COM Automation in BCB4

This paper is designed to explain how to create a simple Automation server and client in Borland C++Builder 4.0. Rather than explaining COM Automation in depth, instead I will focus on highlighting the differences between the COM implentation in CBuilder 3.0 and CBuilder 4.0.

If you already understand how to build COM servers and clients in CBuilder 3.0, then you can jump here to see the differnce between the 3.0 and 4.0 implementation. In particular, the class used to create the server has been changed from a standard C++ class to a template class. Jump here if you don't care at all about the theory of how templates work in BCB's COM implementation, and just want to see how to create an instance of an automation server in BCB4.

The source for the example used in the paper is available here.

Creating a Simple Automation Server

To get started, you should launch C++Builder and create a new project. Create a new directory where you can house your server. Beneath that new directory create two directories, one called Client and the other called Server. Save your new project in the Server directory. Call the main form MainServer.cpp and call the project file MyServer.dpr.

Select File | New from the menu, and turn to the ActiveX page. Choose Automation Object, and click the Okay button. Enter in the name of your CoClass, which in this case can be called MyObject. Leave the threading model at Apartment, enter a short description if you wish, and leave the Generate Event Code button unchecked. Click Ok.

The Type Library Editor appears. In the Object List Pane on the left you will see your CoClass, called MyObject, listed. Above it is a default interface implemented on the CoClass called IMyObject. We are going to add two methods to IMyObject.

Right click on IMyObject and choose New | Method from the menu. Call your method GetName. Turn to the Parameters page and click the Add button near the bottom of the editor. Name your parameter Value and set its type to BSTR *. This is a pointer to a BASIC string. If you click on the Modifier field an elipsis (...) button will appear on the far right of the editor. Click on it to pop up the Paramter Flags dialog that will allow you to select the RetVal and Out flags for the parameter. Click the Okay button. When you are done, the editor should look as it does in Figure 1.

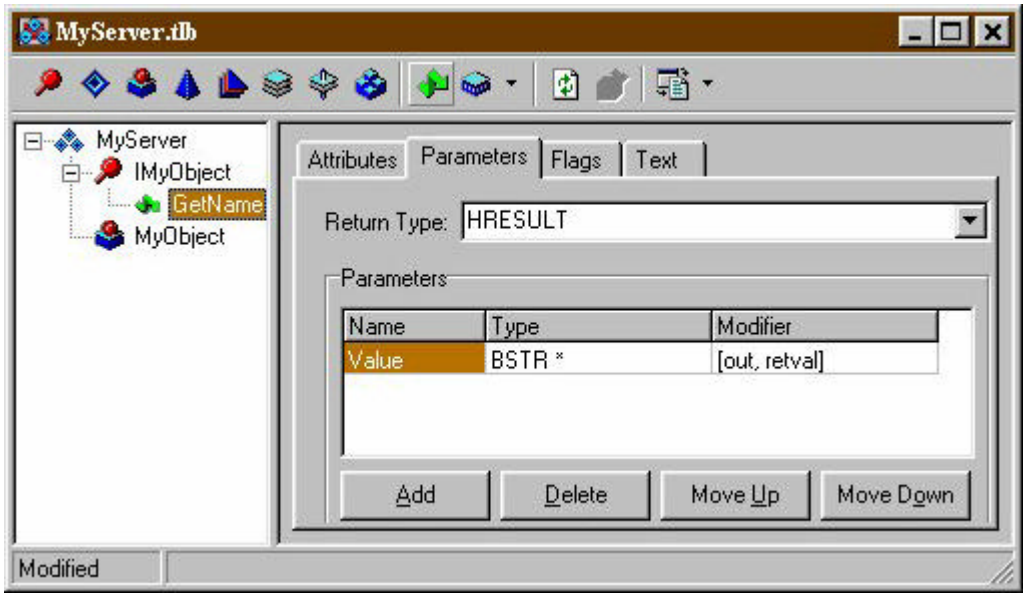

Figure 1: Creating an Out and RetVal parameter for a function that takes a BSTR.

It is now time to add your second method to your server. Click once on the Object List Pane and then right click on IMyObject, just as you did when creating the GetName method. This time name the method you are creating Square. Click on the parameters page and create two parameters, both of type long, one an in parameter, and one an out parameter. When you are done the editor should look as it does in Figure 2.

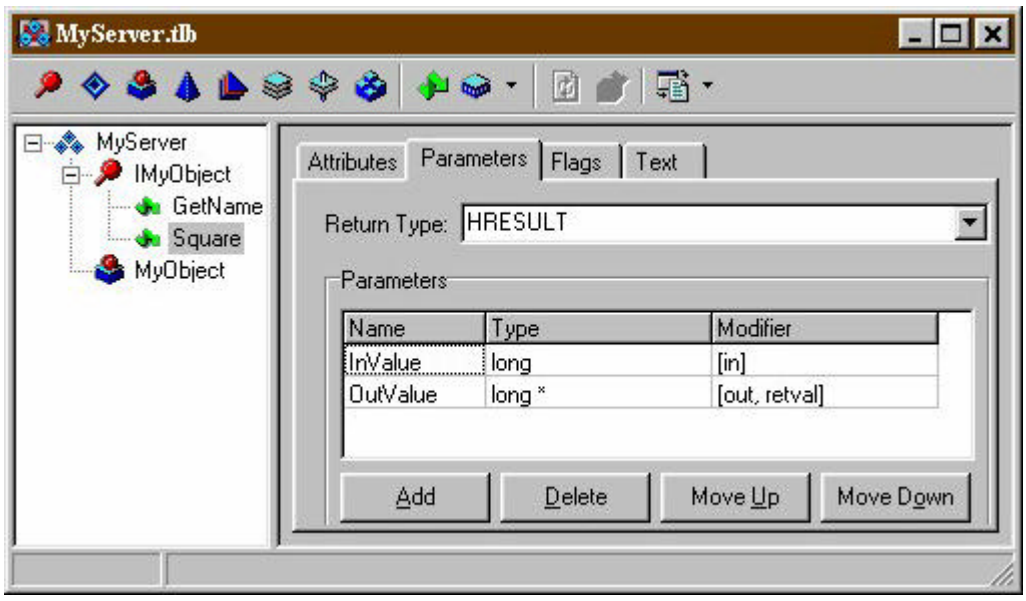

Figure 2: Creating two parameters in the type library editor.

When you save your work, you will find that a file called MyObjectImpl.cpp has been created. Open this file in the BCB text editor and you will find that the stubs for your two methods have been created for you automatically. Fill them in as follows:

```
STDMETHODIMP TMyObjectImpl::GetName(BSTR* Value)
{
  *Value = WideString("Guatemalan Server").Detach();
  return S_OK;
}
```

```
STDMETHODIMP TMyObjectImpl::Square(long InValue, long* OutValue)
{
  *OutValue = InValue * InValue;
  return S_OK;
}
```

This is the implementation for the interface that you declared in the type library editor. As you can see, it is very simple and straightforward. All you are doing is filing in the code for some very simple methods. The whole point is that all the difficult background work has been done for you, and you need do nothing more than implement your methods just as you would if you were planning on calling them locally.

At this stage, the creation of your server is complete. You can now save your work, then compile and run the server once to register it with the system. When you are done you could proceed directly to the creation of your client application.

In this particular case, however, I am not going to move directly to the client, but will instead take a moment to view the code that has been autogenerated by BCB. I do this in part so that you can understand the difference between the implementation in BCB 4 and BCB 3, and in part so you can move confidently through this code with some understanding of what it means. I want to stress however, that you need not understand this auto-generated code in order to use our COM implementation. You can treat the whole thing as a black box if you would like. I discuss it here only because I know that there are those who will be curious about its meaning and significance.

In particular, it's helpful to understand how the code works if you are having trouble instantiating or calling an object. If you can't work properly with an instance of your object for some reason, then you are going to open up your XXX_TLB.h file and try to see what is wrong. At that time, you will be glad to understand at least a little bit about how the code in the XXX_TLB.h file is structured.

Understanding the Auto-generated Code.

The auto-generated code found in BCB4 differs from the auto-generated code in BCB 3 because it uses templates rather than simple C++ classes. In particular, you will find that MyServer_TLB.h is full of template classes that were never declared in BCB 3. Or rather, they were declared in BCB3, but they were declared there as standard C++ classes rather than as template classes. The use of template classes is appropriate, since it ties in nicely with the underlying COM implementation, which is called the Active Template Library, or ATL. The ATL was created by Microsoft as a wrapper of COM, and it consists primarily as a series of templates.

Looking for a deeper explanation of the change, I asked R&D to explain why they switched to templates. They were nice enough to offer up the following three reasons, which I have edited only very slightly

- Templates allow you to pay only for what you use: Using templates allows the compiler to only expand the methods that the client is using. For example, imagine a Client that automates WORD: It's likely to only call a handful of the methods exposed by WORD. If we were to put all the methods in a .CPP, the resulting .OBJ would contain all the methods and these would all be linked in (NOTE: The linker can perform some smart linking but that technology is limited

when it comes to virtual methods).

- When the full implementation is complete, Templates will provide additional ease of use for clients. The client application will simply need to include SERVER_TLB.H, there will be no need to explicitly add the .CPP to the project or to link with a .LIB file. (With the first versions of CBuilder 4.0 to go out the door, you still have to include the CPP to get at the GUIDs used during object creation. This requirement will disappear in later versions of BCB4. Even as it is now, there is nothing in the CPP but these GUIDs, so the cost of linking in the code is relatively small.)
- Templates allow you to pay only once for multiple use: You could argue that using inline functions in a header achieves the above two goals. However, you'd pay for each use and reuse of a particular routine. Again, imagine a client automating Word. More than once the client is likely to call WordApp->ActiveDocument->DoSomething. With templates, the method to retrieve the ActiveDocument is expanded only once. If we were to use inline functions, each call would result in a separate expansion.

Hopefully these explanations will help you understand why we are now using templates. Even if they don't mean much to you, the only really important oint to grasp is that we are now using templates where we once used standard C++ classes.

Let's get started exploring these template classes by emphasizing the difference between the interface you declared in the type library editor and the implementation found in MyObjectImpl.cpp. Here is the interface you created in the type library editor, as it is declared in MyServer_TLB.h:

```
interface IMyObject : public IDispatch

{

public:

  virtual HRESULT STDMETHODCALLTYPE

    GetName(BSTR* Value/*[out,retval]*/) = 0; // [1]

  virtual HRESULT STDMETHODCALLTYPE Square(long InValue/*[in]*/,

    long* OutValue/*[out,retval]*/) = 0; // [2]


#if !defined(__TLB_NO_INTERFACE_WRAPPERS)


  BSTR __fastcall GetName(void)

  {

    BSTR Value= 0;

    OLECHECK(this->GetName(&Value));

    return Value;

  }


  long __fastcall Square(long InValue/*[in]*/)

  {

    long OutValue;
```

```
    OLECHECK(this->Square(InValue, &OutValue));
    return OutValue;
  }

#endif //   __TLB_NO_INTERFACE_WRAPPERS

};
```

This interface is called IMyObject, and it descends from IDispatch. IDispatch is an interface that implements methods that make it possible for you to call a COM object from tools such Visual Basic, Word or Excel. A C++ programmer usually does not need to rely on the methods found in IDispatch, but including them in an implementation means that the object can be used in a wider variety of settings such as Microsoft Word or Excel. You never have to explicitly implement IDispatch, as that task is handled for you behind the scenes by the ATL.

When you were in the type library editor you declared some of the parameters you were working with as retval. A retval parameter is one that is returned from a function. In COM, all functions must return HResult, which is an error value declaring whether the call to the function succeeded or failed. However, it is possible to do some hand waving that makes it appear to the programmer that the function returns a value such as BSTR or a long, rather than an HRESULT.

If I had not declared my functions with retvals, then the inteface for this class would have looked much simpler:

```
interface IMyObject : public IDispatch
{
public:
  virtual HRESULT STDMETHODCALLTYPE
    GetName(BSTR* Value/*[out]*/) = 0; // [1]
  virtual HRESULT STDMETHODCALLTYPE Square(long InValue/*[in]*/,
    long* OutValue/*[out]*/) = 0; // [2]
};
```

The code that is missing from this implementation of the interface looks like this:

```
#if !defined(__TLB_NO_INTERFACE_WRAPPERS)

  BSTR __fastcall GetName(void)
  {
    BSTR Value= 0;
    OLECHECK(this->GetName(&Value));
    return Value;
```

```
  }


  long __fastcall Square(long InValue/*[in]*/)
  {
    long OutValue;
    OLECHECK(this->Square(InValue, &OutValue));
    return OutValue;
  }
```

As you can see, this code provides implementations of the GetName and Square functions that return a BSTR and a long rather than an HRESULT. If there is an error in this code, then the OLECHECK macro would return an HRESULT with the error value in it. These implementations of the functions are just part of the hand waving that makes it look like a method is returning something other than an HRESULT.

Now that you have had a look at it, I'm going to tell you that in BCB itself, this interface is not really that important. Instead, a second interface, called a smart interface, is the one you will use in most cases. In MyServer_TLB.h You will find that beneath this first declaration for the interface to your object is a template class called TCOMIMyObjectT:

```
template &ltclass T /* IMyObject */ >
class TCOMIMyObjectT : public TComInterface&ltIMyObject>,
  public TComInterfaceBase&ltIUnknown> {
public:
  TCOMIMyObjectT() {}
  TCOMIMyObjectT(IMyObject *intf, bool addRef = false) :
    TComInterface&ltIMyObject>(intf, addRef) {}
  TCOMIMyObjectT(const TCOMIMyObjectT& src) :
    TComInterface&ltIMyObject>(src) {}
  TCOMIMyObjectT& operator=(const TCOMIMyObjectT& src)
    { Bind(src, true); return *this;}


  HRESULT         __fastcall GetName(BSTR* Value/*[out,retval]*/);
  BSTR            __fastcall GetName(void);
  HRESULT         __fastcall Square(long InValue/*[in]*/,
    long* OutValue/*[out,retval]*/);
  long            __fastcall Square(long InValue/*[in]*/);


};
typedef TCOMIMyObjectT&ltIMyObject> TCOMIMyObject;
```

It is this second interface, the smart interface, that you will deal with most often in your code. Notice that it declares the methods GetName and Square both with the retval passed as a normal parameter, and returned as a function result. Notice also that a series of constructors and operators are provided to make it easy for you to work with this class.

I want to reiterate that it is this template class, and not the interface for IMyObject, that you will use most often in BCB. Furthermore, in BCB 3, TCOMIMyObjectT would have been called TCOMIMyObject, and would have been implemented as a regular C++ class, not as a template. Note the T at the end of TCOMIMyObjectT, that specifies that this is a template class, and not just a standard C++ class.

To help bind the code in BCB4 to BCB3, there is a typedef immediately beneath the declaration for TCOMIMyObjectT that maps the object to a variable called TCOMImyObject. An important side benefit of this declaration is that allows you to avoid having to work with the complicated template syntax yourself.

It would be tempting to assume that TCOMIMyObject is the correct instance of the class to use when calling this method from a server. However, that is not correct, instead a special class creator is declared for you. I will discuss that class creator in just one moment. First I need to briefly discuss the disp interface declarations.

Beneath the declaration of TCOMIMyObjectT is the declaration for a second class called IMyObjectDispT. IMyObjectDispT is a dispatch interface, which is used primarily by Visual Basic, or by other disadvantage languages such as the versions of Visual Basic found in Excel, Word or scripting languages. In short, the Disp interface is not important to you as a C++ programmer, and is provided only as a service to those who want to program in VB. The Disp interface is implemented automatically for you by BCB and the ATL, but you need not worry about it in your own programs, other than to know that it is automatically available to those who program in VB.

In MyServer_TLB.h you will also find implementations of the methods for both IMyObjectDispT and TCOMIMyObjectT. These implementations are relatively trivial, and represent nothing more than plumbing that you need not consider in your day to day programming.

It is not until you reach the very bottom of MyServer_TLB that you at last find the template class that is used to create an instance of your interface. In many ways, this is the single most important bit of code in the whole file, since this is the code that your client will call when it wants to create an instance of the object supported by your server:

```
typedef TCoClassCreatorT<TCOMIMyObject,
  IMyObject, &CLSID_MyObject, &IID_IMyObject>CoMyObject;
```

This declaration declares a variable called CoMyObject which is of type TCoClassCreatorT. TCoClassCreatorT is a template class declared in the VCL file called UtilCls.h:

```
template <class TOBJ, class INTF, const CLSID* clsid,
  const IID* iid>
class TCoClassCreatorT : public CoClassCreator
{
```

```
public:
  static TOBJ    Create();
  static HRESULT Create(TOBJ& intfObj);
  static HRESULT Create(INTF** ppintf);

  static TOBJ    CreateRemote(LPCWSTR machineName);
  static HRESULT CreateRemote(LPCWSTR machineName, TOBJ& intfObj);
  static HRESULT CreateRemote(LPCWSTR machineName, INTF** ppIntf);
};
```

As you can see, this template class provides methods called Create and CreateRemote, than can be used to create instances of your object. Several instances of these constructors are declared so that you can find ways to meet all your potential programming needs.

Given the typedef that declares the variable CoMyObject to be of type TCoClassCreatorT, you can at last see how to create an instance of your object:

```
  TCOMIMyObject MyObject = CoMyObject::Create();
```

You will see this line of code again, later in the article, when I discuss how to create the client. For now, all you need to know is that it is the code used by your client to create an instance of the object you want from your server.

Those familiar with BCB 3.0 will recognize this line of code. It is the exact same line you used to create an instance of your object in BCB 3. In short, from an external programmer's point of view, the implementation of COM Automation in BCB 3 and BCB 4 is identical. It is only when you look beneath the surface that you see the big impact that the introduction of templates has had on the structure of your XXX_TLB.h file.

As I said earlier, there is no real need for you to understand what goes on inside MyServer_TLB.h, and you definitely should never attempt to edit this file unless you are sure you know what you are doing. Nevertheless, I believe that many programmers want to have at least a minimal understand of how MyServer_TLB.h works, and as a result I have written a few words that can hopefully help you comprehend what the new template code is all about.

The Implemenation of your Interface

In this section of the paper I will briefly discuss the implementation of your interface. The classes found in MyServer_TLB.h represent the plumbing that makes your implementation function. To oversimplify the matter somewhat, the interface for your class is declared in MyServer_TLB.h, and the implementation of that interface is declared in MyObjectImpl.h and MyObjectImpl.cpp. In this section I'm going to briefly discuss that implementation.

Here is the declaration for the implementation of IMyObject, as found in MyObjectImpl.h:

```
class ATL_NO_VTABLE TMyObjectImpl :
  public CComObjectRootEx<CComSingleThreadModel>,
```

```
  public CComCoClass<TMyObjectImpl, &CLSID_MyObject>,
  public IDispatchImpl<IMyObject, &IID_IMyObject, &LIBID_MyServer>
{
public:
  TMyObjectImpl()
  {
  }

  // Data used when registering Object
  //
  DECLARE_THREADING_MODEL(otApartment);
  DECLARE_PROGID("MyServer.MyObject");
  DECLARE_DESCRIPTION("My Object");

  // Function invoked to (un)register object
  //
  static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
  {
    TTypedComServerRegistrarT<TMyObjectImpl>
    regObj(GetObjectCLSID(), GetProgID(), GetDescription());
    return regObj.UpdateRegistry(bRegister);
  }



BEGIN_COM_MAP(TMyObjectImpl)
  COM_INTERFACE_ENTRY(IMyObject)
  COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

// IMyObject
public:

  STDMETHOD(GetName(BSTR* Value));
  STDMETHOD(Square(long InValue, long* OutValue));
};
```

The glue that ties this implementation to the IMyObject interface is the Active Template Library, or ATL. Remember that the ATL is a standard created by Microsoft,

and used by BCB.

This declaration is so complicated because it relies on the ATL, which is not a trivial API. This paper is not about the ATL, nor, in my opinion is it important that you understand how the ATL works. The ATL is designed to implement all the details of COM programming for you, so that you don't need to think about them.

Just as most developers don't understand how a compiler creates an instance of a standard C++ object, there is no need for you to understand how the ATL makes it possible for you to create a COM implementation of an interface. Of course, if you understand the ATL, then that is all good and well. In my opinion, however, the ATL is just plumbing, and not something that a programmer needs to understand.

It's interesting to note that the BCB wrapping of the ATL was created specifically so you would not have to wrestle with the details of the ATL. The ATL itself was in turn created so you would not have to wrestle with the details of COM. You will, of course, be forgiven if you find it a bit ironic that Microsoft created a wrapper of COM that is itself so complicated that it needs a wrapper!

From your point of view, the only thing that matters in this entire declaration are the last two lines:

```
STDMETHOD(GetName(BSTR* Value));
STDMETHOD(Square(long InValue, long* OutValue));
```

Here you find the declaration for your methods that you implemented in MyObjectImpl.cpp.

Creating the Client

It is now time to create the client. This process is nearly identical to the steps you undertook in CBuilder 3, only now you use the template class discussed above when creating an instance of your client.

To create the client the first thing you need do is add a new application to your project group. This will be the Client application that talks to your server. To get started, choose View | Project Manager from the BCB menu. Click the New button at the top of the dialog, or right click on the ProjectGroup icon at the top of the list pane and choose New from the menu. Select the icon that allows you to create a new application.

Save your work to disk in the Client directory you created earlier in this paper. Save the main form as MainClient.cpp and save your project as TempClient.dpr. Save the BPG file in the directory beneath your Server and Client directory.

The next step is to include the MyServer_TLB.cpp file in your project, so that you can reference the functions that will allow you to create an instance of the server. To include this file in your client project, you can just click on it in the project manager and drag it from the server project to the client project. If this does not work for you, then just add the CPP file to the TempClient.cpp file:

```
#include <vcl.h>
#pragma hdrstop
USERES("TempClient.res");
```

```
USEFORM("MainClient.cpp", Form2);

USEUNIT("..\Server\MyServer_TLB.cpp");
```

You should also make sure that the Server directory is on the include path for your client project. Double click on TempClient.exe in the Project Manager to make sure it is your selected project. Choose Project | Options | Directories/Conditionals from the menu. Click on the Include path icon and add ..\Server to your path as shown in Figure 3. Now #include "MyServer_TLB.h" at the top of your source for your main form.

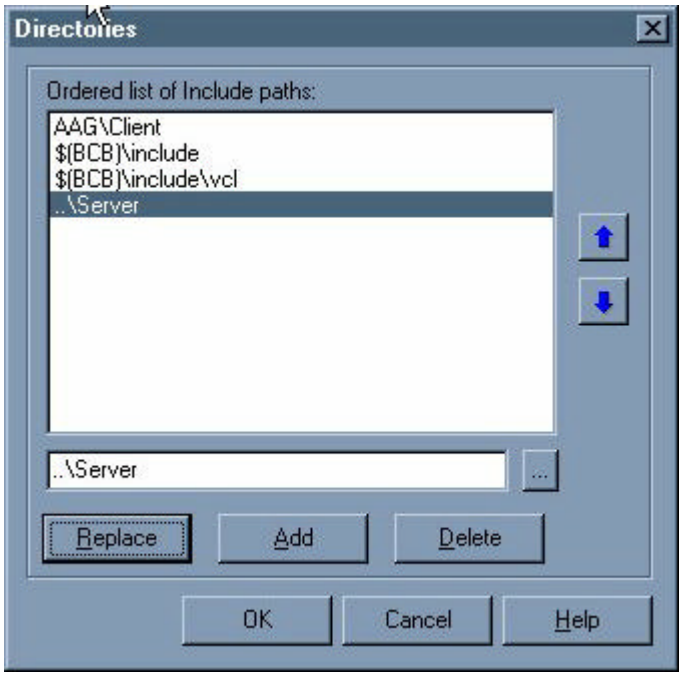

Figure 3: Adding the path to the server in the Include Path section of your client application

Here is the complete source for the main form of your client application:

Listing1: The header for the client's main form.

```
#ifndef MainClientH

#define MainClientH


#include <Classes.hpp>

#include <Controls.hpp>

#include <StdCtrls.hpp>

#include <Forms.hpp>


class TForm2 : public TForm

{

__published:    // IDE-managed Components

  TButton *CreateObjBtn;

  TButton *Button1;

  TEdit *Edit1;
```

```
  void __fastcall CreateObjBtnClick(TObject *Sender);
  void __fastcall Button1Click(TObject *Sender);
private:        // User declarations
  TCOMIMyObject FMyObject;
public:         // User declarations
  __fastcall TForm2(TComponent* Owner);
};

extern PACKAGE TForm2 *Form2;

#endif
```

Listing2: The source for the client's main form.

```
#include <vcl.h>
#pragma hdrstop
#include "MyServer_TLB.h"
#include "MainClient.h"

#pragma package(smart_init)
#pragma resource "*.dfm"
TForm2 *Form2;

__fastcall TForm2::TForm2(TComponent* Owner)
        : TForm(Owner)
{
}

void __fastcall TForm2::Button1Click(TObject *Sender)
{
  FMyObject = CoMyObject::Create();
  ShowMessage(FMyObject->GetName());
}

void __fastcall TForm2::Button1Click(TObject *Sender)
{
  long Result;
```

```
  long SqrNum = StrToInt(Edit1->Text);
  FMyObject->Square(SqrNum, &Result);
  Edit1->Text = IntToStr((int)Result);
}
```

Note the declaration for an instance of TCOMIMyObject in the headers main form:

```
TCOMIMyObject FMyObject;
```

You can use this instance of the object when working with the server.

As you can see, this implementation of the client in your cpp file is exactly the same as in BCB 3.0. The fact that the object used to create the server is now a template makes no difference at all in your code. In particular, here is the line of code used to create an instance of your server:

```
FMyObject = CoMyObject::Create();
```

The client calls Create on CoMyObject, which happens to be a template rather than a standard C++ object. What's happening behind the scenes is different, but the code you write on the client is the same.

The rest of the client application is trivial. Once you have an instance of TCOMIMyOjbect back from the server, then you can call its methods just as you would an object inside your own application. This is the point of an Automation server: it allows you to instantiate an instance of an object that is located in a separate application, even if that application is located on a separate machine. (If it is on a separate machine, then call CreateRempte rather than Create.) Once you have a reference to the object, you can call its methods as if it were in your own application. Performance is necessarily a bit slower, but the act of calling the methods is the same.

Summary

As you can see, the changes between the COM model is BCB 3 and BCB 4 are trivial. I show them to you here only so you can isolate the differences, and see why they were implemented. In particular, the class placed in the TLB file for creating an instance of your server has been changed from a standard C++ class to a template class.

If you are new to BCB, COM or to Automation, then you will likely still have a lot of questions that have not been answered by the text of this article. Many of the answers you seek are available in the BCB documentation, or in my book C++Builder 3 Unleashed.

I personally do not have time to answers questions about this technology, but if you want to make suggests, or you have bug reportson on this article that you want to submit, then you can write mail to Charlie Calvert

Back To Top
Home Page