ATL - Attributes Tutorial on Visual C++ .Net

Program examples compiled using Visual Studio/C++ .Net 2003 compiler on Windows XP Pro machine with Service Pack 2. Topics and sub topics for this tutorial are listed below. Don't forget to read Tenouk's small disclaimer. The supplementary note for this tutorial is .NET.

Using Visual C++ and attributes, you can speed up and simplify the process of COM programming. This tutorial uses attributes to implement both a client and a server application. During the course of this tutorial, you will use **attributes** and **events**.
The tutorial develops a singleton server object (an object that can have only one instance) that has its own dual interface and a dispatch interface used for firing off events. The server object takes data, given to it through the `Send()` method of its dual interface, and transmits it to all connected components through the `Transfer()` event on its dispatch interface.
In addition, the tutorial implements a client (an ActiveX control) that contains a server object. The control responds to the `Transfer()` event fired by the server object and has its own dual interface that implements several methods: `Connect()`, `Send()`, and `Disconnect()`. If the `Transfer()` event is fired with a variant containing a `BSTR`, the string is displayed in the center of the control.
The tutorial is divided into seven steps, each building on the application developed in the previous step. Keep in mind that all the coding part was done manually, copy and paste, instead of using `'wizard'`.

- Step 1: Creating the Projects.
- Step 2: Adding the Server Object.
- Step 3: Implementing the Server.
- Step 4: Adding the Client Object.
- Step 5: Adding the Client Interfaces.
- Step 6: Implementing the Client.
- Step 7: Using the Client Control.

Step 1: Creating the Projects

In this step, you will create an initial solution containing two ATL projects. The two projects will implement the server and client objects of the tutorial.

To create a new solution

1. In the Visual Studio environment, click **New** from the **File** menu and then click **Blank Solution**.

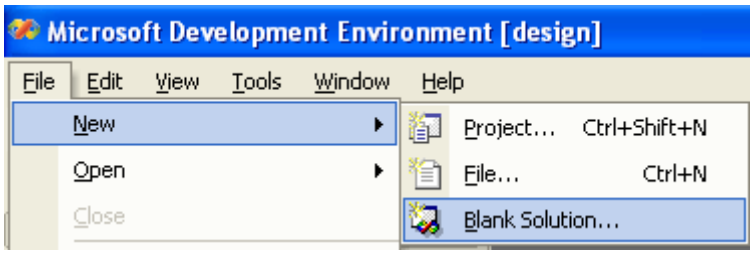


Figure 1: Creating a new Visual C++ .Net project, starting with blank solution.

2. In the **Name** box, type **DispSink**.

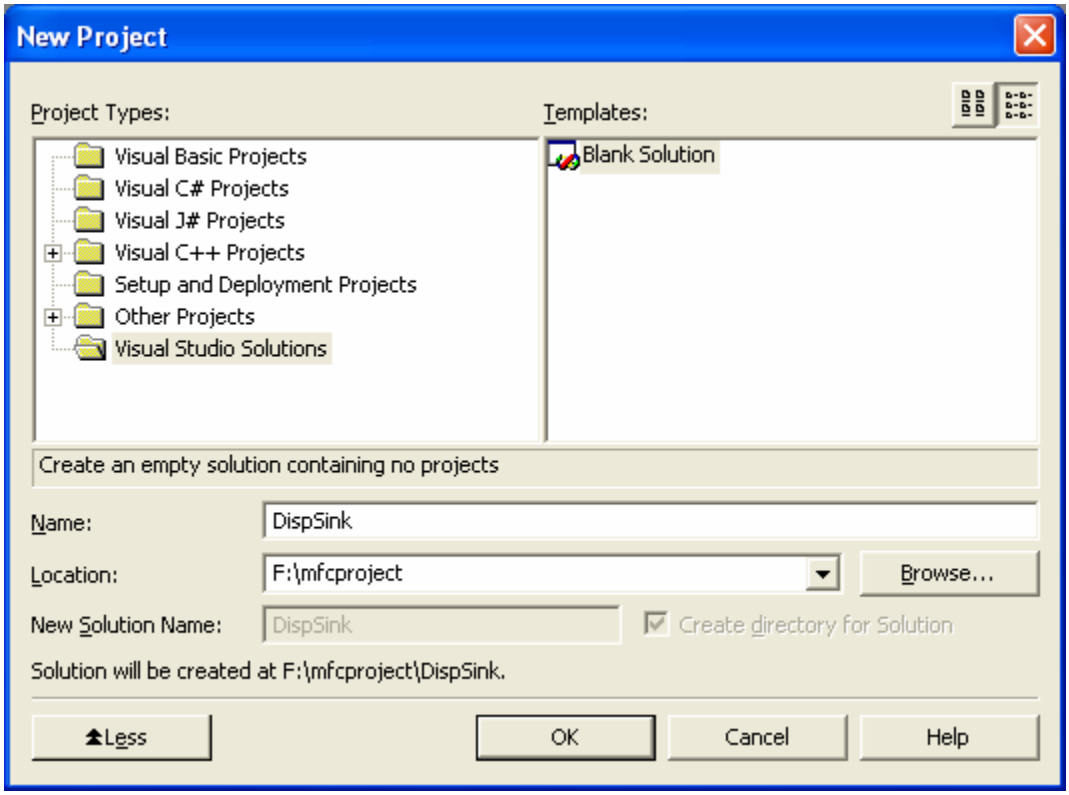


Figure 2: Entering the blank solution name.

3. Click **OK** to create a blank solution.

Once the solution has been created, add the two ATL projects to the empty solution.

To create a new project

1. In **Solution Explorer**, right-click the **DispSink** solution node.

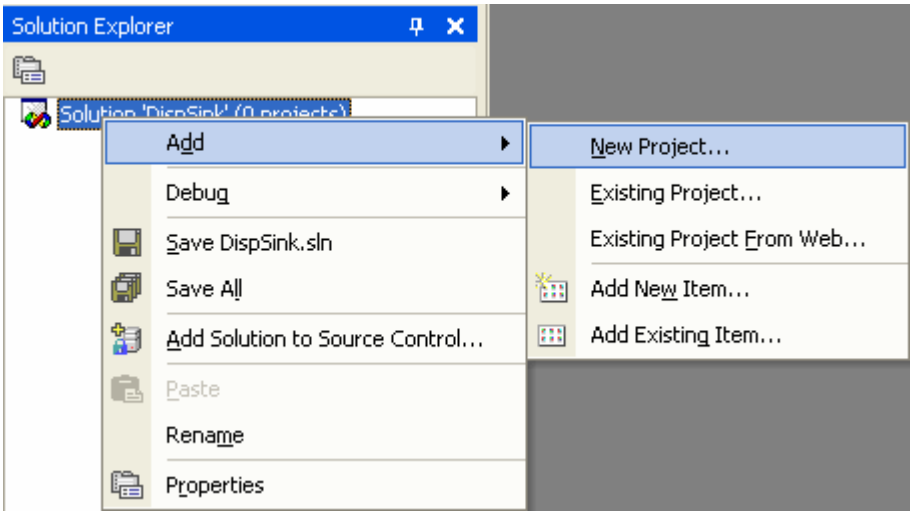


Figure 3: Adding new project to Visual C++ .Net solution.

2. On the shortcut menu, click **Add** and then click **New Project**. The **New Project** dialog box appears.
3. From the **Visual C++ Projects** folder, select **ATL Project**.

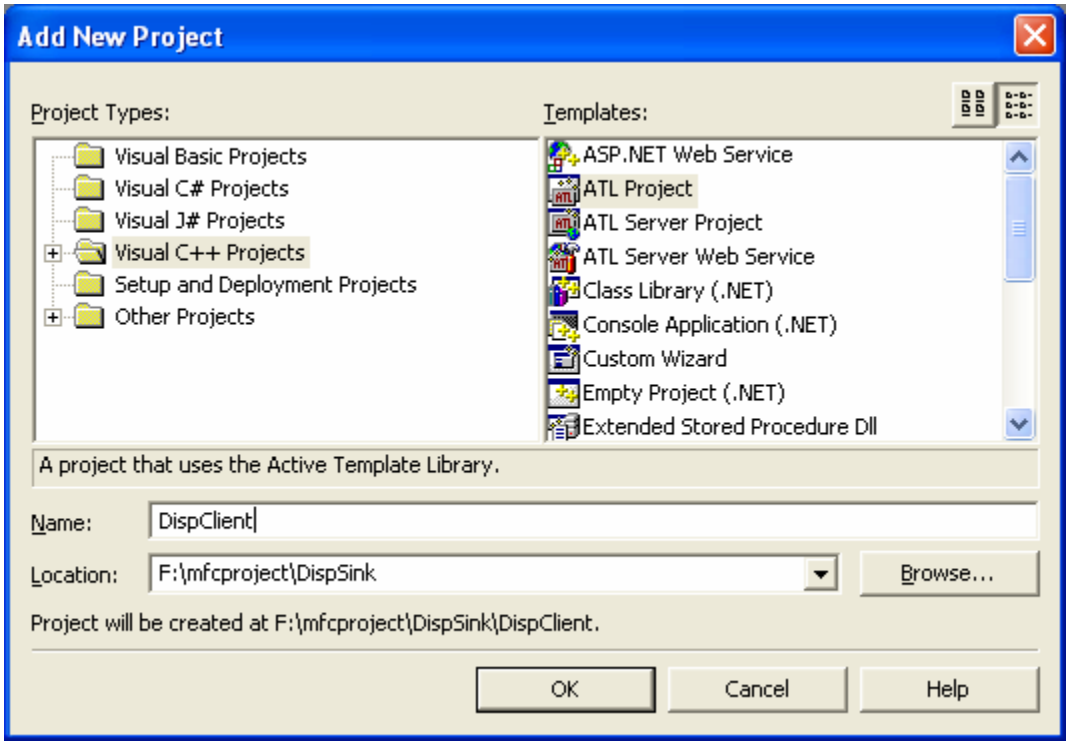


Figure 4: Adding **DispClient**, a new ATL project to solution.

4. In the **Name** box, type **DispClient**.
5. Click **OK** to start the **ATL Project Wizard**. The ATL Project Wizard offers several choices to configure the initial project. Because the wizard creates an attributed project by default, you do not have to change any wizard settings.

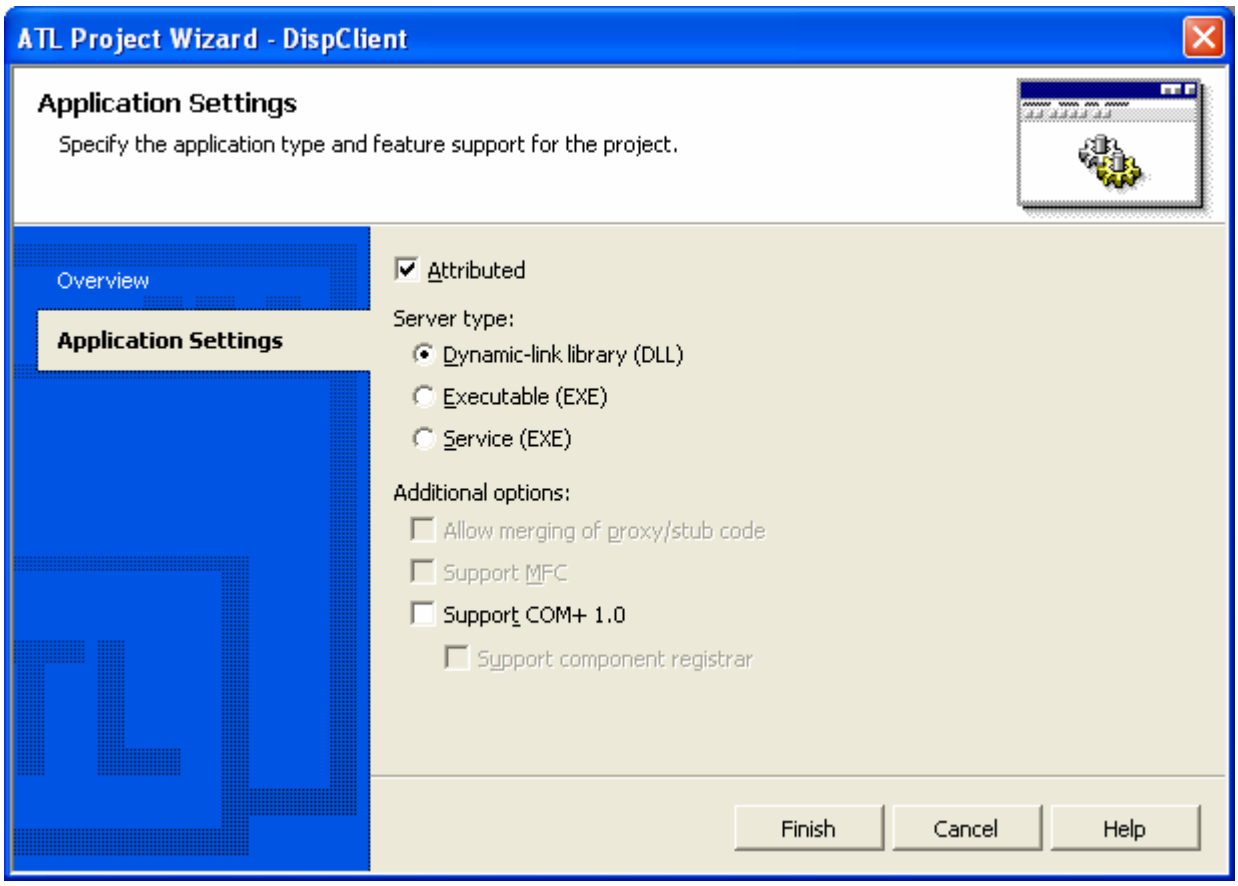


Figure 5: ATL Project Wizard Application Settings page.

6. Click **Finish** to generate the **DispClient** project. The files generated by the wizard are listed in the following table.

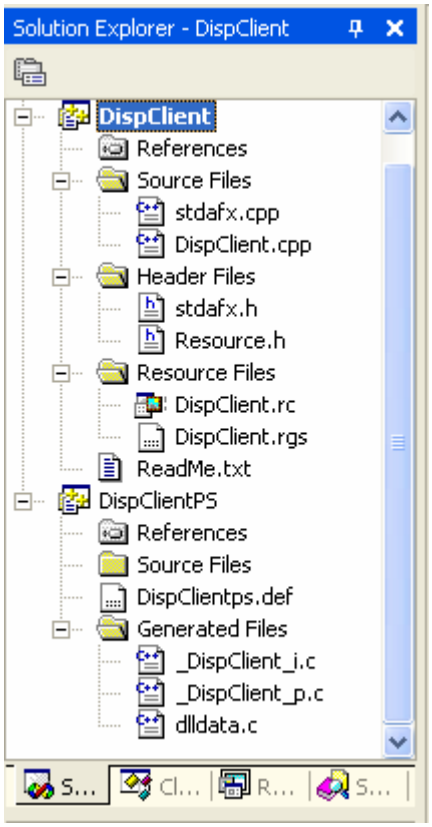


Figure 6: The generated files and resources for **DispClient** ATL project.

File	Description
DispClient.cpp	Contains the module attribute, which implements `DLLMain()`, `DLLRegisterServer()`, and `DLLUnregisterServer()`. The module type also defines the `GUID` for the type library. Notice that the `GUID` and helpstring have been automatically generated.
DispClient.h	A MIDL-generated file that will contain interface definitions. For purposes of this tutorial, this file will be unnecessary.
DispClient.rc	The resource file, which initially contains the version information and a string containing the project name.
DispClient.rgs	Contains entries that are added to the registry, which will register your COM object.
DispClient.vcproj	A file containing the project settings.
DispClientps.def	The module definition file for the proxy/stub DLL. For purposes of this tutorial, this is unnecessary.
ReadMe.txt	A file containing an explanation of the files generated by the application wizard.
Resource.h	The header file for the resource file.
StdAfx.cpp	The file that will `#include` the ATL implementation files.
StdAfx.h	The file that will `#include` the ATL header files.

Table 1.

You will also notice a **DispClientPS** project. This project creates the **proxy** and **stub** that allow your object to be accessed from outside of its COM apartment.

To create a new server project

1. In **Solution Explorer**, right-click the **DispSink** solution node.
2. On the shortcut menu, click **Add**, and then click **New Project**. The **New Project** dialog box appears.

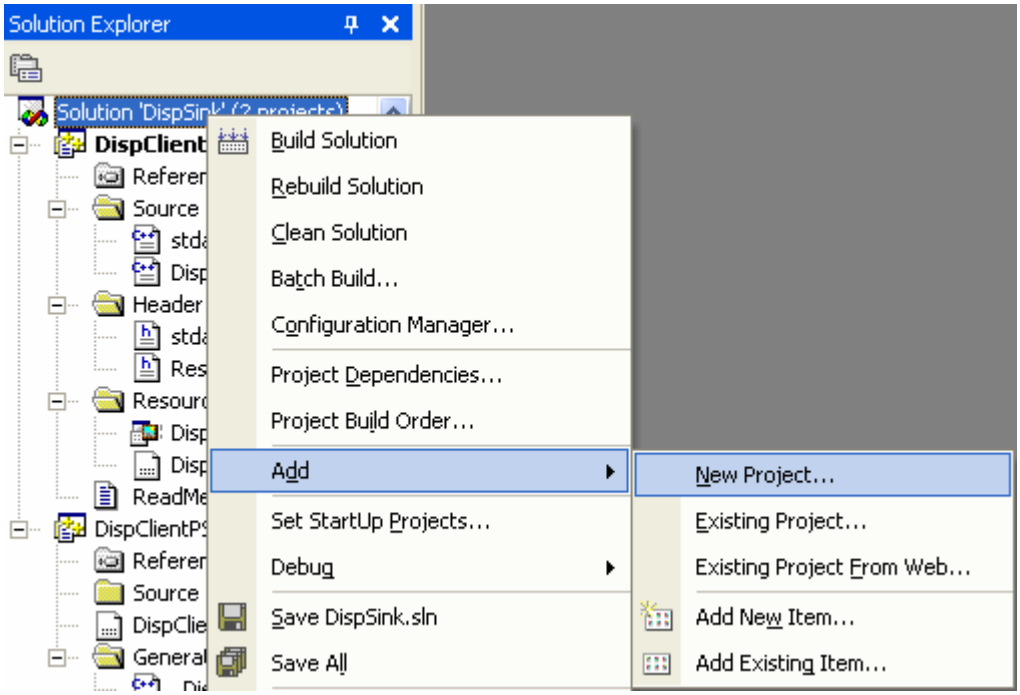


Figure 7: Adding another new project to solution.

3. From the Visual C++ Projects folder, select **ATL Project**.
4. In the **Name** box, type **DispServer**.

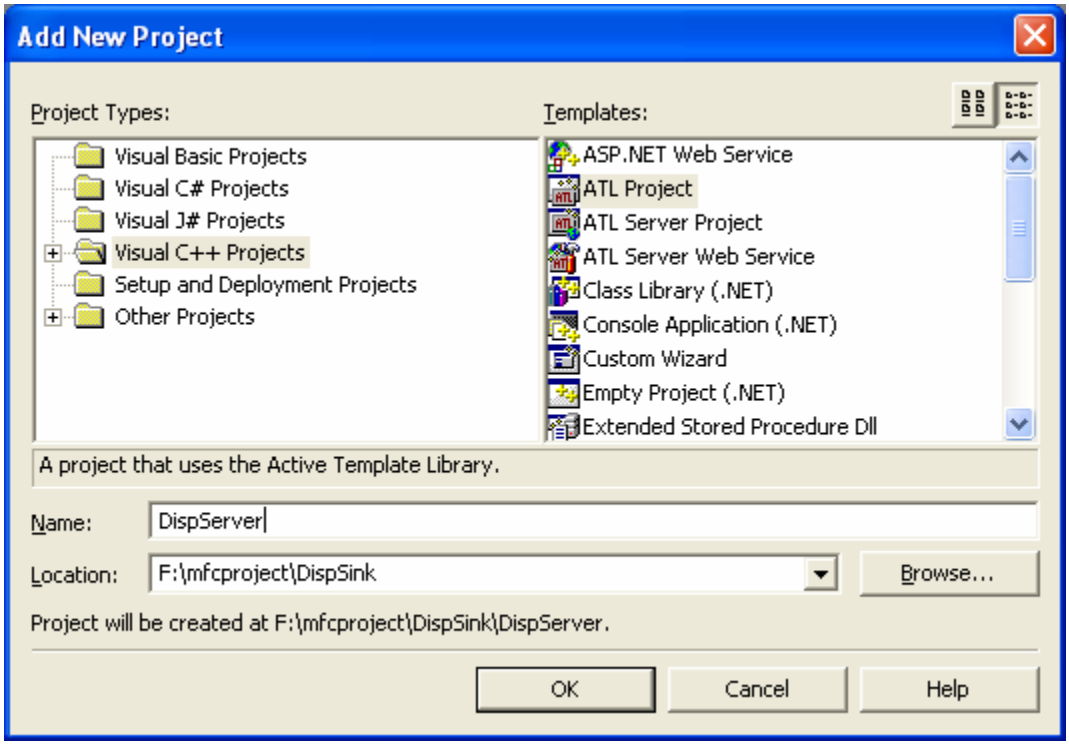


Figure 8: Adding new **DispServer**, an ATL project to solution.

5. Click **OK** to start the ATL Project Wizard. The ATL Project Wizard appears.
6. Click the **Application Settings** tab to display the current options for the initial project.

7. Select the **Executable** server type.

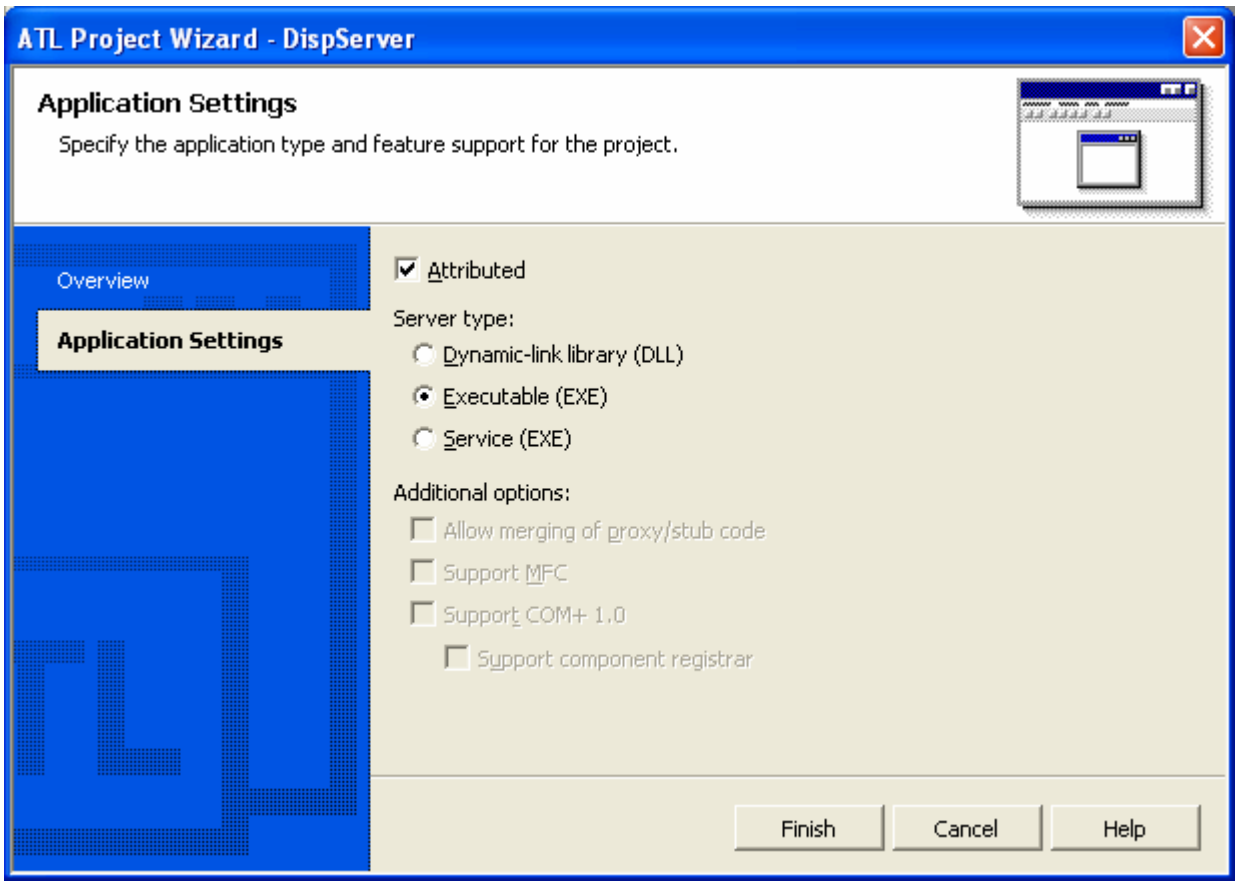


Figure 9: Modifying **Application Settings** options.

8. Click **Finish** to generate the **DispServer** project.

For the **DispServer** project, the application wizard creates a similar set of files compared to the **DispClient** project. The only difference is in the file **DispServer.cpp**, where the module type is **exe** instead of **dll**.
The next step focuses on the implementation of the server object.

Step 2: Adding the Server Object

In this step, you will use Class View to add objects to the project. You need to add a single ATL object (named **CDispServ**) to the server. This object also serves as an event source.

To add a class to the project

1. In Class View, right-click the **DispServer** project. On the shortcut menu, click **Add** and then click **Add Class**. The **Add Class** dialog box appears.

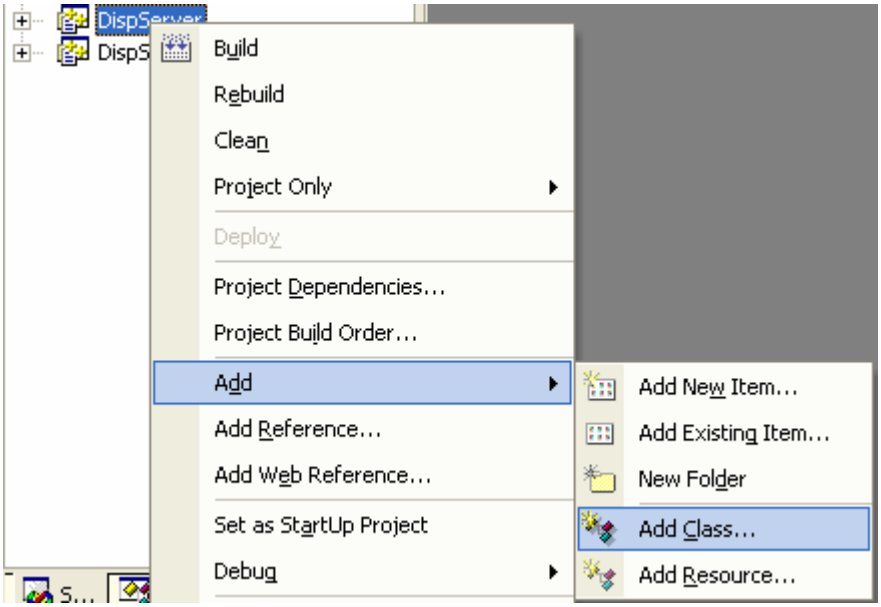


Figure 10: Adding new class to **DispServer**.

2. Select **ATL Simple Object** and click **Open**. The **ATL Simple Object Wizard** appears.

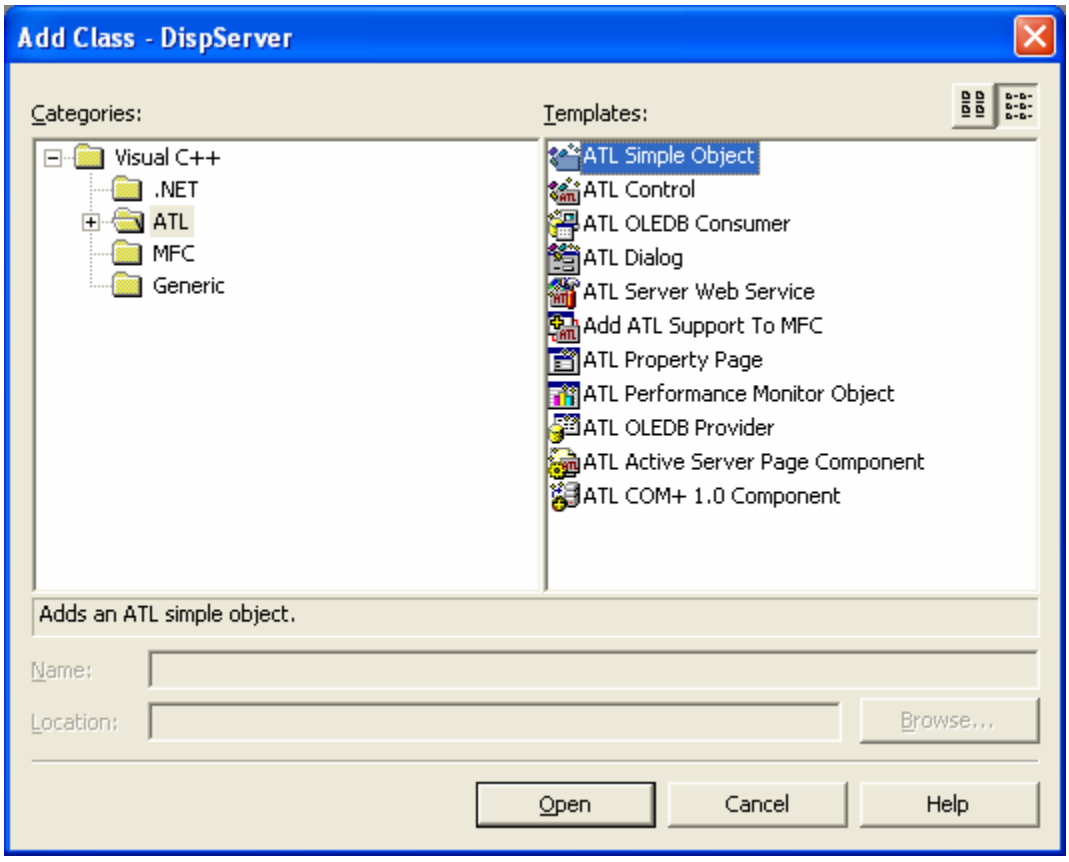


Figure 11: Adding an **ATL Simple Object** template to DispServer.

3. In the **Short name** field, type **DispServ**. The remaining fields are automatically completed. The additional fields contain information on the name of the class as well as the names of the files that should be created. The **Type** field is a description of the object, while the **ProgID** field is the readable name that can be used to look

up the `CLSID` of the object. Note that the **Attributed** box is selected and unavailable. An ATL object created by the wizard in an attributed project is always attributed.

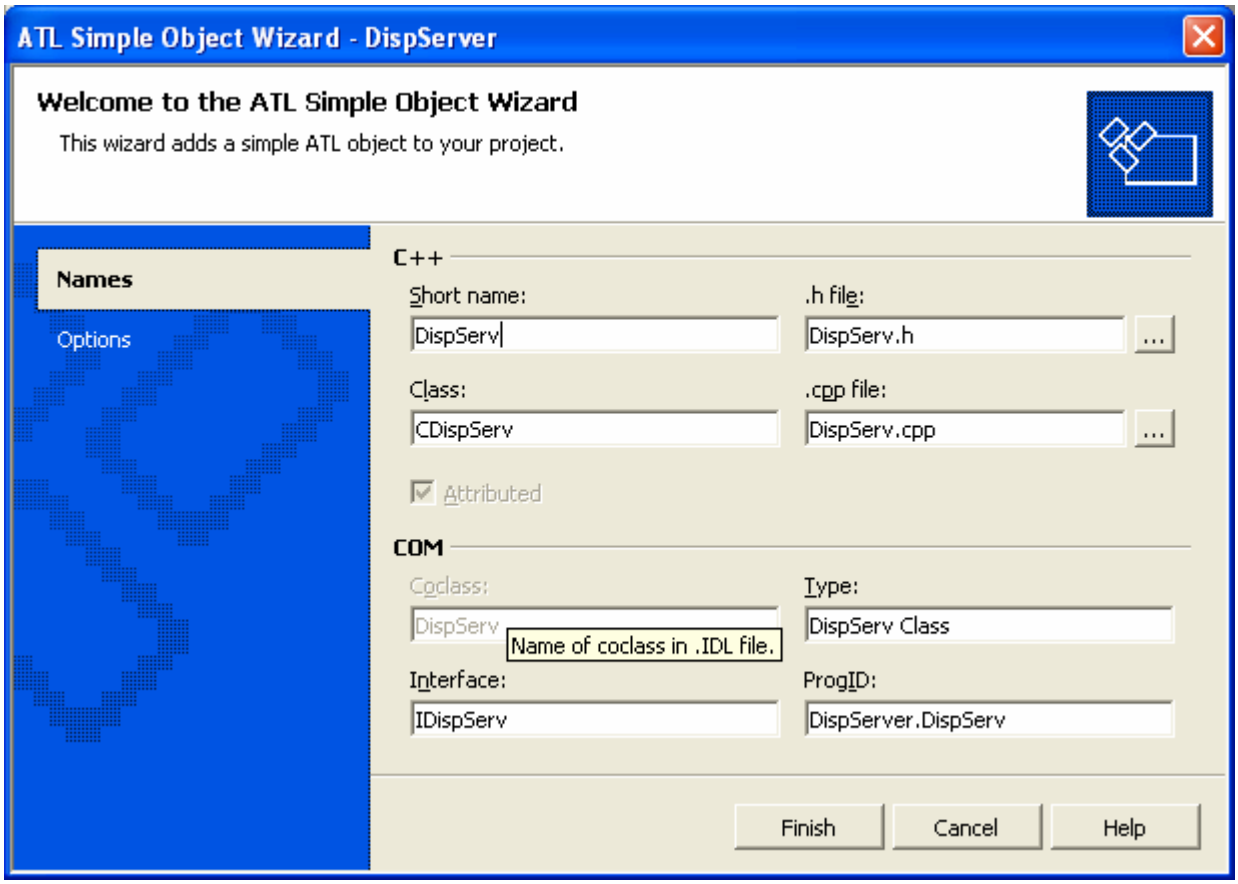


Figure 12: An ATL Simple Object Wizard, **Names** page.

4. Click the **Options** tab in the wizard.
5. On the **Options** tab, select **Connection points support**. This allows the object to act as an event source.

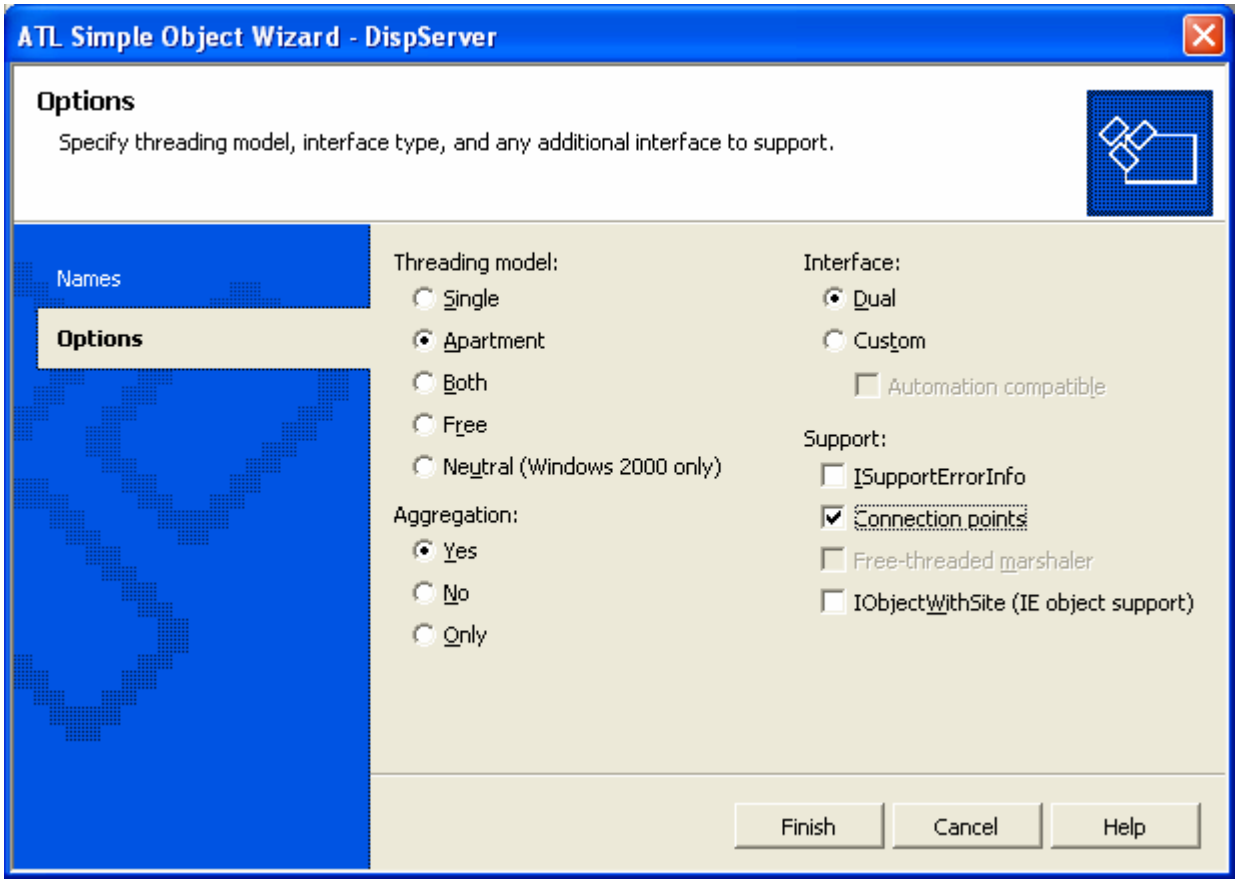


Figure 13: ATL Simple Object Wizard, **Options** page.

6. Click **Finish** to generate the `DispServ` class.

Note in Class View that the `CDispServ` class, as well as the `IDispServ` and `_IDispServEvents` interfaces, have been created and are now visible. To implement the new class, the wizard added two new files to the project:

- **DispServ.h** contains most of the implementation of the `CDispServ` class, as well as the interfaces. The `CDispServ` class has automatically been made a COM event source through the `event_source` attribute with the `_IDispServEvents` interface automatically specified as the event interface for `CDispServ`. **UUIDs**, **progids**, **help strings**, and **version numbers** have been automatically generated for the class and the interfaces.
- **DispServ.cpp** contains the remainder of the `CDispServ` class. At this point, it includes a few necessary files.

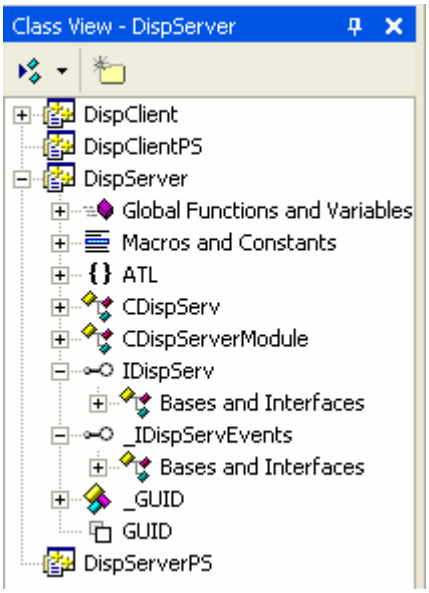


Figure 14: The added interfaces and events (of course respective files also added)

You can build the application by clicking **Build DispServer** from the **Build** menu, though **DispServer** does not actually do anything interesting yet. However, it does have the capability to register itself. This is done automatically when the project is built. The next step implements the functionality of the **DispServer** object.

Step 3: Implementing the Server

In this step, you will add the functionality to make the class do something interesting. In the last two steps, you have created the `CDispServ` class, which now exposes a custom interface (`IDispServ`), and an event source (the `_IDispServEvents` event interface). Beyond this implementation, however, it does not actually do anything. The main purpose of the `CDispServ` class is to receive data using the `Send()` method and transmit the information using the `Transfer()` event. Therefore, a **DispClient** object connected to a **DispServer** object can call `Send()` through the `IDispServ` interface, passing it data. The `Send()` method then fires the `Transfer()` event, propagating the data to all connected **DispClient** objects. Previously, this required creating a connection point proxy class to fire the events. With the new events feature, the situation is simplified; you will add the `Transfer()` event to the `_IDispServEvents` interface.

To add and define the Transfer event

1. In Class View, click the **DispServer** project to expand the node.
2. Double-click the `_IDispServEvents` interface. This opens the interface definition in the **Code editor**.
3. Define the `Transfer()` event for the `_IDispServEvents` interface by adding the following code to the `_IDispServEvents` interface definition:

   ```
   [id(1), helpstring("method Transfer")] HRESULT Transfer(VARIANT data);
   ```

4. Make `IDispatch` a base class of the interface.

Your interface definition should now match the following definition:

```
__interface _IDispServEvents : public IDispatch
{
   [id(1), helpstring("method Transfer")] HRESULT Transfer(VARIANT data);
};
```

```
]
__interface _IDispServEvents : public IDispatch
{
    [id(1), helpstring("method Transfer")] HRESULT Transfer(VARIANT data);
};
```

Listing 1.

To fire the `Transfer()` event, make a call to `_IDispServEvents_Transfer()` from your event source. The form for firing an event is always `InterfaceName_EventName`.

To add and define the Send method

1. In Class View, double-click the `IDispServ` interface. This opens the interface definition in the Code editor.
2. Define the `Send()` method for the `IDispServ` interface by adding the following code to the `IDispServ` interface definition:

```
[id(1), helpstring("method Send")] HRESULT Send(VARIANT data);
```

Your interface definition should now match the following definition:

```
__interface IDispServ : IDispatch
{
   [id(1), helpstring("method Send")] HRESULT Send(VARIANT data);
};
```

```
]
__interface IDispServ : IDispatch
{
    [id(1), helpstring("method Send")] HRESULT Send(VARIANT data);
};
```

Listing 2.

To implement the `Send()` method for the `CDispServ` class, scroll down to the bottom of **DispServ.h** and add the following code directly below the last `public:` section of the class:

```
STDMETHOD(Send)(VARIANT data)
{
   _IDispServEvents_Transfer(data);
   return (S_OK);
}
```

```
public:
    STDMETHOD(Send)(VARIANT data)
    {
        _IDispServEvents_Transfer(data);
        return (S_OK);
    }
};
```

Listing 3.

The full `CDispServ` class should now look like this:

```
class ATL_NO_VTABLE CDispServ : public IDispServ
{
public:
   CDispServ()
   {
   }

   __event __interface _IDispServEvents;

   DECLARE_PROTECT_FINAL_CONSTRUCT()

   HRESULT FinalConstruct()
   {
      return S_OK;
   }

   void FinalRelease()
   {
   }

public:
   STDMETHOD(Send)(VARIANT data)
   {
      _IDispServEvents_Transfer(data);
      return S_OK;
   }
};
```

The final task implements the `CDispServ` class as a singleton server, which means that all clients will connect to the same server.

To define `CDispServ` as a singleton server

1. In Class View, double-click the **CDispServ** node.
2. Add the following lines directly below the `DECLARE_PROTECT_FINAL_CONSTRUCT()` in class `CDispServ`:

```
DECLARE_CLASSFACTORY_SINGLETON(CDispServ);
```

```
__event __interface _IDispServEvents;

DECLARE_PROTECT_FINAL_CONSTRUCT()

DECLARE_CLASSFACTORY_SINGLETON(CDispServ);

HRESULT FinalConstruct()
```

Listing 4.

The server is now complete. You can build the server by selecting **Build DispServer** from the **Build** menu. Once the server is successfully built, it will register itself.
The Visual Studio development environment also provides wizards that let you add properties and methods. Just right click on the interface node in Class View and select a wizard from the context menu. The next step adds a simple client object to the **DispClient** project.

Step 4: Adding the Client Object

In this step, you will add a client object (named `CDispCtl`) to the client project. This client object will be implemented by a simple, lightweight control added by the **Add Class** dialog box.

Adding the client object to the project

1. In Class View, right-click the **DispClient** project.
2. On the shortcut menu, click **Add** and then click **Add Class**. The **Add Class** dialog box appears.

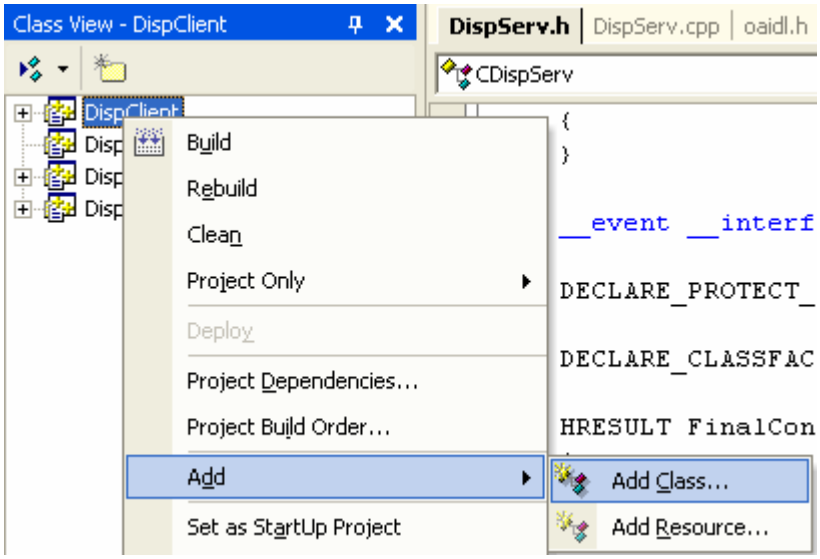


Figure 15: Adding a new class to **DispClient**.

3. From the **ATL** folder, double-click **ATL Control**. The **ATL Control Wizard** appears.
4. In the **Short name** field, type **DispCtl**. The remaining fields are automatically completed.

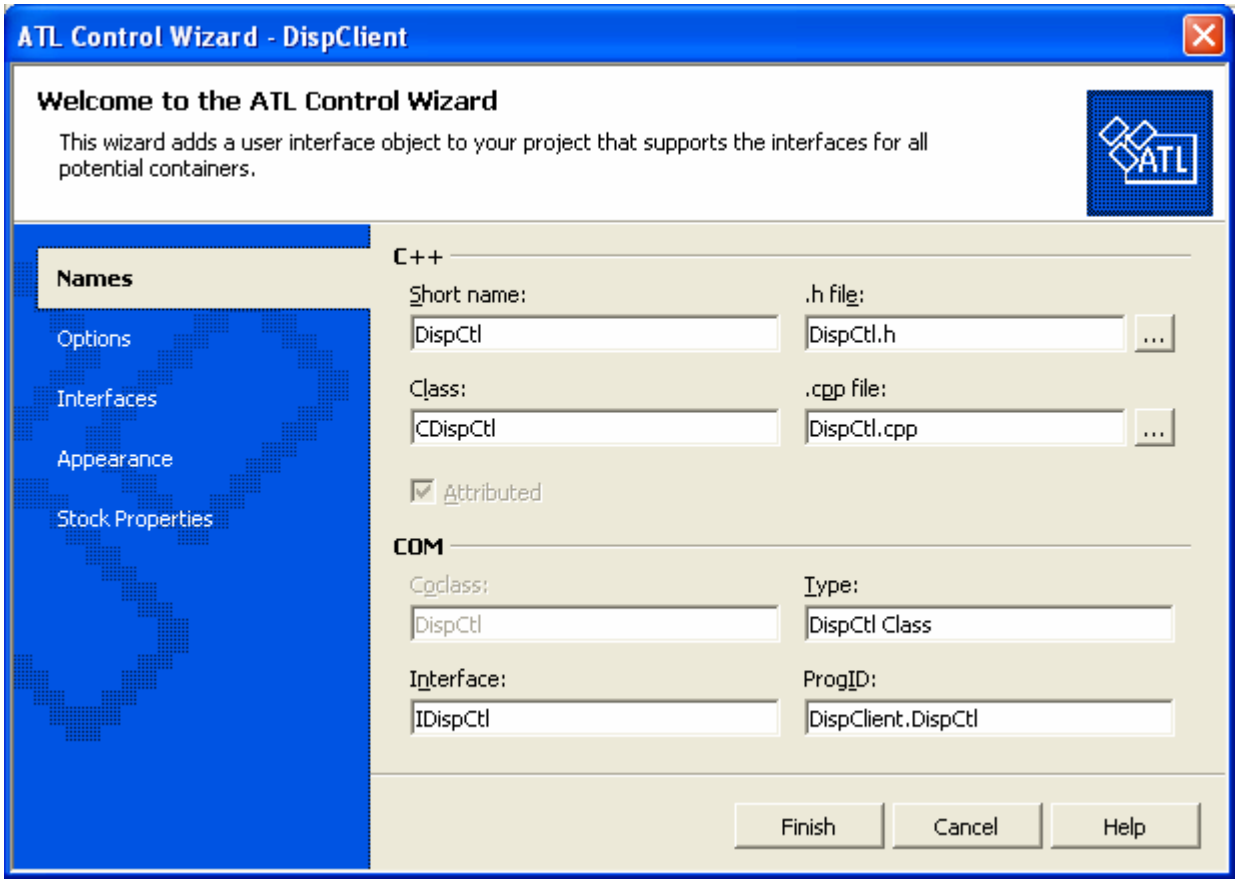


Figure 16: ATL Control Wizard, **Names** page.

5. Click the **Options** tab.
6. Select **Minimal control**.
7. Change the **Threading Model** to **Single**.

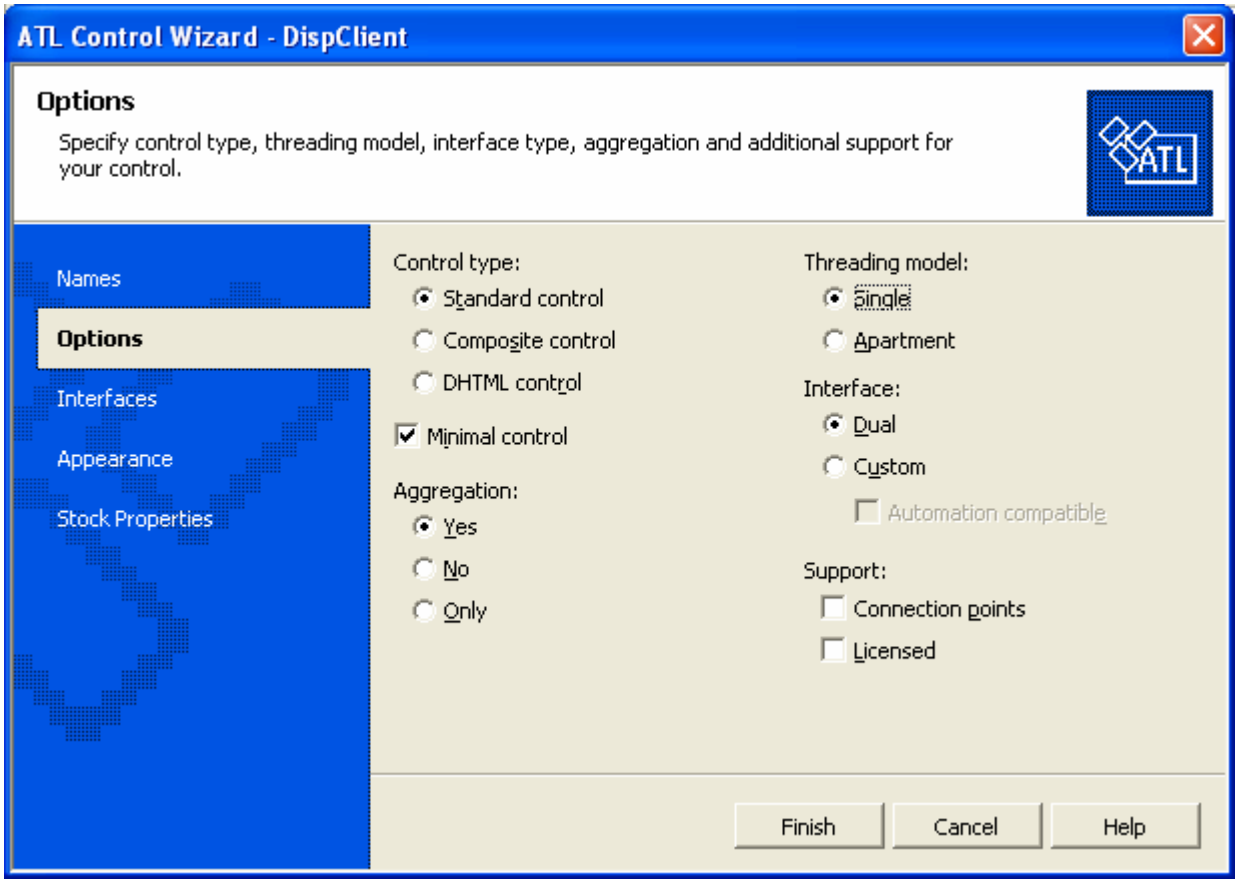


Figure 17: ATL Control Wizard, **Options** page.

8. Click the **Interfaces** tab.
9. Move **`IDataObject`** and **`IProvideClassInfo2`** to the **Supported** column.

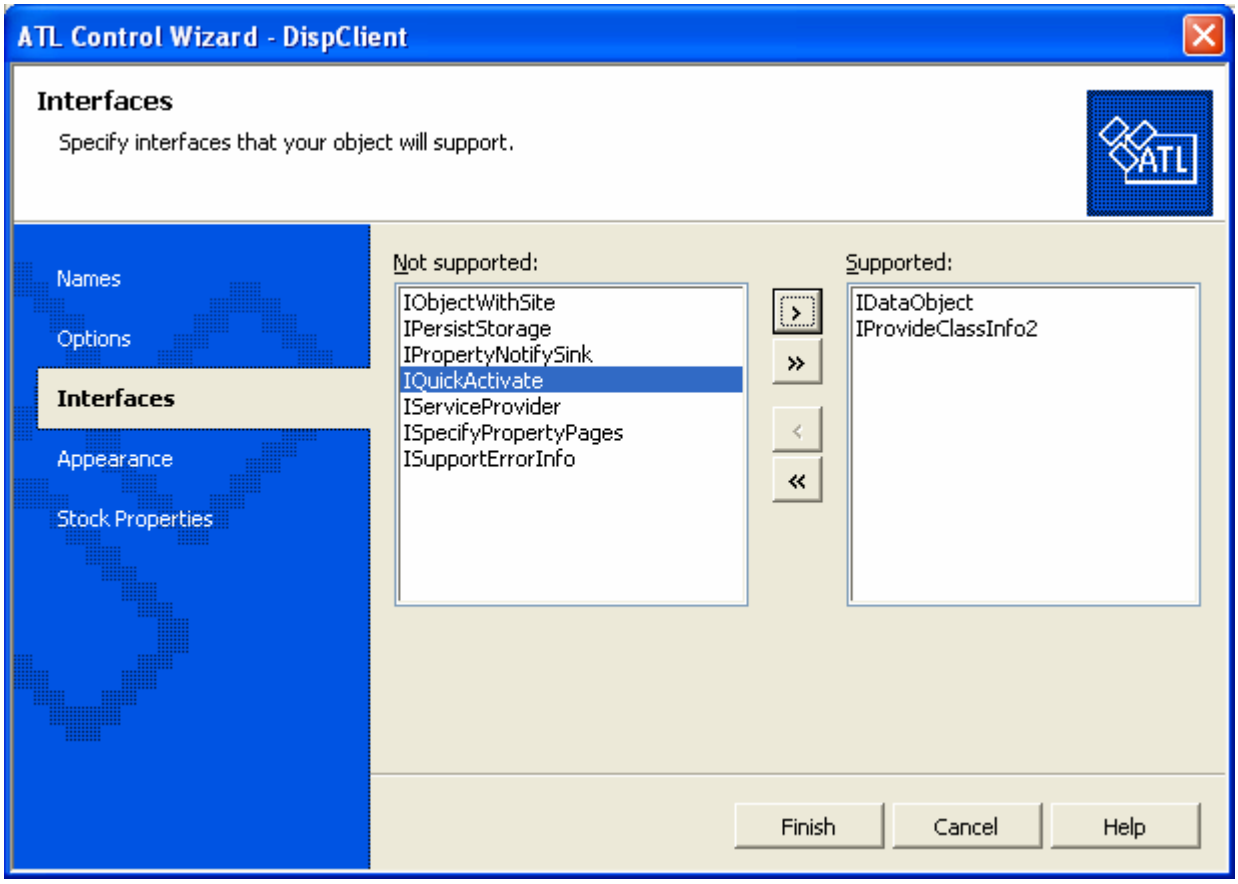


Figure 18: ATL Control Wizard, **Interfaces** page.

10. Click **Finish** to generate the `CDispCtl` class.

`CDispCtl` inherits from all the ATL template classes required for a light control. Additionally, the class inherits from `IDataObjectImp`, providing methods to support **Uniform Data Transfer**, and `IProvideClassInfo2Impl`, allowing the access of type information for the object's coclass from other COM objects.
Registration of the control is achieved using the **registration_script** attribute. The parameter on this attribute is the name of the registration script, in this case **Control.rgs**, which is the default registration script for controls.
Also included is a default implementation of the `OnDraw()` method.
The final step is to add the **event_receiver** attribute to the new class.

Adding an attribute to an existing class

1. In Class View, double-click the **DispClient** node and click the `CDispCtl` class. This will display the class definition for `CDispCtl`.
2. Just above the class definition is the attribute block for the class. Add `event_receiver(com)` to the attribute block. Don't forget to put a comma at the end of the previous line of code.

```
// CDispCtl
[
    coclass,
    threading("single"),
    vi_progid("DispClient.DispCtl"),
    progid("DispClient.DispCtl.1"),
    version(1.0),
    uuid("E6D81F5E-2252-4A53-A006-C0BB1A11512F"),
    helpstring("DispCtl Class"),
    registration_script("control.rgs"),
    event_receiver(com)
]
```

Listing 5.

The resulting attribute list should resemble the following:

```
[
   coclass,
   threading("single"),
   progid("DispClient.DispCtl"),
   version(1.0),
   uuid("E26DD5C9-DE30-46FF-B6B6-51F31840B437"),
   registration_script("control.rgs"),
   event_receiver(com)
]
```

In the next step, you will add the needed interfaces to the control object.

Step 5: Adding the Client Interfaces

In this step, you will implement the various methods of the `IDispCtl` interface. To allow the client object to handle events fired from the `_IDispServEvents` interface you need to allow access to the interface exposed by the `CDispServ` class.

To allow the client object to handle events

1. Open **DispCtl.h** in your source editor.
2. Add the following line below the `#include <atlctl.h>` line:

```
#import "progid:DispServer.DispServ.1" embedded_idl, no_namespace
```

```
// DispCtl.h : Declaration of the CDispCtl
#pragma once
#include "resource.h"       // main symbols
#include <atlctl.h>
#import "progid:DispServer.DispServ.1" embedded_idl, no_namespace
```

Listing 6.

This assumes that your class name is `DispServer`.

In the last step, we added a simple client object (`CDispCtl`) with an empty interface to the project. At this point, you need to add the following methods to the `IDispCtl` interface:

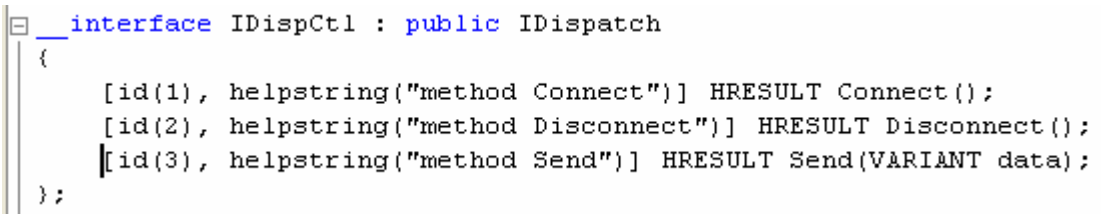
- `Connect()` - Causes the client to hook to the server, enabling it to receive events.

- `Disconnect()` - Unhooks the event source of the client.
- `Send()` - Sends the specified data to the server.

To add the methods to your interface

1. In Class View, double-click the **IDispCtl node** under the **DispClient** project.
2. In the **Source editor**, add the following lines to the `IDispCtl` interface:

```
[id(1), helpstring("method Connect")] HRESULT Connect();
[id(2), helpstring("method Disconnect")] HRESULT Disconnect();
[id(3), helpstring("method Send")] HRESULT Send(VARIANT data);
```



Listing 7.

The interface should now resemble the following code:

```
[
   object,
   uuid(...),
   dual,
   helpstring("IDispCtl Interface"),
   pointer_default(unique)
]
__interface IDispCtl : IDispatch
{
   [id(1), helpstring("method Connect")] HRESULT Connect();
   [id(2), helpstring("method Disconnect")] HRESULT Disconnect();
   [id(3), helpstring("method Send")] HRESULT Send(VARIANT data);
};
```

In the next step, you will add the implementation of the `IDispCtl` methods and modify other sections of code.

Step 6: Implementing the Client

In this step, you will implement `IDispCtl`'s methods in `CDispCtl`, add an event handler, and modify the `OnDraw()` function.

Implementing the IDispCtl Interface

`CDispCtl` is where you will implement methods declared in `IDispCtl`. The implementation begins with adding the following three data members, used by the new methods:

- `m_bConnected(bool)` - Indicates the connect state of the server.
- `m_pIServ(_IDispServEvents*)` - A pointer to the `IDispServ` interface the client will connect to.
- `m_OutText(variant)` - Holds the data received from the server.

To add the data members

1. In Class View, double-click the **IDispCtl** node under the **DispClient** project.

2. In the **Source editor**, add the following lines at the end of the `CDispCtl` class, in a `private`: section:

```
private:
   // Data
   bool m_bConnected;
   CComPtr<IDispServ> m_spIServ;
   CComVariant m_OutText;
```

```
    void FinalRelease()
    {
    }
private:
   // Data
   bool m_bConnected;
   CComPtr<IDispServ> m_spIServ;
   CComVariant m_OutText;
};
```

Listing 8.

3. To initialize the new data members, modify the default constructor to match the following:

```
CDispCtl()
{
   m_bConnected = false;
   m_OutText = L"Not connected";
}
```

```
public:

    CDispCtl()
    {
        m_bConnected = false;
        m_OutText = L"Not connected";
    }
```

Listing 9.

4. To ensure that you disconnect from the server upon exiting, add a destructor for the `CDispCtl` class. Add the following directly below the default constructor declaration:

```
~CDispCtl()
{
   Disconnect();
}
```

```
        m_bConnected = false;
        m_OutText = L"Not connected";
    }

    ~CDispCtl()
    {
        Disconnect();
    }
```

Listing 10.

The first method you will implement is the `Connect()` method. This method creates an instance of `CDispServ` using `CoCreateInstance()` and connects the `Transfer()` event from the newly created instance of `CDispServ` to the event handler method, `OnTransmit()` (not yet implemented). The connection is achieved by the `__hook` keyword.

To implement the `Connect()` method

Below the data members created earlier, add the following code:

```
HRESULT Connect()
{
   HRESULT hr;
   hr = m_spIServ.CoCreateInstance(__uuidof(CDispServ));
   if (SUCCEEDED(hr))
   {
      hr = __hook(&_IDispServEvents::Transfer, m_spIServ,
      CDispCtl::OnTransmit);
   }
   if (hr == S_OK)
   {
      m_bConnected = true;
      m_OutText = L"Connected";
   }
   FireViewChange();
   return(hr);
}
```

```
CComVariant m_OutText;

HRESULT Connect()
{
    HRESULT hr;
    hr = m_spIServ.CoCreateInstance(__uuidof(CDispServ));
    if (SUCCEEDED(hr))
    {
        hr = __hook(&_IDispServEvents::Transfer, m_spIServ,
        CDispCtl::OnTransmit);
    }
    if (hr == S_OK)
    {
        m_bConnected = true;
        m_OutText = L"Connected";
    }
    FireViewChange();
    return(hr);
}
```

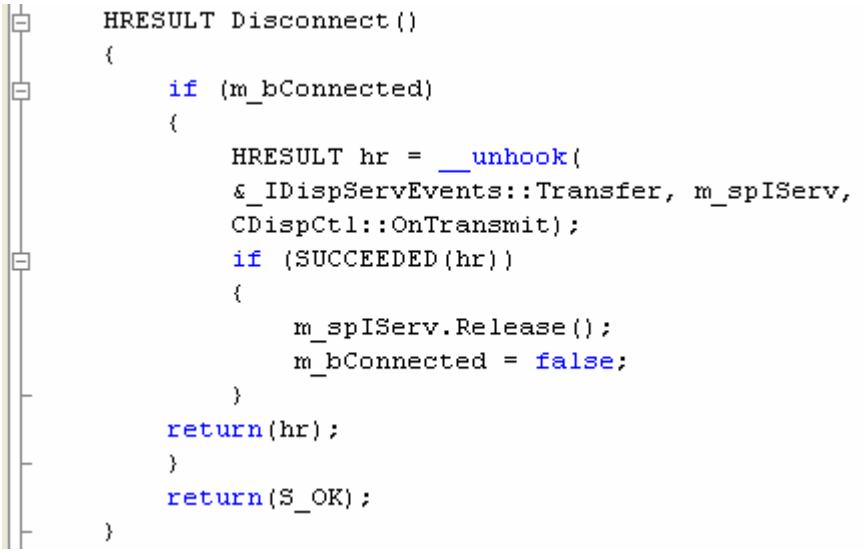
Listing 11.

The implementation of the `Disconnect()` method uses `__unhook` to disconnect the event and `Release()` to release the instance of `CDispServ` created earlier.

To implement the `Disconnect()` method

Below the `Connect()` method, add the following code directly:

```
HRESULT Disconnect()
{
   if (m_bConnected)
   {
      HRESULT hr = __unhook(&_IDispServEvents::Transfer, m_spIServ,
      CDispCtl::OnTransmit);
      if (SUCCEEDED(hr))
      {
         m_spIServ.Release();
         m_bConnected = false;
      }
      return(hr);
   }
   return(S_OK);
}
```



```
HRESULT Disconnect()
{
    if (m_bConnected)
    {
        HRESULT hr = __unhook(
        &_IDispServEvents::Transfer, m_spIServ,
        CDispCtl::OnTransmit);
        if (SUCCEEDED(hr))
        {
            m_spIServ.Release();
            m_bConnected = false;
        }
    return(hr);
    }
    return(S_OK);
}
```

Listing 12.

Adding the Event Handler

For the control to respond to events fired from the server, you need to implement a handler (called `OnTransmit()`) for the `Transmit()` event. The `OnTransmit()` event handler takes the data passed in from the `Transfer()` event and places it in the `m_OutText` data member. It then calls `FireViewChange()` (not yet implemented), which updates the control by displaying the contents of the `m_OutText` data member.

To implement a handler for the Transmit event

Add the following code to the source file, below `Disconnect()`:

```
HRESULT OnTransmit(VARIANT data)
{
   if (data.vt == VT_BSTR)
      m_OutText = data;
   FireViewChange();
   return(S_OK);
}
```

```
HRESULT OnTransmit(VARIANT data)
{
    if (data.vt == VT_BSTR)
        m_OutText = data;
    FireViewChange();
    return(S_OK);
}
```

Listing 13.

The last method you will implement is the `Send()` method. This method sends data to the server object.

To implement the `Send()` method

Add the following code to the source file, below `OnTransmit()`:

```
HRESULT Send(VARIANT data)
{
   if (m_bConnected)
      m_spIServ->Send(data);
   return(S_OK);
}
```

```
HRESULT Send(VARIANT data)
{
    if (m_bConnected)
        m_spIServ->Send(data);
    return(S_OK);
}
```

Listing 14.

Modifying the `OnDraw()` Method

The final modification you will make is to the `CDispCtl::OnDraw` method. The `OnDraw()` method needs to output the contents of the data member `m_OutText` to the screen.

To modify the `OnDraw()` method

Replace the body of the existing `OnDraw()` method with the following:

```
USES_CONVERSION;
LPCTSTR text = OLE2CT(m_OutText.bstrVal);
RECT& rc = *(RECT*)di.prcBounds;
Rectangle(di.hdcDraw, rc.left, rc.top, rc.right, rc.bottom);
SetTextAlign(di.hdcDraw, TA_CENTER|TA_BASELINE);
TextOut(di.hdcDraw,
   (rc.left + rc.right) / 2,
   (rc.top + rc.bottom) / 2,
   text,
   lstrlen(text));

return S_OK;
```

```
// IDispCtl
public:
        HRESULT OnDraw(ATL_DRAWINFO& di)
        {
            USES_CONVERSION;
            LPCTSTR text = OLE2CT(m_OutText.bstrVal);
            RECT& rc = *(RECT*)di.prcBounds;
            Rectangle(di.hdcDraw, rc.left,
                rc.top, rc.right, rc.bottom);
            SetTextAlign(di.hdcDraw, TA_CENTER|TA_BASELINE);
            TextOut(di.hdcDraw,
            (rc.left + rc.right) / 2,
            (rc.top + rc.bottom) / 2,
            text,
            lstrlen(text));

            return S_OK;
        }
```

Listing 15.

The **DispClient** control is now complete. Build the control by selecting **Build DispClient** from the **Build** menu.

Step 7: Using the Client Control

In this step, you will use the client control to invoke the methods.

To use the client control

1. On the **Tools** menu, click **ActiveX Control Test Container**. This starts **Tstcon32.exe**.
2. Click the **New Control** button on the toolbar to insert a client controls (**CDispCtl**).

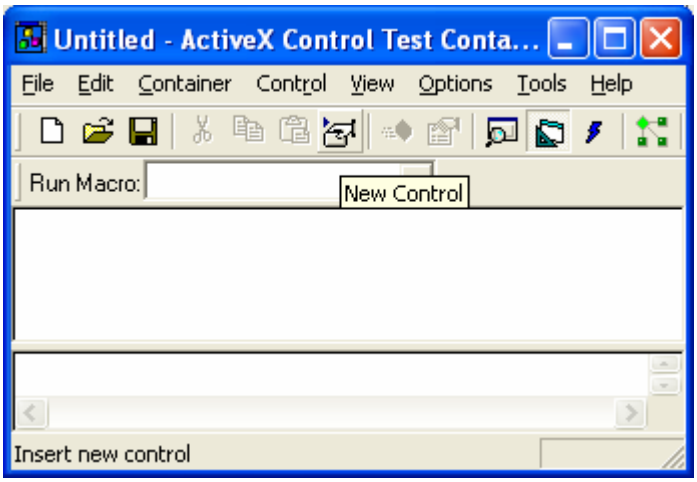


Figure 19: Inserting new control to **ActiveX Control Test Container**.

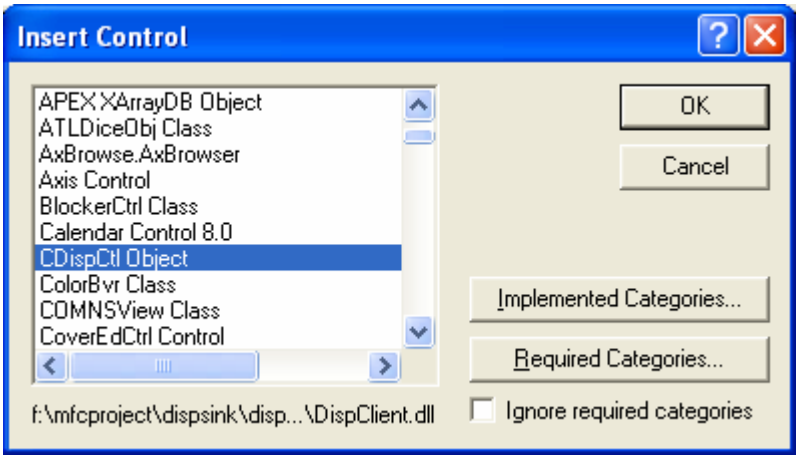


Figure 20: Selecting control in **ActiveX Control Test Container**.

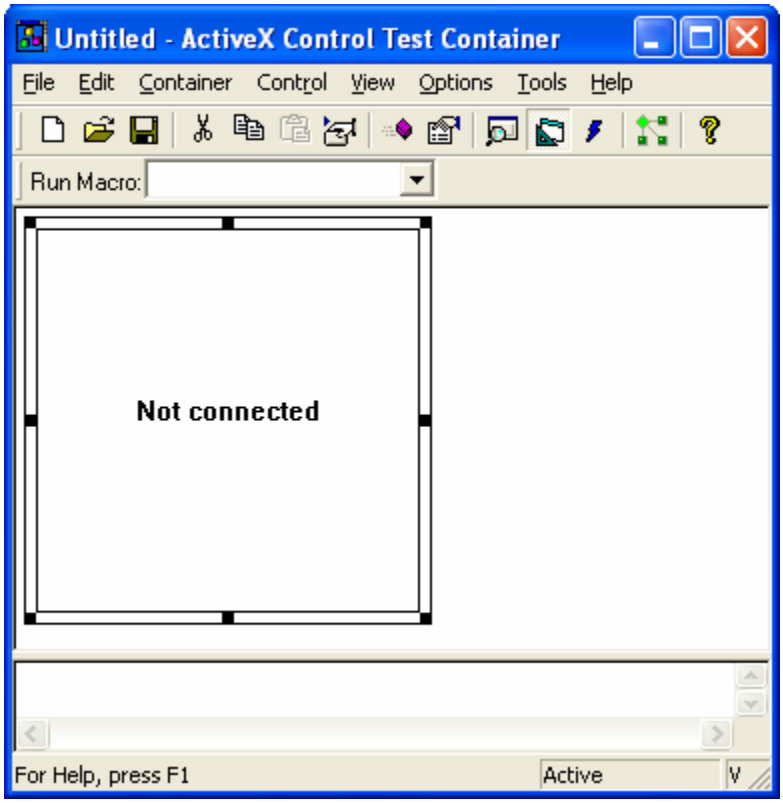


Figure 21: **CDispCtl** object in **ActiveX Control Test Container**.

3. Right-click the control and click **Invoke Method** on the shortcut menu. The **Invoke Method** dialog box appears.
4. Ensure that **Method Name** is set to **Connect** and click **Invoke**.

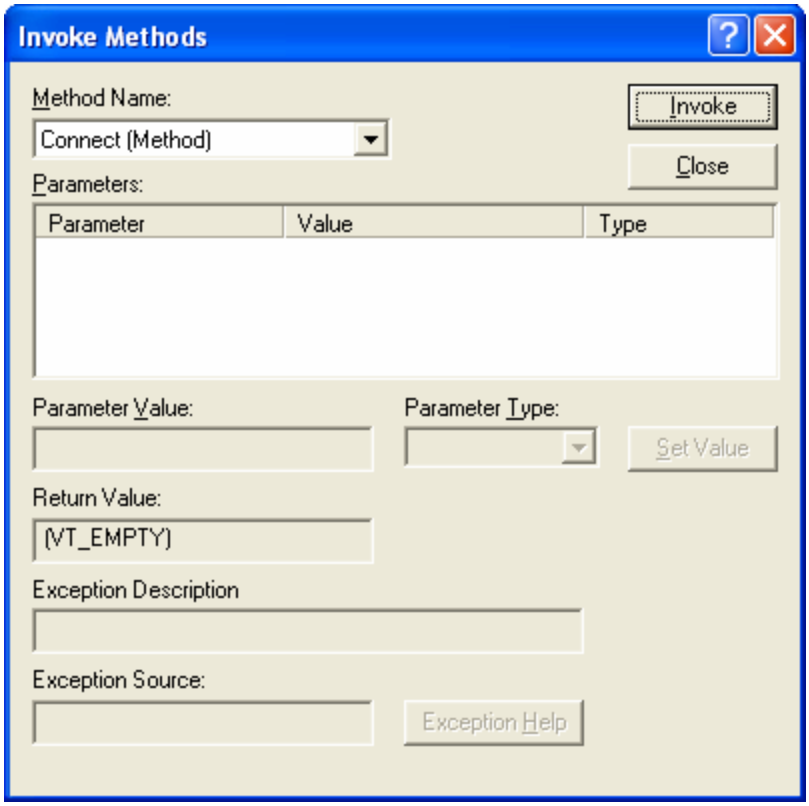


Figure 22: Invoking the `Connect()` method of the **CDispCtl** control in **ActiveX Control Test Container**.

5. Right-click the control and click **Invoke Method** on the shortcut menu. The **Invoke Method** dialog box appears.
6. Set **Method Name** to **Send**.
7. Change the parameter type on the **Send** method to VT_BSTR.
8. Enter any string in the **Parameter Value** box.

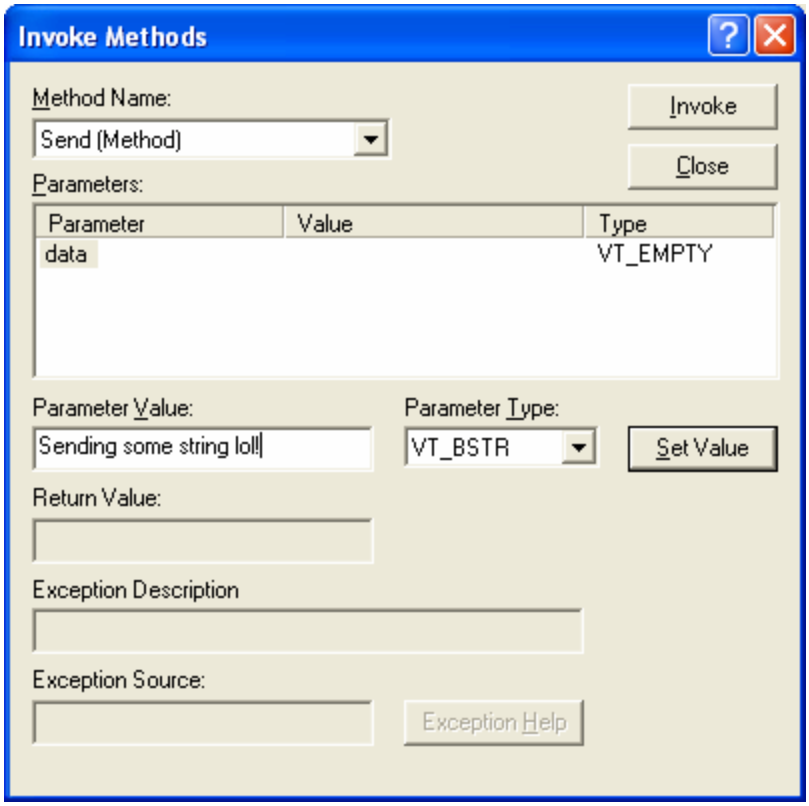


Figure 23: Invoking the `Send()` method of the **CDispCtl** control in **ActiveX Control Test Container**.

9. Click the **Invoke** button. The string is displayed in all connected controls.

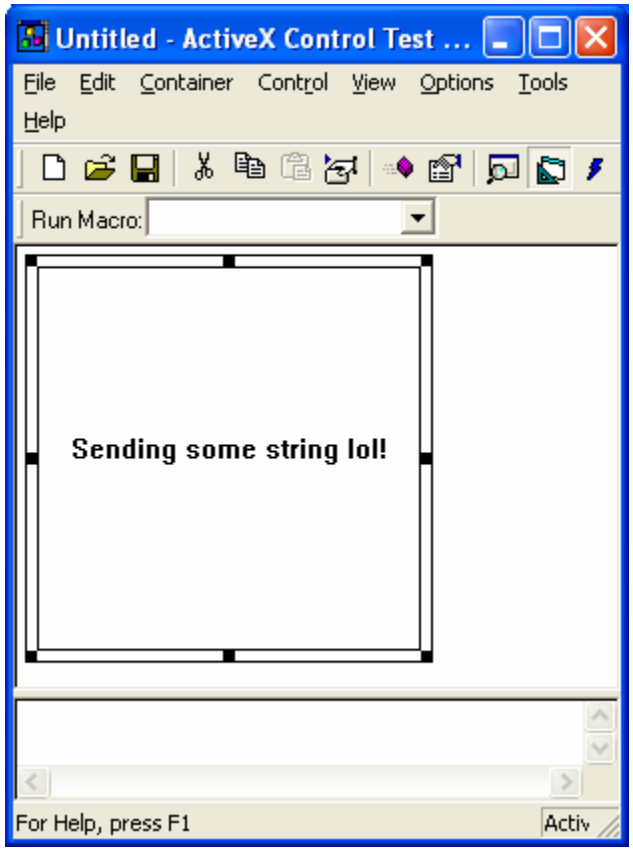


Figure 24: **CDispCtl** in action.

----------End----------

Further reading and digging:

1. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
2. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
3. MSDN Library
4. DCOM at MSDN.
5. COM+ at MSDN.
6. COM at MSDN.
7. Windows data type.
8. Win32 programming Tutorial.
9. The best of C/C++, MFC, Windows and other related books.
10. Unicode and Multibyte character set: Story and program examples.