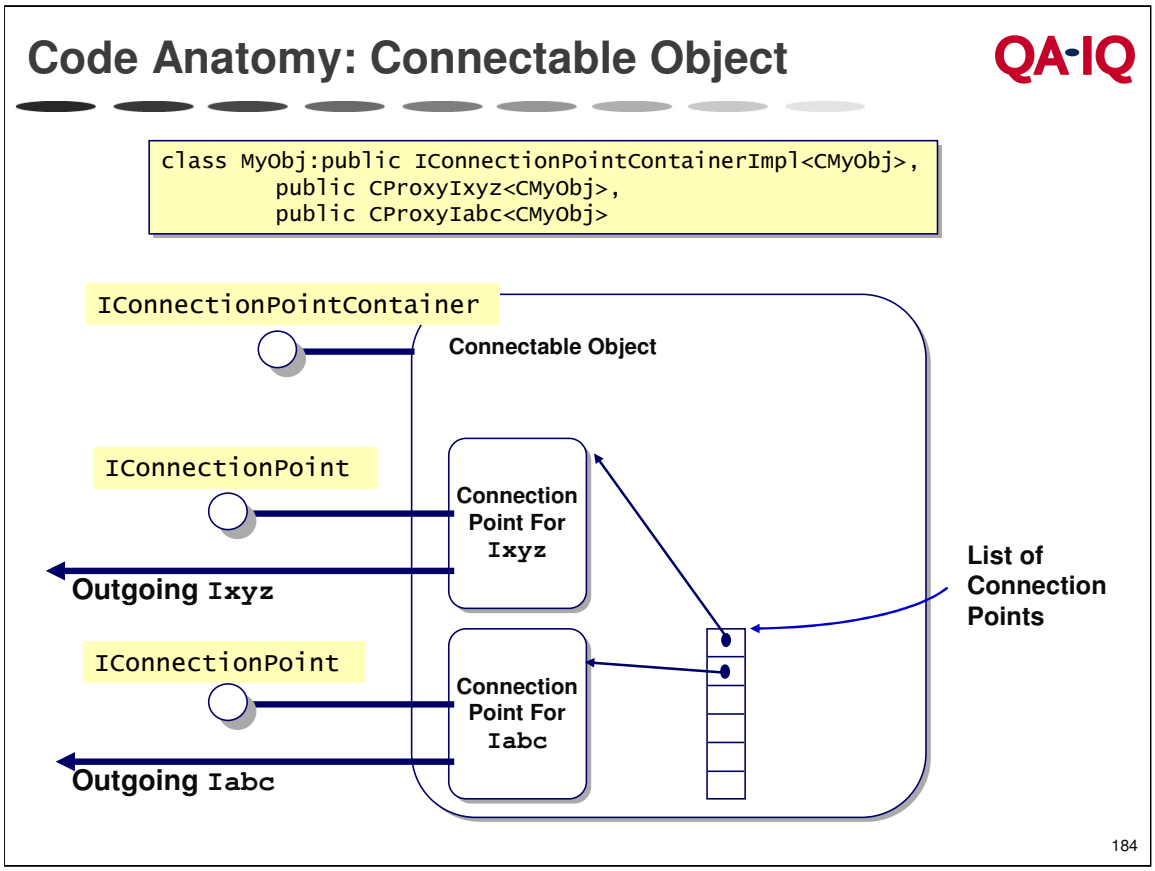
  Stop using my interface `Ixyz`

  IConnectionPoint::Unadvise(dwConnectID); — Identifies sink to disconnect
- Recall that a single connection point may have lots of outgoing connections to various clients
  - A connection ID identifies a given client to the connection point

183

The connection identifier is generated by the connection point when a client calls `IConnectionPoint::Advise()`. The second parameter to this call is a pointer to a `DWORD` that will be used to store the connection identifier assuming things go well. This identifier can then be used by the client in a call to `IConnectionPoint::Unadvise()` when it wants the connection point to `Release()` the interface pointer it is currently using on the client object (sink).

When generating connection identifiers, the onus is on the connection point to come up with a value that is not currently in use. One mechanism would be for the connection point to maintain a counter of `Advise()`'ed clients and increment it by one each time a new connection is established e.g. `m_Connect++`. The current connection counter value is then used as the connection identifier for the new client. Be careful though if not using the apartment threading model as this will not be thread-safe!



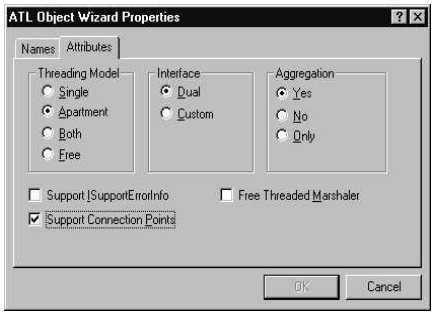
For each different outgoing interface, a connectable object will implement a separate connection point. For example, if the connectable object can support two outgoing interfaces: `Ixyz` and `Iabc`, there will be two connection points.

The connectable object maintains a (dynamic) list of connection points to facilitate queries from clients, such as `IConnectionPointContainer::FindConnectionPoint()` and `IConnectionPointContainer::EnumConnectionPoints()`.

Implementing Connection Points (1)

QA-IQ

- **Check Support Connection Points in ATL Object Wizard Properties**
  - **Defines a dispinterface in IDL for the outgoing interface and adds this to the coclass statement with a[default, source] attribute**
  - **Adds IConnectionPointContainerImpl<> to list of base classes**
  - **Adds IConnectionPointContainer to interface map**
  - **Adds an empty connection point map to the class**



185

With ATL 3, as provided with Visual Studio 6.0, you can specify support for connection points when using the Object Wizard. If you want to add support for connections points after using the wizard, you must make some changes manually.

First, an interface definition must be added to the IDL file. This is the same as adding a custom or dual interface to the IDL file. Having added an empty interface to IDL, add the interface to the `coclass` and mark it with the `[source]` attribute. This attribute marks an interface as being an outgoing interface; that is, one the coclass can call, not an interface that it implements.

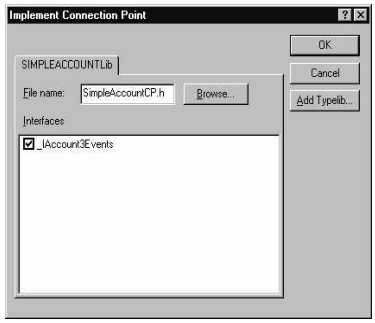
The class definition must also be amended. `IConnectionPointContainerImpl<>` must be in the list of classes that the object derives from. A `COM_INTERFACE_ENTRY_IMPL` must be added to the interface map for `IConnectionPointContainer`. A connection point map must also be added using the `BEGIN_CONNECTION_POINT_MAP()` and `END_CONNECTION_POINT_MAP()` macros.

The sink interface must be added to the connection point map using the `CONNECTION_POINT_ENTRY()`.

Implementing Connection Points (2)

QA-IQ

- **Use the Connection Point Wizard**
  - **Use ClassView to add methods to the event interface**
  - **Use FileView to compile the .idl file**
    - Produces type library and adds it to the project
  - **Return to ClassView and select Implement Connection Point…**
  - **Check event-interface box in dialog and click OK**
    - Produces a header file for an event proxy class
    - Adds interface to the connection point map



186

The Connection Point Wizard, which replaces the ATL Proxy Generator provided with Visual C++ 5.0, uses a type library to generate code to fire events on the sink interface held by the connection point object.

For more information, see *Adding Connection Points to an Object* in the Visual C++ 6.0 Documentation.

ATL Support Classes

QA-IQ

- **IConnectionPointContainerImpl<>**
  - **Implements IConnectionPointContainer**
  - **Maintains list of connection points**
    - See connection map
- **IConnectionPointImpl<>**
  - **Proxy class derives from it**
    - Object class derives from proxy
  - **Maintains connections to sinks**
    - Using CComDynamicUnkArray by default
    - Stores IUnknowns of sinks

187

The ATL template class `IConnectionPointContainerImpl<>` provides the implementation of `IConnectionPointContainer`. The class maintains a list of connection points in the connection map for the object.

`IConnectionPointImpl<>` implements `IConnectionPoint` It maintains a list of connected sinks. By default, this list is maintained in a dynamic array implemented by `CComDynamicUnkArray`. However, `CComUnkArray<>`, a static array, could also be used. The array stores the `IUnknown` pointers for the client sinks. Proxy classes generated by the ATL Proxy component will be derived from `CConnectionPointImpl<>`.

Other Stuff

QA-IQ

- A connectable object should never QueryInterface() a sink interface unless told to do so by the client
- A connection point reference count is included in the connectable object's reference count
  - Keeps the containing object and connection point in memory
- Clients can safely call IConnectionPointContainer::Release() regardless
- An outgoing interface could be IDispatch
  - Connection point would then call sink via Invoke()
- Connection points don't work well with DCOM
  - Security issues
  - Performance problems

188

The sink interfaces alongside `IUnknown` are the only interfaces a sink is expected to implement. This keeps the implementation of a client that has multiple sinks simpler. A connectable object should not therefore query a sink interface for anything other than `IUnknown` or the sink interface.

Connection point reference counts need to be included in the connectable object's reference count, so that a client can call `IConnectionPointContainer::Release()` without fear of the connectable object going away along with its connection points.

There is one other important issue to consider with connection points - they don't operate well with DCOM. This is primarily for two reasons.

The first is of these is security. In order for connection points to work, the connectable object needs to be able to invoke code within the client, which will involve either (a) dropping security on the client, or (b) enabling login and other privileges on each client machine for the server. Security also rears its ugly head because rogue software which gains access to the connectable object can then enumerate the clients.

The second problem is one of performance. The number of round trips involved in the establishing of communications is not conducive to great network operation.

It is for both of these reasons that many developers provide their own bi-directional communication mechanism for use in DCOM.

Summary

QA-IQ

- A connectable object has at least one outgoing interface
- A connectable object implements IConnectionPointContainer
- A connectable object has one connection point per outgoing interface
- A connection point implements IConnectionPoint and one outgoing interface
- The client of a connectable object will create at least one sink
- A sink implements the incoming interface that the connection point will use
- ATL provides support for connection points

189