Under Construction: VisiBroker For Delphi 5

by Bob Swart

Some months ago, we examined CORBA and Delphi in this column. Specifically, we created a CORBA server in Delphi 5 and made it communicate with a CORBA client written in JBuilder. At that time, CORBA in Delphi was supported through the use of the Type Library. There was no direct mapping from IDL to Pascal, or at least not one that was available to most of us. Since then, things have changed...

CORBA

Back in Issue 43, I described how to use Delphi 4 to create CORBA clients and servers that could communicate with each other. This was CORBA using the Type Library, with a little COM layer. Two issues later (May 1999), Hubert A Klein Ikkink joined me for a project where we created a Delphi 4 CORBA server and a JBuilder 2 CORBA client. This was possible because the Type Library of Delphi is able to produce standard CORBA Interface Definition Language (IDL) files. An IDL file is one of the features that makes CORBA independent of both platform and language. IDL can be used to specify the module, methods, arguments, structures and even exceptions. The crux is the mapping between IDL and a programming language (on any platform), like an IDL2Java. Back in Issue 45, Delphi 4's Type Library produced a CORBA IDL file, which could be used by JBuilder's VisiBroker IDL2Java compiler to produce the Java code for the CORBA client.

What we didn't mention (at least not explicitly in that article), is that the other way around would have been far more difficult. Sure, a JBuilder CORBA server can produce a CORBA IDL file, but the hard part would be to import this IDL file into Delphi 4's Type Library. While we can paste an IDL file inside the Type Library text pane and hit the `Refresh Implementation` button, this doesn't always work for me (or generate the expected result, for that matter). What we'd need is a true IDL2Pas, so any CORBA IDL file could be turned into a set of Object Pascal files. Unfortunately, as you may know, this didn't exist at the time. And Delphi 5 Enterprise did not ship with an IDL2Pas (which was very disappointing, by the way). But December 1999 brought us Sinterklaas and Santa Claus and a true IDL2PAS from Inprise...

VisiBroker For Delphi 5

In the first half of December 1999, Inprise decided to make VisiBroker for Delphi 5 available as a free download from www.borland.com/visibroker/delphi/ (you just need to complete a short survey). Note that you need your official Delphi 5 serial number and registration key (the same ones you needed to install Delphi 5 Enterprise), and you also need some units (like the `CORBA` unit) that are only available in the Enterprise edition of Delphi 5, so don't bother downloading this add-on tool if you don't have access to Delphi 5 Enterprise.

After installing VisiBroker for Delphi 5, you can find both the IDL2PAS.BAT and JAR files in the BIN directory of Delphi 5, as well as a number of interesting examples in the DEMOS\IDL2PAS directory and new documentation in the IDL2PAS DOCS directory. Finally, check out the SOURCES\RTL\CORBA directory for a number of new files (CORBA.PAS and ORBPAS30.PAS). The IDL2PAS.PDF file in the DOCS directory, the VisiBroker for Pascal Reference Guide, is especially interesting (it's in Adobe Acrobat format).

IDL2PAS

The IDL2PAS utility takes the IDL file and generates the interface and client stub code (but not the server skeleton). So, if we run the IDL2PAS batch file on the BOBNOTES.IDL file (from Issue 45, see Listing 1), we get two resulting units: BOBNOTES_I.PAS and BOBNOTES_C.PAS, the interface and the client stub respectively. Note that no server skeleton code is generated. Yep, that's right: VisiBroker for Delphi currently supports Delphi CORBA clients only (based on IDL files from 'other' CORBA servers). So the IDL2PAS currently only generates `xxx_I` (interface) and `xxx_C` (client) units, and not `_S` (server) units.

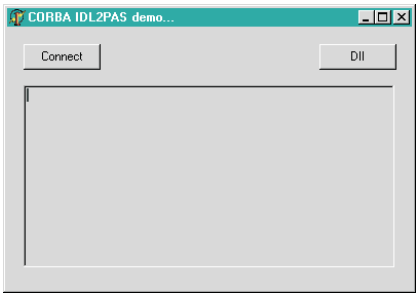
Armed with this BOBNOTES.IDL file and the two generated units, we can now write a new Delphi 5 CORBA client to connect and communicate with the existing CORBA server.

The CORBA client's main form is defined as a memo with two buttons: `Connect` (the static way) and

➤ *Listing 1: BobNotes.IDL*

```
module BobNotes
{
  interface ICorBobNotes;
  interface ICorBobNotes
  {
    void GetLines(in wstring User, in wstring Password, out wstring Lines);
    void SetLines(in wstring User, in wstring Password, in wstring Lines);
  };

  interface CorBobNotesFactory
  {
    ICorBobNotes CreateInstance(in string InstanceName);
  };
};
```



➤ *Figure 1:*
Delphi 5 CORBA client.

secondly `DII` (the dynamic way, using Dynamic Interface Invocation).

Actually, there are now three ways for a Delphi CORBA client to connect to a Delphi CORBA server. In order to show you the strengths (ease of use) and weaknesses (flexibility) of each of these different solutions, I'll now implement all three. In the end, it should be obvious which one should be avoided, and which one(s) should be preferred. So, let's take a deep breath, and get back to...

CORBA: The Old Way

Delphi 4 Client/Server and Delphi 5 CORBA servers use the Type Library to store the interfaces, methods, parameter info, etc. If (and only if) we know beforehand that we need to connect to a Delphi 4 Client/Server or Delphi 5 Enterprise CORBA server, then we can use the old technique to connect to it. The 'old' technique is based on the Type Library import unit, which contains the definitions of the CORBA interface and Server Factory class. Note that we don't need to add the Type Library itself (from the CORBA server project) to our CORBA client project, but only the import unit. In our example, the BobNotes CORBA server (from Issue 45) was written in Delphi 4 Client/Server, so we can use the BobNotes_TLB.PAS unit and add it to our Delphi 5 CORBA client application.

Inside this import unit, the `TCorBobNotesCorbaFactory` is defined, with a class method `CreateInstance`. Being a class method, we can call it without the need for an instance of the `TCorBobNotesCorbaFactory`, and all we need to pass as argument is the name of the instance, which is `CorBobNotes`. This returns an interface `ICorBobNotes`, which we can use to call the methods of the remote CORBA server, such as the `GetLines` method. Note that since the `ICorBobNotes` interface class is defined in the `BobNotes_TLB` import unit, we get full Code Insight support inside the Delphi code editor, so we can not only see the `GetLines` method but also get help on the number and type of the arguments to this method. Quite handy!

Note that we need to set the `Client` to `nil` again when we're done using it. This is the clean way of working with CORBA servers: use the instance only for as long as you need. If you need it for more than a few method calls, you may want to add a property to your main class, which is initialised and finalised with your class itself.

The serious downside of this 'old' method of connecting Delphi CORBA clients to Delphi CORBA servers is the fact that it only works with Delphi CORBA servers. And with later versions of Delphi, we can restrict this to Delphi CORBA servers that use a Type Library (and hence have a Type Library import unit). This means that as soon as we need to connect to a CORBA server written in another language, even C++Builder, we're out of luck, and can't use this technique.

Dynamic Interface Invocation

Suppose we don't have the luxury of connecting to a Delphi CORBA server, but we must connect to a 'foreign' CORBA server (written in JBuilder, for example, or any programming environment, for that matter). In that case, no Type Library is used, and no Type Library import unit is available either.

Other environments, like JBuilder, can use the built-in IDL compiler, such as IDL2Java, to compile the CORBA IDL file to a native set of Java files. We'll see the same thing in the next section (the third way). For now, we assume a regular Delphi 4 Client/Server or Delphi 5 Enterprise without the VisiBroker for Delphi. And in those situations, we can decide to dynamically connect to the CORBA server using a late binding technique called DII, Dynamic Interface Invocation. Using DII, the Delphi CORBA client will dynamically use the IDL definitions. To be able to do so, the IDL must be stored someplace in a table on the network and the ORB, so the Delphi CORBA client can bind to it and invoke the methods defined by the IDL. The place to store the IDL definitions is called the Interface Repository. We can start the Interface Repository with the following command:

```
start irep drbob BobNotes.IDL
```

Remember that the VisiBroker Smart Agent must be running before we can start the Interface Repository (or any CORBA client, for that matter). Once the Interface Repository is started, we can search for the available IDL definitions, like BobNotes.

Once the VisiBroker Smart Agent and the Interface Repository (with the BobNotes.IDL) are running, we can write the dynamic (or late) binding code. This time, we can't use a Type Library or any other file with type definitions, so we must revert to using `TAny` types instead. We must also add the `CorbaObj` unit to the `uses` clause, which contains the `Orb` object.

➤ *Listing 2:*
Type Library Import Units.

```
uses
  BobNotes_TLB;
procedure TForm1.Button1Click(Sender: TObject);
var
  Client: ICorBobNotes;
  Lines: WideString;
begin
  Client := TCorBobNotesCorbaFactory.CreateInstance('CorBobNotes');
  Client.GetLines('Bob','swart',Lines);
  Memo1.Lines.Add(Lines);
  Client := nil;
end;
```

From the Orb object, we should call the Bind method passing the complete IDL name of the Factory, which is IDL:BobNotes/CorBobNotesFactory:1.0. After we've obtained a handle to the Factory, we can call the CreateInstance method of this Factory object, passing CorBobNotes as argument. Compare this to the code we needed to write for the 'old' technique, and you see that it's very similar.

The big difference is in the fact that both the Factory and Client variables are of type TAny. And while I knew I had to call the CreateInstance method of the Factory, I only tried this because I remembered calling this method when using the 'old' way. Once I have an instance, I just have to know that I can call the GetLines method. I also need to remember the number and types of the arguments (otherwise I will not get a compile error, but a runtime error). To make matters worse, it appears that the method names are case sensitive (or at least they are when calling them from a CORBA server written in JBuilder).

Note that we need to declare two local variables of type WideString (the User and Pass), since passing these arguments by value will generate a runtime error (argument type incorrect). When using value arguments, the Delphi compiler will generate code for regular strings instead of WideStrings. Using variables of type WideString as argument to the TAny method call will make sure the CORBA client starts looking for the correct dynamic method call at runtime!

Also note that this time we need to set the Client and Factory TAny variables to unassigned (since nil isn't compatible with TAny types). The general idea is the same: only hold on to the Factory and Client as long as you need (or use) them.

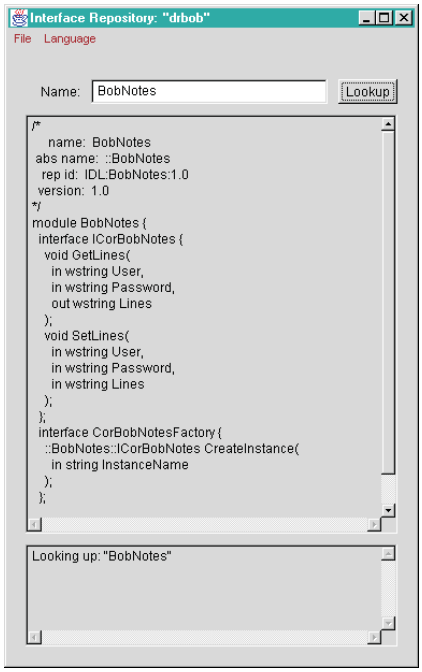
The main disadvantage of this method of calling CORBA servers from Delphi CORBA clients is the fact that we have absolutely no design-time support. No Code Insight, but apart from that any possible error will only be reported at runtime! The upside, of course, is that you only need an IDL file stored in the Interface Repository to be able to talk to a CORBA server (provided it's running). However, there must be something more. Something to talk to non-Delphi CORBA servers while we don't have to give up the design-time and Code Insight support.

➤ *Listing 3: Dynamic Interface Invocation.*

```
uses CorbaObj;
procedure TForm1.Button2Click(Sender: TObject);
var
  Factory,Client: TAny;
  Lines: WideString;
  User,Pass: WideString;
begin
  Factory := Orb.Bind('IDL:BobNotes/CorBobNotesFactory:1.0');
  Client := Factory.CreateInstance('CorBobNotes');
  User := 'Bob';
  Pass := 'swart';
  Client.GetLines(User,Pass,Lines);
  Memo1.Lines.Add(Lines);
  Client := unassigned;
  Factory := unassigned;
end;
```

➤ *Listing 4: IDL2PAS Generated Units.*

```
uses CORBA, OrbPas30, BobNotes_I, BobNotes_C;
procedure TForm1.Button1Click(Sender: TObject);
var
  Factory: CorBobNotesFactory;
  Client: ICorBobNotes;
  Lines: WideString;
begin
  Factory := TCorBobNotesFactoryHelper.Bind('CorBobNotes');
  Client := Factory.CreateInstance('CorBobNotes');
  Client.GetLines('Bob','swart',Lines);
  Memo1.Lines.Add(Lines);
  Client := nil;
  Factory := nil
end;
```



➤ *Figure 2: Interface Repository.*

Well, you guessed it: the best of both worlds now exists, and is...

VisiBroker For Delphi 5

VisiBroker for Delphi uses the VisiBroker C++ ORB (as found in C++Builder 4 Enterprise, for example). Inprise has encapsulated the C++ ORB with a Delphi wrapper, contained in the ORBPAS33.DLL (for which we must add the ORBPAS30 unit to the uses clause).

Given an IDL file, we can run the IDL2PAS batch file to produce the xxx_I.PAS (interface) and xxx_C.PAS (client stub) units. These, together with the new CORBA and the OrbPas30 units, are the only things we need to include to be able to statically invoke a CORBA server written in any language.

Let's consider the Factory, which is of type CorBobNotesFactory (as defined in the BobNotes_C unit). We can obtain a Factory by calling the Bind method of the generated TCorBobNotesFactoryHelper class with the CorBobNotes argument. Once we have the Factory, we can create an instance of the CORBA server in the usual way (as for the other two methods), calling the CreateInstance method with the CorBobNotes string as argument.

If you compare the code from Listings 2, 3 and 4, you'll see that

they have the 'client' code in common. The main difference is the lack of design-time support using DII, and only being able to connect to Delphi CORBA servers (using Type Libraries) for the first solution. Although DII remains the most flexible solution for IDL files where the interface is bound to change often, I prefer the IDL2PAS solution, Listing 4. Especially with the other enhancements that we find in VisiBroker for Delphi.

More Enhancements

Apart from being able to translate IDL files to Object Pascal units, the IDL2PAS compiler has some extra features not present in the 'out-of-the-box' CORBA support in Delphi 5. One of them is support for exceptions, which we'll see in a moment, but a more relevant feature is the support for IDL structures: Object Pascal record types.

We can combine the `User` and `Password` wstrings into a single structure `TUserPass`, defined as follows in IDL:

```
struct UserPass {
  wstring User;
  wstring Pass;
}
```

Unfortunately, we can't test this with a Delphi 5 CORBA server (as only IDL2PAS for Delphi CORBA clients supports these structures), but we can report that it works fine when connecting to a JBuilder CORBA server using the above IDL `struct`. And now that JBuilder for Linux is available, with VisiBroker for Linux, this opens up all kinds of exciting new architectural possibilities for cross-platform cross-language distributed applications. But more on that later...

Exceptions

Error handling in CORBA applications is usually handled by CORBA exceptions. Unfortunately, the previous support for CORBA didn't include support for CORBA exceptions. And this was a major problem, since a real-world CORBA class hierarchy is often accompanied by a set of CORBA exception types as well.

A detailed exploration of CORBA Exceptions is left for a later article (when we'll also get back to some other VisiBroker techniques).

Conclusions

VisiBroker for Delphi adds IDL2PAS for statically linked CORBA clients. It also adds the ability for records and exceptions, which finally brings the CORBA implementation in Delphi to a professional level. I can't wait until IDL2PAS for Delphi CORBA servers becomes available, although I fear we may have to wait until Delphi 6 Enterprise for that. But I can wait. It can only get better, eh? Another feature missing from this release is callbacks.

Finally, please note that the free download of VisiBroker for Delphi provides a *development* licence only. A *deployment* licence (which is available separately, contact your local Inprise office for pricing details) is still required in order to deploy applications built with VisiBroker for Delphi.

Next Time

Next time in the one and only real *The Delphi Magazine*, we examine *internet security*, a topic long overdue, but critically important for any serious internet application (from a simple survey to a blockbuster e-commerce website). We'll see what, why, how and more. All the reason you need *to stay tuned*...

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is an IT Consultant for TAS Advanced Technologies and freelance technical author.

All About CORBA

CORBA stands for **C**ommon **O**bject **R**equest **B**roker **A**rchitecture, and is an object-oriented communication architecture between a client and a server. Communication is handled by the **ORB** (Object Request Broker) and **IIOP** (Internet InterORB Protocol). Using **IDL** (Interface Definition Language) we can specify objects with methods and properties. Methods are like functions that can be called by the client, and will be implemented (serviced) by the server. In order to do so, the IDL file must be compiled to a native programming language for a specific platform. This results in stub code for the client (so we can invoke the methods without worrying about the underlying communication) and skeleton code for the server (which is the basis for our communication on the server side).

CORBA is both platform and language independent. This can only work if the parameters and return types of the methods are transported over the network in a portable format. Conversion from a native type to a portable IDL type is called marshalling, while conversion back to a native platform/language type is called unmarshalling. Usually, the marshalling process is done by an IDL compiler, like IDL2Java or IDL2PAS.