Under Construction: VisiBroker 3.3 For Delphi 5

by Bob Swart

Exactly a year ago, in Issue 54, I wrote about the free VisiBroker 3.3 for Delphi 5 which was then made available as a free download and plug-in for Delphi 5 Enterprise users. It contained a somewhat limited IDL2Pas and support for CORBA exceptions, but client-side only.

One year later, in January 2001, Borland is about to release the final version of VisiBroker 3.3 for Delphi 5, with an enhanced IDL2Pas, including a Wizard, and full support for both client- and server-side CORBA exceptions. In this article, I'll explore some of the enhanced features that the new VisiBroker for Delphi 5 brings us.

No More Type Library

The free VisiBroker 3.3 for Delphi 5 Enterprise that shipped in December 1999 could generate client stubs for CORBA clients, and contained client-side support for CORBA exceptions, but did not have any server-side support. We still needed to use either the Delphi 5 Type Library to create a new CORBA server, or rely on a CORBA server written in some other language (and use the corresponding IDL file to let VisiBroker for Delphi generate the CORBA client stubs).

```
module DrBob42
{
  interface Rates
  {
    float interest_rate();
  };
  interface Account
  {
    float balance();
    float get_rates(in Rates myRates);
  };
  struct AccountError
  {
    float  Balance;
    string ErrorMessage;
  };
  exception AccountException
  {
    AccountError Error;
  };
  interface MyAccount: Account
  {
    void deposit(in float amount);
    void withdraw(in float amount) raises(AccountException);
  };
};
```

➤ *Listing 1*

The best news of the year so far (until the launch of Kylix) is the fact that the new VisiBroker 3.3 for Delphi 5 now also contains full CORBA server support. Indeed, there is no more Type Library, but a full IDL2Pas compiler that takes your IDL (interface definition language) file and turns it into client stubs or server skeletons.

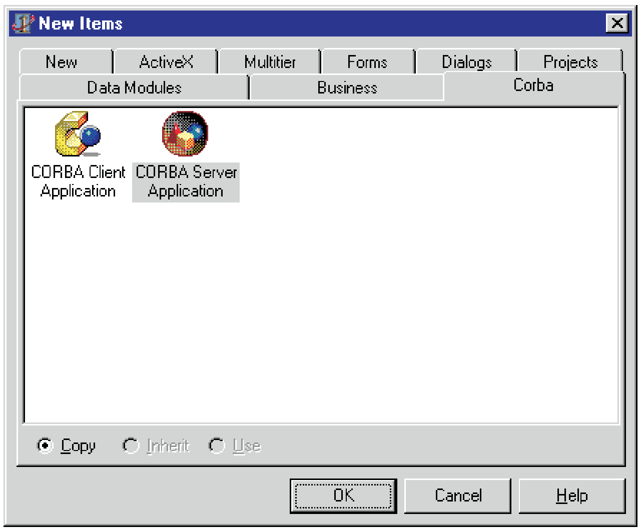
Interface Definition

The IDL file contains the definition of the interface between the CORBA server and the CORBA clients. For this article, I've constructed a somewhat artificial IDL file (Listing 1) that will cover most of the existing and new features and enhancements of VisiBroker 3.3 for Delphi 5.

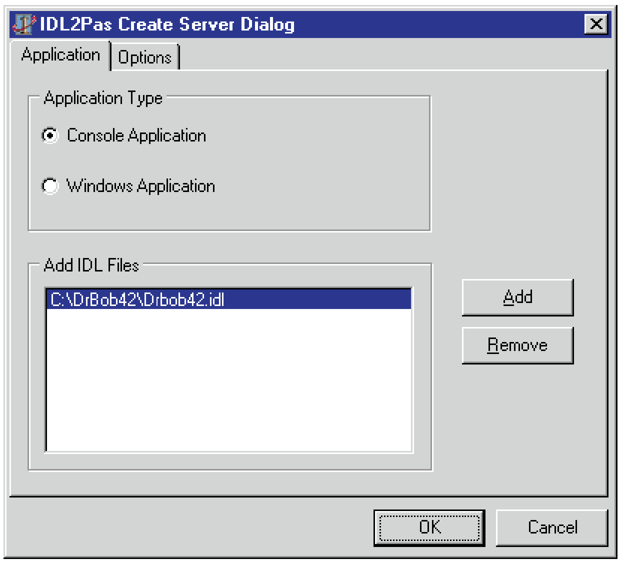
There are more features available in VisiBroker 3.3 for Delphi 5, such as the support for sequences and enumerated types, but these will be covered in an extended version of this article (that will be made available as white paper on the Borland website shortly after you've read this original version).

Let's now describe the meaning of the interfaces defined in the IDL file. First of all, we have an interface called `Rates`, which has one method to return the current `interest_rate`. This is no big deal, but in the second interface we make use of the first interface, by passing it as argument to the `get_rates` method (so the internals of `get_rates` will have to use the `Rates` interface to call the `Rates.interest_rate` method).

The third construct inside the module is a `struct AccountError` with a float to hold the current `Balance` and a string to hold an `ErrorMessage`. This `struct` will clearly be used in an error situation, which is why I've embedded it inside an exception type called `AccountException`, the fourth construct of module `DrBob42`.



➤ *Figure 1: The CORBA tab of the Object Repository.*


IDL2Pas Create Server Dialog

Application | Options

Application Type
- (●) Console Application
- ( ) Windows Application

Add IDL Files
- C:\DrBob42\Drbob42.idl

Add | Remove

OK | Cancel

➤ *Figure 2: Adding DrBob42.idl to the Console CORBA Server Application.*

The fifth and last construct is the most advanced: using interface inheritance and methods (possibly) raising CORBA exceptions. Regarding interface inheritance: it's even possible to use multiple interface inheritance (or interface multiple inheritance, depending on how you look at it), but I always try to avoid multiple inheritance wherever I can, including inside IDL files.

IDL2PAS Wizard

After we've installed 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 Docs\IDL2Pas directory. Finally, check out the Sources\RTL\CORBA directory for a number of new files (including CORBA.PAS and ORBPAS30.PAS). The IDL2Pas.pdf file in the DOCS directory is especially useful and quite interesting to read: it's the VisiBroker for Pascal Reference Guide.

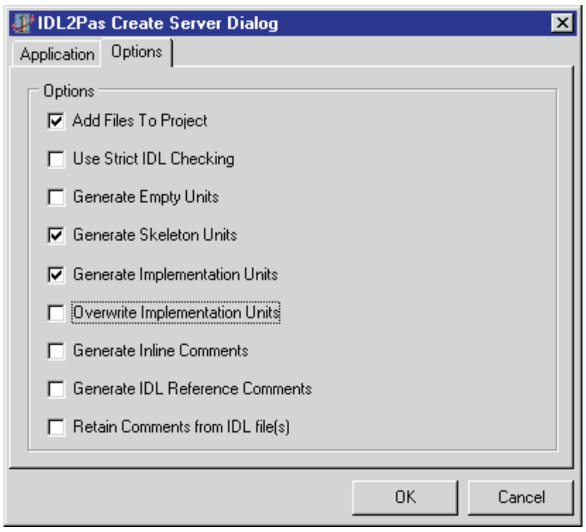
Now then, assuming you've purchased or downloaded or in some other way acquired the new VisiBroker 3.3 for Delphi 5, start your engines (read: Delphi 5 Enterprise) and you'll find a new tab in the Object Repository called `CORBA`. Inside there are two icons for two new project Wizards: one for a CORBA Client Application and one for a CORBA Server Application (see Figure 1).

Since the CORBA Server is usually the place to start, select the CORBA Server Application icon and double-click it, or click on the `OK` button. This will bring up the whole new IDL2Pas Wizard in which you can simply add all the IDL files that need to be part of your CORBA Server.

Note from Figure 2 that we can either select a Console Application or a Windows Application for our CORBA Server. The difference should be obvious, and I've selected a Console Application here (but feel free to start playing with a Windows CORBA Server Application first, if you like). We'll get the same choice (console versus Windows) when we create the CORBA Client, and since you're not limited to just one CORBA Client (or CORBA Server for that matter), the choice is arbitrary: you could create all kinds, as we will see both in this article and the next one.

The `Options` tab of the IDL2Pas Wizard contains a number of helpful options (see Figure 3). They range from adding the generated .pas files to the current project (alternatively you may just want to run IDL2Pas on one or more IDL files to generate the client stubs and server skeletons) to generate the different kinds of output files (skeleton and implementation units) and generating or retaining comments in the generated files.

➤ *Figure 3: IDL2Pas CORBA Server Options.*


IDL2Pas Create Server Dialog

Application | Options

Options
- ☑ Add Files To Project
- ☐ Use Strict IDL Checking
- ☐ Generate Empty Units
- ☑ Generate Skeleton Units
- ☑ Generate Implementation Units
- ☐ Overwrite Implementation Units
- ☐ Generate Inline Comments
- ☐ Generate IDL Reference Comments
- ☐ Retain Comments from IDL file(s)

OK | Cancel

A very helpful and important option is `Overwrite implementation Units` (unchecked in Figure 3). When this option is checked, the implementation units, containing the source code you just wrote with your implementation, will be overwritten when you run IDL2Pas. The first time this happens, you'll lose all your work (in the _impl.pas unit), and you won't be happy. Needless to say, it won't happen a second time to me.

Fortunately, the settings of all these options are saved inside the defproj.dof file in the Delphi5\Bin directory, so you only have to specify your favourite settings once and they'll be the same every time.

When creating Delphi CORBA Clients, the three options for the skeleton and implementation units will be disabled (these are irrelevant for CORBA Clients, of course). Otherwise, the IDL2Pas Create Client Dialog is exactly the same as the IDL2Pas Create Server Dialog.

CORBA Server

Running IDL2Pas on the DrBob42.idl file creates four files: DrBob42_i.pas (with the interface definitions), DrBob42_c.pas (with the client stubs, the code which the client application can use or call), DrBob42_s.pas (with the server skeletons) and finally DrBob42_impl.pas with our implementation of the skeletons. This last file is the one you don't want the IDL2Pas to accidentally overwrite next time it processes the IDL file. The DrBob42_impl.pas file contains the ObjectPascal class definitions for `TRates`, `TAccount` and `TMyAccount` that we need to implement. These are also the three CORBA classes we need to create in the server itself, so the clients can talk to them. Note that `AccountError` and `AccountException` are defined in DrBob42_i.pas and require no further implementation (both are just 'dumb' structures).

Talking about the CORBA Server: apart from the aforementioned four generated files, the IDL2Pas Wizard also generates a new Delphi project, which has the generated files in its `uses` clause, and also contains some comments (examples) to guide you into writing your own CORBA Server initialisation code. I guess it's a bit complex to parse the IDL file, obtain the interface names and automatically generate and insert variable declarations and constructor calls for these CORBA interfaces. Maybe a future version of the IDL2Pas wizard can support this feature (or we can build it ourselves someday).

In our case, we need to change the main project file from the generated example as seen in Figure 4, and make sure instances are created of `Rates`, `Account` and `MyAccount` (which are in fact merely aliases for `TRatesSkeleton`, `TAccountSkeleton` and `TMyAccountSkeleton`). Inside the DrBob42_s.pas file (containing the server skeletons), we see `TRatesSkeleton`, `TAccountSkeleton` and `TMyAccountSkeleton` classes, each with the same constructor `Create` that takes two arguments: the first for an instance name (which can be anything), and the second for an instance of the CORBA interface itself):

```
constructor Create(
  const InstanceName: string;
  const Impl: Rates);
```

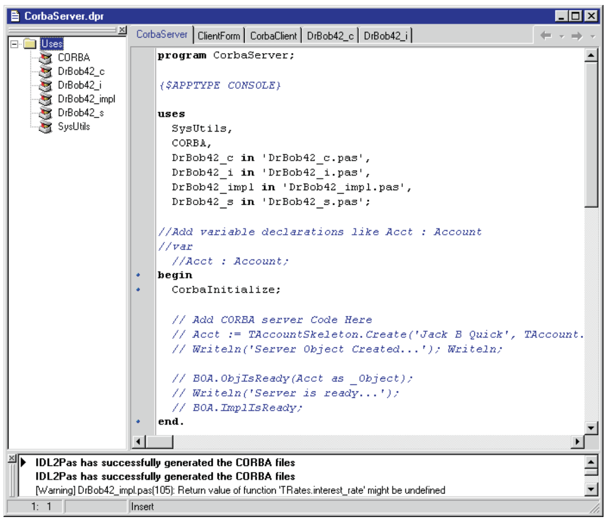
Once all three CORBA classes have been created with help of their skeleton, we need to call the `ObjIsReady` method of the BOA (Basic Object Adaptor) to tell the BOA that this CORBA object is ready to be used by CORBA clients. And finally, once all CORBA Objects have been registered as being ready, we need to call the `ImplIsReady` method of the BOA to tell it that the entire CORBA Server application is ready to go into the 'waiting loop'. This waiting loop means that it looks like the CORBA Server is now hanging, while in fact it is waiting for, receiving and responding to CORBA requests (from CORBA Clients), not unlike the Windows messaging loop we all know. When you terminate the console application, the waiting loop is ended and the CORBA server is closed. For a Windows CORBA application, the call to `BOA.ImplIsReady` is not needed, since the Windows loop itself will make sure the CORBA server can receive and respond to CORBA requests (until the Windows CORBA Server application is closed, of course).

The resulting CORBA Server application for our IDL file can be seen in Listing 2.

Note the cast `as _Object` which is used in the `BOA.ObjIsReady` calls. This is not really needed. In fact, although it was part of the example snippet that was generated, there is no use for it, so I won't be using it again (the compiler will not complain if you compile without the `as _Object` parts, the CORBA Server will still run, and the code looks much clearer without the unnecessary focus on the `as _Object` statements.

Server Skeletons

Now that we've created our CORBA Objects, it's time to actually implement them (otherwise the CORBA Server won't do much good), so let's turn to the DrBob42_impl.pas file. The header of this file, like all four generated files, explains that the file was actually generated by the *Inprise VisiBroker IDL2Pas CORBA IDL compiler* (I wonder if the final



```
program CorbaServer;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  CORBA,
  DrBob42_c in 'DrBob42_c.pas',
  DrBob42_i in 'DrBob42_i.pas',
  DrBob42_impl in 'DrBob42_impl.pas',
  DrBob42_s in 'DrBob42_s.pas';

//Add variable declarations like Acct : Account
//var
  //Acct : Account;
begin
  CorbaInitialize;

  // Add CORBA server Code Here
  // Acct := TAccountSkeleton.Create('Jack B Quick', TAccount.
  // Writeln('Server Object Created...'); Writeln;

  // BOA.ObjIsReady(Acct as _Object);
  // Writeln('Server is ready...');
  // BOA.ImplIsReady;
end.
```

```
IDL2Pas has successfully generated the CORBA files
IDL2Pas has successfully generated the CORBA files
[Warning] DrBob42_impl.pas(105): Return value of function 'TRates.interest_rate' might be undefined
```

➤ *Figure 4: Corba Server generated code in the Delphi 5 IDE.*

➤ *Listing 2: CORBA Console Server Application.*

```
program CorbaServer;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  CORBA,
  DrBob42_c in 'DrBob42_c.pas',
  DrBob42_i in 'DrBob42_i.pas',
  DrBob42_impl in 'DrBob42_impl.pas',
  DrBob42_s in 'DrBob42_s.pas';
var
  // The CORBA server Skeletons
  Rate: Rates;
  Acct: Account;
  MyAcct: MyAccount;
begin
  CorbaInitialize;
  // Add CORBA server Code Here
  writeln('Init');
  Rate := TRatesSkeleton.Create('Rate', TRates.Create);
  writeln('Server Rate Object Created...');
  Acct :=
    TAccountSkeleton.Create('Account', TAccount.Create);
  writeln('Server Account Object Created...');
  MyAcct := TMyAccountSkeleton.Create('MyAccount',
    TMyAccount.Create);
  writeln('Server MyAccount Object Created...');
  writeln;
  BOA.ObjIsReady(Rate as _Object);
  write('And ');
  BOA.ObjIsReady(Acct as _Object);
  write('the ');
  BOA.ObjIsReady(MyAcct as _Object);
  writeln('Server is ready...');
  BOA.ImplIsReady;
end.
```

```
unit DrBob42_impl;
{This file was generated on 29 Dec 2000 10:32:07 GMT by
  version 03.03.03.C1.06 of the Inprise VisiBroker IDL2Pas
  CORBA IDL compiler.}
{Please do not edit the contents of this file. You should
  instead edit and recompile the original IDL which was
  located in the file {C:\DrBob42\Drbob42.idl.}
{Delphi Pascal unit : DrBob42_impl derived from IDL
  module : DrBob42}
interface
uses
  SysUtils,
  CORBA,
  DrBob42_i,
  DrBob42_c;
type
  TRates = class;
  TAccount = class;
  TMyAccount = class;
  TRates = class(TInterfacedObject, DrBob42_i.Rates)
  protected
    finterest_rate: Single;
  public
    constructor Create;
    function interest_rate: Single;
  end;
  TAccount = class(TInterfacedObject, DrBob42_i.Account)
  protected
    fbalance: Single;
  public
    constructor Create;
    function balance: Single;
    function get_rates(const myRates: DrBob42_i.Rates):
      Single;
  end;
  TMyAccount = class(TInterfacedObject, DrBob42_i.MyAccount)
  protected
    fbalance: Single;
  public
    constructor Create;
    procedure deposit(const amount: Single);
    procedure withdraw(const amount: Single);
    function balance: Single;
    function get_rates(const myRates: DrBob42_i.Rates):
      Single;
  end;
implementation
uses Dialogs;
constructor TRates.Create;
begin
  inherited;
  finterest_rate := 7; // seems like a nice interest rate
  ShowMessage('TRates.Create');
end;
function  TRates.interest_rate: Single;
begin
  Result := finterest_rate;
end;
constructor TAccount.Create;
begin
  inherited;
  fbalance := 0; // balance starts empty
  ShowMessage('TAccount.Create');
end;
function  TAccount.balance: Single;
begin
  Result := fbalance;
end;
function  TAccount.get_rates(const myRates:
  DrBob42_i.Rates): Single;
begin
  Result := myRates.interest_rate
end;
constructor TMyAccount.Create;
begin
  inherited;
  fbalance := 0;
  ShowMessage('TMyAccount.Create');
end;
procedure TMyAccount.deposit(const amount: Single);
begin
  fbalance := fbalance + amount;
end;
procedure TMyAccount.withdraw(const amount: Single);
begin
  fbalance := fbalance - amount;
end;
function  TMyAccount.balance: Single;
begin
  Result := fbalance;
end;
function  TMyAccount.get_rates(const myRates:
  DrBob42_i.Rates): Single;
begin
  Result := myRates.interest_rate
end;
initialization
end.
```

➤ *Listing 3: CORBA Server Skeleton Implementation.*

product will contain Borland instead of Inprise in the name). All the generated files also contain a warning that says *'Please do not edit the contents of this file. You should instead edit and recompile the original IDL file'* including the location of that IDL file. Confusingly, this warning also appears in the DrBob42_impl.pas file, the one, you guessed it, we *need* to modify to include our implementation. Oops!

Fortunately, DrBob42_impl.pas also contains several cues to tell us to insert user variables and user code at the right places. Once all these commented cues have been replaced by actual code, your implementation is probably complete as well. If we just store the `interest_rate` and balance in shared properties (instead of retrieving them from a real database, for example), then our minimum CORBA Skeleton implementation can be seen in Listing 3.

Please note that this is a simple implementation, with no consideration of multi-threading issues (when more than one CORBA client is connected to the same CORBA server, each talking with the same global account).

Note that the `Create` constructors in Listing 3 all contain a `ShowMessage` statement that will tell you, when you start the Server, that this CORBA skeleton object is indeed created. This might help you pinpoint a problem when one of your objects raises exceptions or experiences other problems.

CORBA Exceptions

Talking about exceptions, I didn't add the `AccountError` structure and `AccountException` type just for fun: I want to use them as well, of course. The obvious place to raise an `AccountException` is inside the withdraw method of the `MyAccount` interface (and if you look closely at the IDL file, you also see that that's the *only* place where we can raise that exception). If the balance is (still) empty, no money can be withdrawn. And you should also get an error if you try to withdraw more money than is currently in your account (although a real bank would probably only show you a warning and charge you interest rates instead).

We need to create an exception, and assign a value to its field `Error` of type `AccountError`. The easiest way to do this is to pass the initial values as argument to the constructor of the `TAccountError` class (which constructs the `TAccountError` structure). The complete code can be seen in the new version of the `TMyAccount`.withdraw method (which starts by checking the fact that the amount to withdraw cannot be negative), see Listing 4.

Note that this example combines the server-side structures technique with server-side exceptions (previously impossible, even with the earlier version of VisiBroker 3.3 for Delphi 5 that shipped in 1999). And also note that since the CORBA Server is a console application, I can simply use `writeln` statements to report an error in the CORBA server output window.

CORBA Client

Now that we've created the CORBA Server project and implemented the Server skeleton, it's time to focus on the CORBA client application. The framework is again generated by the IDL2Pas Wizard, but this time we need to look at the interfaces, as defined in DrBob42_i.pas, and use the client stubs, as available in DrBob42_c.pas.

Because we generated a Windows CORBA Client application, we get a main form, and must perform some special CORBA initialization before doing anything else. We can either insert a call to `CorbaInitialize` in the main project source code, or make sure this routine is called in the `OnCreate` event of the main form. I'll use the latter technique here, so I won't have to bother you with the CORBA client main project file. In fact, if you call `CorbaInitialize` in the `OnCreate` event of your main form, then you don't even have to include the generated `DrBob42_i` and `DrBob42_c` units in the `uses` clause of the CORBA client project file. Of course, the consequence is that we need to add these units to the Client main form, but a comment to tell you that is already generated in the main form unit by the IDL2Pas wizard itself. The IDL2Pas wizard has also added a special method called `InitCorba` to the `Form` class in the main form unit. The `InitCorba` routine contains the call to `CorbaInitialize`, but could also be used to create (global) instances of the CORBA server objects, as I've done in Listing 5.

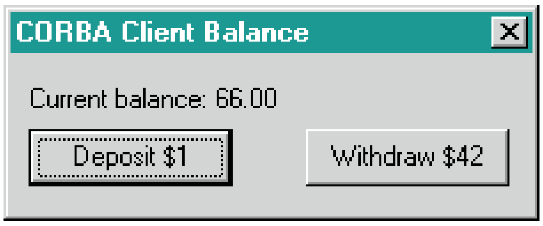
Note that we do not explicitly have to destroy the CORBA objects (and that the objects themselves again are the `Rates`, `Account` and `MyAccount` types that are just aliases for the Server skeleton types, but this time called the client stubs).

Using Client Stubs

It's nice that our client form creates the CORBA objects in the `OnCreate` event, but this would not be very useful if we didn't use the CORBA objects in some way. So, I've added two buttons to the client form: one to deposit one dollar to the `MyAccount` object and another to (try to) withdraw 42 dollars from that account. Finally, I have added a label that will display the current balance of `MyAccount` after each of the two buttons have been pressed (and of course the corresponding `MyAccount` CORBA server method has been executed).

The `OnClick` events from the two buttons can be seen in Listing 6. Note the fact that we can actually use the `EAccountException` type, which holds the field called `Error` of type `AccountError` with two fields called `Message` (the error message) and `Account` (the value of balance or the amount, used in the error message).

➤ *Figure 5: CORBA Client Form.*



➤ *Listing 4: New TMyAccount.withdraw Implementation.*

```
procedure TMyAccount.withdraw(const amount: Single);
var
  Error: TAccountError;
begin
  if amount <= 0 then begin
    writeln('Cannot withdraw negative amount ',amount:1:2);
    Error := TAccountError.Create(amount,'Cannot withdraw negative amount %f');
    raise EAccountException.Create(Error);
  end else if fbalance <= 0 then begin
    writeln('Balance zero or negative: ',fbalance:1:2);
    Error := TAccountError.Create(fbalance,'Balance zero or negative: %f');
    raise EAccountException.Create(Error);
  end else if amount > fbalance then begin
    writeln('Balance not enough: ',fbalance:1:2);
    Error := TAccountError.Create(fbalance,'Balance not enough: %f');
    raise EAccountException.Create(Error);
  end else
    fbalance := fbalance - amount;
end;
```

➤ *Listing 5: CORBA Client Main Form Implementation.*

```
unit ClientForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Corba, DrBob42_i, DrBob42_c;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    Rate: Rates;
    Acct: Account;
    MyAcct: MyAccount;
  protected
    procedure InitCorba;
  public
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.InitCorba;
begin
  CorbaInitialize;
  Rate := TRatesHelper.Bind;
  Acct := TAccountHelper.Bind;
  MyAcct := TMyAccountHelper.Bind;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  InitCorba;
end;
end.
```

It is important that we make sure that the `MyAcct` variable is indeed pointing to a valid CORBA object. If the initialisation (done with the `MyAccountHelper.Bind` function) failed, then `MyAcct` will still be `nil`, which is why I usually either disable all subsequent action buttons in the `OnCreate` method, or explicitly include an assert in the `OnClick` methods of the buttons themselves (as can be seen in Listing 6).

The final example that you can see in Listing 6 is based on a combination of interface inheritance (the fact that `MyAccount` inherits from `Account`) and passing interfaces as arguments (the fact that we can pass `Rate` as argument to the `MyAcct.get_rates` method).

```
procedure TForm1.ButtonDepositClick(Sender: TObject);
begin
  Assert(MyAcct <> nil,'No connection to CORBA Server');
  MyAcct.deposit(1);
  LabelBalance.Caption := Format('Current balance: %f (%f%%)',
    [MyAcct.balance,MyAcct.get_rates(Rate)])
end;
procedure TForm1.ButtonWithdrawClick(Sender: TObject);
begin
  Assert(MyAcct <> nil,'No connection to CORBA Server');
  try
    try
      MyAcct.withdraw(42);
    except
      on E: EAccountException do
        ShowMessage(Format(E.Error.ErrorMessage,[E.Error.Balance]))
    end
  finally
    LabelBalance.Caption := Format('Current balance: %f (%f%%)',
      [MyAcct.balance,MyAcct.get_rates(Rate)])
  end;
end;
```

➤ *Listing 6: OnClick Implementations.*

Action!

As usual, before you can start the CORBA client, you must first make sure the CORBA server is running. And before you can run the CORBA server, you must make sure that the VisiBroker Smart Agent is running (at least somewhere on the IP-subnet).

Note that VisiBroker 3.3 for Delphi 5 contains a developer licence to develop and test all this, but not a deployment license. So when you're ready to install and deploy your CORBA application in the field, you need to contact your local Borland office and enquire about purchasing a VisiBroker licence.

Further VisiBroker Enhancements

In this article, we've seen examples of using the IDL2Pas to generate both Server Skeletons and Client Stubs. We've implemented the Server Skeletons, and used IDL features like interface inheritance, interfaces passed as arguments, IDL structures and server-side exceptions. Next month, we'll turn the tables and write a CORBA Windows Server application and console Client application. We'll also experiment with some advanced features that have been left out of the column this month, such as callbacks, enumerated types, sequences, arrays, and more, so stay tuned!

And if you are interested in using CORBA with Delphi 5, then I can only urge you to start working with the new VisiBroker 3.3 for Delphi 5: CORBA support in Delphi 5 the way it should have been from the start!

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is an IT Consultant for the Everest Delphi OplossingsCentrum and a freelance technical author.

CORBA

CORBA, the **C**ommon **O**bject **R**equest **B**roker **A**rchitecture, is a multi-tier communication protocol. In other words, using CORBA, two or more applications (or tiers) can communicate with each other. Usually, a CORBA architecture defines a CORBA server and CORBA clients (that communicate with the server). The main advantage of CORBA over, for example, COM or pure Java, is the fact that CORBA is cross-platform (unlike COM) and cross-language (unlike Java). Specifically, a CORBA client on Windows can communicate with a CORBA server on Linux or even on a mainframe. The only way in which CORBA can be cross-platform and cross-language is by making sure that the interface specifications (the 'contact' between the client and the server) is defined in a special uniform language, called Interface Definition Language (IDL). In IDL, you can define modules, interfaces, methods and much more (as you can see in this article). The interface, defined in the IDL file, must be compiled to a native representation for both the client and the server, resulting in server skeleton files (that need to implement the methods) and client stubs (that can call the methods). There have been IDL2Cpp and IDL2Java compilers for years, and the first IDL2Pas from Borland shipped in December 1999 but contained mainly client-side CORBA support. Now, with the new IDL2Pas we finally have full client and server support for CORBA in Delphi 5 Enterprise.