Borland

VisiBroker® 3.3 for Delphi™ 5

By Bob Swart

Table of Contents

Introduction

When Delphi™ 5 Enterprise shipped, it had limited support for CORBA® (no IDL2Pas compiler for example). In December 1999, VisiBroker® 3.3 for Delphi 5 was made available as a free download and plug-in for Delphi 5 Enterprise. It contained a somewhat limited IDL2Pas and support for CORBA exceptions, only on the client's side.

Now, one year later, Borland has released the final version of VisiBroker 3.3 for Delphi 5. It includes an enhanced IDL2Pas, including a Wizard, and full support for both client and server-side CORBA exceptions. In this white paper, I'll explore the enhanced features that the new VisiBroker for Delphi 5 brings us, and show how CORBA is done the Delphi way…

VisiBroker®

white paper

CORBA®

The **C**ommon **O**bject **R**equest **B**roker **A**rchitecture, simply called CORBA, 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 is, for example, COM or 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 a mainframe. The 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) are defined in a special uniform language, called Interface Definition Language (IDL). In IDL, you can define modules, interfaces, methods and much more (as you will see in this white paper). 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 (needed 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, Borland finally has full client and server support for CORBA in Delphi 5 Enterprise.

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 another language (and use the corresponding IDL file to let VisiBroker for Delphi generate the CORBA client stubs).

The best news of the year so far (until the launch of Kylix™, the first native Rapid Application Development (RAD) environment for the Linux operating system (OS)). is the fact that the new VisiBroker 3.3 for Delphi 5 now also contains full CORBA server support. No more Type Library, but a full functional 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 interface definition between the CORBA server and the CORBA clients. For this white paper, I've constructed a somewhat artificial IDL file that will cover most of the existing and new features and enhancements of VisiBroker 3.3 for Delphi 5. The following IDL file uses interfaces, methods, interface inheritance, structs, exceptions, sequences, enumerated types, etc.

```
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);
  };
```

```
typedef string Identifier;

enum EnumType
{
  first,
  second,
  third
};

struct StructType
{
  short age;
  long l;
  Identifier name;
};

union UnionType switch (long)
{
  case -1: short age;
  case  0: long l;
  case  1: Identifier i;
};

const unsigned long ArraySize = 3;

typedef StructType StructArray[ArraySize];

typedef     sequence<StructType>
StructSequence;

interface ADT
{
  void test(in Identifier one, in EnumType
two, in StructType three,
     in  UnionType  four,  in  StructArray
five, in StructSequence six);
};
};
```

Let's now describe the meaning of the interfaces defined in the IDL file. First, 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 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.

The fifth construct is the most advanced: using interface inheritance and methods (possibly) raising CORBA exceptions. Regarding interface inheritance: it's 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.

After the interface inheritance example, the remainder of the IDL file contains the several data type definitions that are supported by IDL2Pas: from simple typedefs, enumerated types, structs and unions to arrays and sequences. And of course, the interface ADT which declares one method test that handles all six Array Data Types (ADT) of the data types defined previously. In short: IDL2Pas is capable of handling just about anything you can imagine or may need to define in your interface definition.

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 (CORBA.PAS and ORBPAS30.PAS), especially the IDL2Pas.pdf file in the DOCS directory (the VisiBroker for Pascal Reference Guide); which is quite interesting to read.

Now then, assuming you've purchased,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).

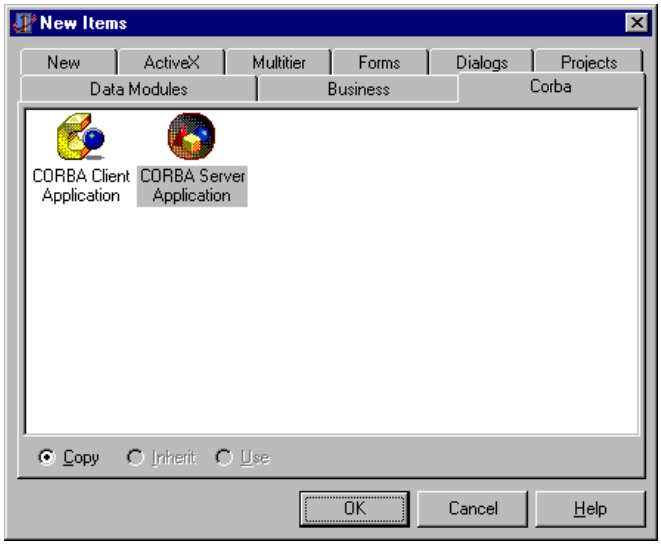
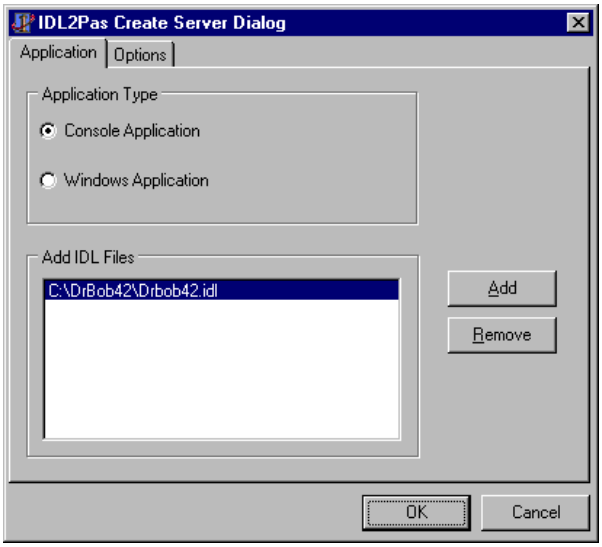


Figure 1. The CORBA tab of the Object Repository.

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

Note from Figure 2 that we can either select a Console 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 CORBA Server Application for Windows first, if you like). We'll get the same choice (console vs. 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'll see in this white paper.



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

A very helpful and important option is the "Overwrite implementation Units" option (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. This is not usually what you want to have happen (or at least you should be conscious of that fact that it will happen if you've selected this option).

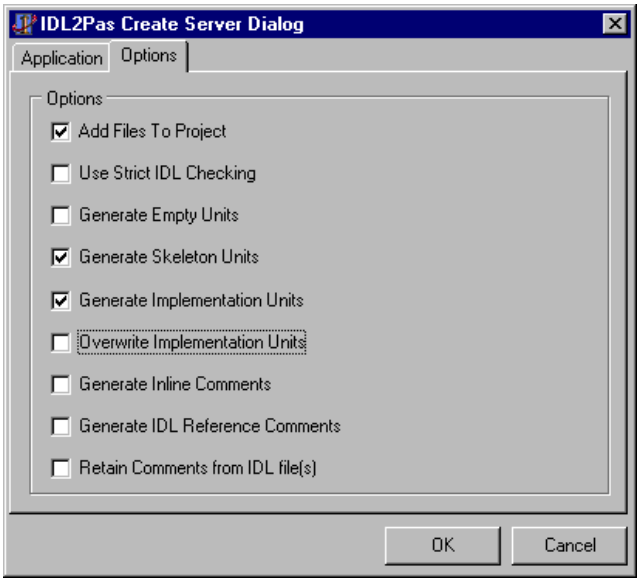


Figure 3. IDL2Pas CORBA Server Options

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 favorite settings once, and they'll be the same every time.

When creating CORBA Delphi Clients, the three options concerning the skeleton and implementation units will be disabled (these are irrelevant for CORBA Clients, of course), as you can see in Figure 6. Otherwise, the IDL2Pas Create Client Dialog is exactly the same as the IDL2Pas Create Server Dialog. Before we start on the CORBA client, let's work on the CORBA server first.

CORBA® Server

The result of running IDL2Pas on the DrBob42.idl file consists of four files: DrBob42_i.pas (with the interface definitions), DrBob42_c.pas (with the client stubs - the code the client application can use/call), DrBob42_s.pas (with the server skeletons) and finally, DrBob42_impl.pas with the implementation of the skeletons by us. And this last file is the one you usually don't want the IDL2Pas to accidentally overwrite the next time it processes the IDL file. The DrBob42_impl.pas file contains the ObjectPascal class definitions for TRates, TAccount, TMyAccount and TADT that we need to implement.

These are also the four 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).

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 initialization code.

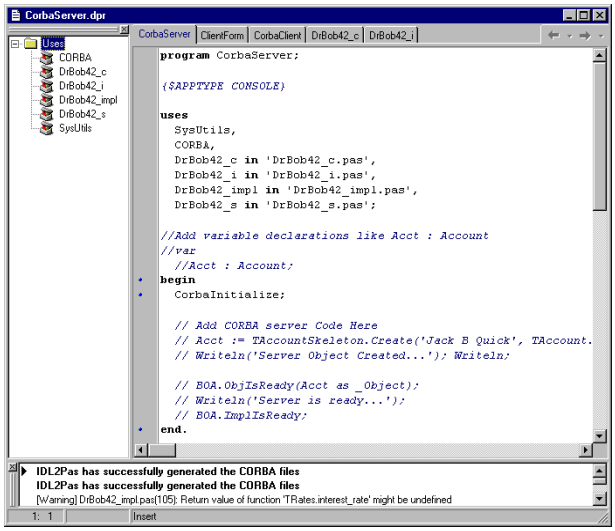


Figure 4. CorbaServer generated code in the Delphi 5 IDE

In our case, we need to change the main project file from the generated example as seen in Figure 4, and make sure that instances are created of Rates, Account, MyAccount and ADT (which are, in fact, merely aliases for TRatesSkeleton, TAccountSkeleton, TMyAccountSkeleton and TADT). Inside the DrBob42_s.pas file (containing the server skeletons), we see TRatesSkeleton, TAccountSkeleton, TMyAccountSkeleton and TADT 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 four 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. 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 1.

```
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;
  ADT: TADT;

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...');
  ADT := TADT.Create('ADT', TADT.Create);
  writeln('Server ADT Object Created...');
  writeln;

  // Make Objects Ready
  BOA.ObjIsReady(Rate as _Object);
  write('And ');
  BOA.ObjIsReady(Acct as _Object);
  write('the ');
  BOA.ObjIsReady(MyAcct as _Object);
  writeln('Server ');
  BOA.ObjIsReady(ADT as _Object);
  writeln('is ready...');

  BOA.ImplIsReady;
end.
```

Listing 1. CORBA Console Server Application

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" (The final product will contain Borland, instead of Inprise, in the name). All 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, the DrBob42_impl.pas file also contains several cues to tell us to insert User variables and User code at the right places. Once all these commented cues have disappeared (and been replaced by actual code), your implementation is probably completed as well. If we

store the interest_rate and balance in shared properties (instead of retrieving them from a real database, for example), and leave the TADT implementation untouched, then our minimum CORBA Skeleton implementation can be seen in Listing 2.
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).

```
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;
  TADT = 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;

  TADT = class(TInterfacedObject,
DrBob42_i.ADT)
  protected
    {*****************************}
    {*** User variables go here ***}
    {*****************************}
  public
    constructor Create;
    procedure test(const one:
DrBob42_i.Identifier;
                   const two:
DrBob42_i.EnumType;
                   const three:
DrBob42_i.StructType;
                   const four:
DrBob42_i.UnionType;
                   const five:
DrBob42_i.StructArray;
                   const six:
DrBob42_i.StructSequence);
  end;

  TSeqAccount = class(TInterfacedObject,
DrBob42_i.SeqAccount)
  protected
    {*****************************}
    {*** User variables go here ***}
    {*****************************}
  public
    constructor Create;
    function  balance(const mySeq:
DrBob42_i.IntSeq): 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;


  constructor TADT.Create;
  begin
    inherited;
    { ************************** }
    { *** User code goes here *** }
    { ************************** }
  end;

  procedure TADT.test(const one:
  DrBob42_i.Identifier;
                      const two:
  DrBob42_i.EnumType;
                      const three:
  DrBob42_i.StructType;
                      const four:
  DrBob42_i.UnionType;
                      const five:
  DrBob42_i.StructArray;
                      const six:
  DrBob42_i.StructSequence);
  begin
    { ************************** }
    { *** User code goes here *** }
    { ************************** }
  end;

  initialization

  end.
```

Listing 2. CORBA Server Skeleton Implementation

Note that the Create constructors in Listing 2 all contain a ShowMessage statement that will tell you - when you start the server - that this CORBA skeleton object has indeed been created. This might help pin-point a problem when one of your objects raises exceptions or experiences other problems.

CORBA® Exceptions

About exceptions: I didn't add the Account Error structure and AccountException type just for fun; I want to use them as well. 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 with interest 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 3.

```
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 neg. 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 3. New TMyAccount.withdraw Implementation

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). Also note that since the CORBA Server is a CONSOLE application, I can simply use written 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. First, we need to start the IDL2Pas Wizard again, but this time, the CORBA Client Application wizard. It will look almost identical to the CORBA Server Application wizard (you need to look at the caption to see the difference). We obviously need to select the same DrBob42.idl file for which we can generate the client stubs for our CORBA client.

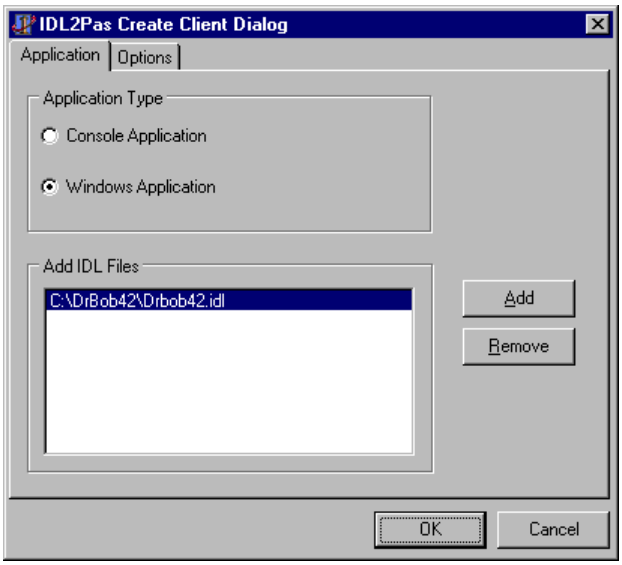


Figure 5. Adding DrBob42.idl to the Console CORBA Client Application

The difference between the CORBA Server Application and Client Application wizard becomes clear in the Options tab of the Wizard: the server skeleton options are disabled (there won't be any server skeleton code generated anyway). Other than that, the same options are available to check or uncheck.

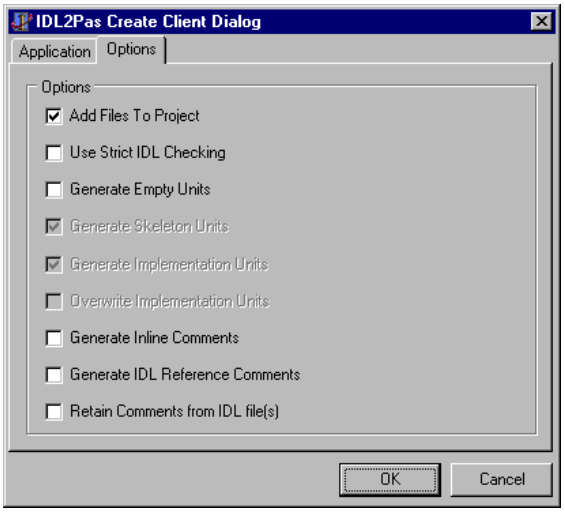
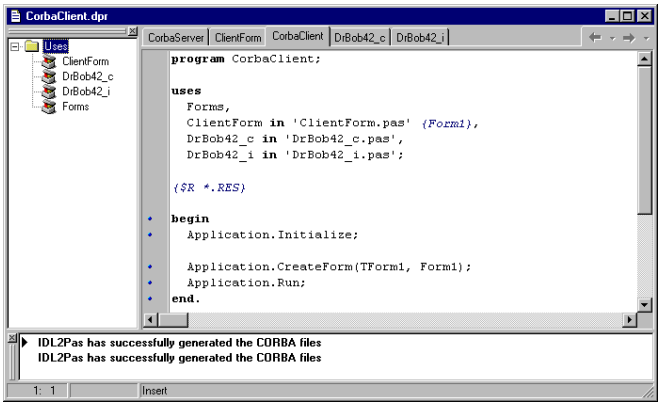


Figure 6. IDL2Pas CORBA Client Options

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, 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. 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 4.

```
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
  { private declarations }
    Rate: Rates;
    Acct: Account;
    MyAcct: MyAccount;
  protected
  { protected declarations }
    procedure InitCorba;
  public
  { public declarations }
  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.
```

Listing 4. CORBA Client Main Form Implementation

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).

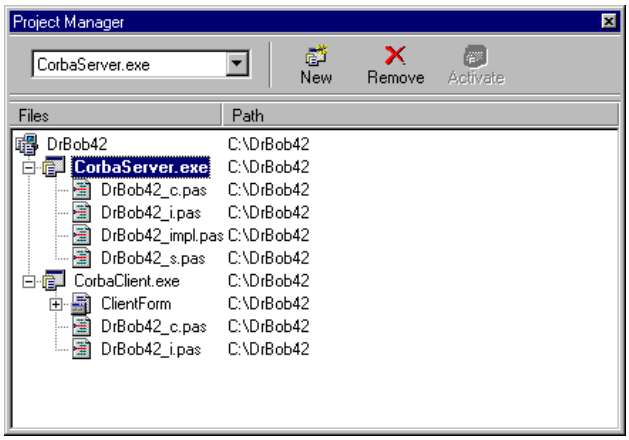


Figure 8. Project Manager for both CORBA Server and Client

Note that it is convenient to use the Project Manager to quickly navigate between the CORBA Server and CORBA Client projects (and also note how they share the same DrBob42_i.pas and DrBob42_c.pas files).

Using Client Stubs

It's nice that our client form creates the CORBA objects in the OnCreate event, but this wouldn't 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 one to (try to) withdraw 42 dollars from that account. Finally, I've added a label that will display the current balance of MyAccount after each of the two buttons have been pressed (and the corresponding MyAccount CORBA server method has been executed).

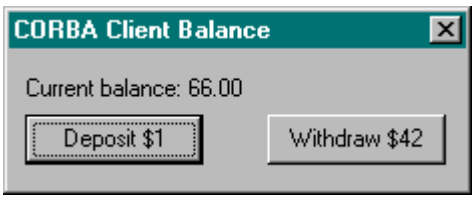


Figure 9. CORBA Client Form in Action

The OnClick events from the two buttons can be seen in Listing 5. Note 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).

```
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.E
rror.Balance]))
    end
  finally
    LabelBalance.Caption :=
      Format('Current balance: %f (%f%%)',

[MyAcct.balance,MyAcct.get_rates(Rate)])
  end;
end;
```

Listing 5. OnClick Implementations

It is important that we make sure that the MyAcct variable is indeed pointing to a valid CORBA object. If the initialization (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 5).

The next example that you can see in Listing 5 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).

The final example in this white paper is using the special structures, enumerated types, arrays and sequences that are passed to the test method of the ADT interface. The client needs to create the arguments and then just pass them as arguments to the ADT CORBA object (see listing 6):

```
procedure FinalExample;
var
  SA: StructArray;
  SS: StructSequence;
  UT: UnionType;
begin
  SA[0] := TStructType.Create(36, 0,
'Robert');
  SA[1] := TStructType.Create(36, 1,
'Erik');
  SA[2] := TStructType.Create(36, 2,
'Swart');

  SetLength(SS, 3);
  SS[0] := TStructType.Create(35, 0,
'Yvonne');
  SS[1] := TStructType.Create(6, 1, 'Erik');
  SS[2] := TStructType.Create(4, 2,
'Tasha');

  UT := TUnionType.Create;
  UT.i := 'Union'; { or UT.age := 42; }

  ADT.test('Hi', second,
TStructType.Create(1, 2, 'DrBob42'), UT, SA,
SS);
end;
```

Listing 6. ADT Example

We didn't actually do anything with the passed parameters inside the test method (see listing 2), but since you get full code insight support, it's easy to actually get your hands on the properties and fields inside the passed structures (so I leave that as an excercise for the reader).

VisiBroker 3.3 for Delphi 5 comes with many more snippets and code examples in the DEMOS directory, and it pays to examine them all to get a feeling (and example code) of how things are working.

Action!

Before you can start the CORBA client, you must first make sure the CORBA server is running. 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 license to develop and test all this, but not a deployment license. When you're ready to install and deploy your CORBA application "in the field", you need to contact your local Borland office and inquire about a VisiBroker license.

Further VisiBroker Enhancements

In this white paper, 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.

If you're interested in CORBA with Delphi 5, then I 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 - www.drbob42.com) is an IT Consultant for the Everest Delphi OplossingsCentrum (DOC) a PinkRoccade nv Company, and has spoken at Borland Conferences since 1993. He is a free-lance technical author for The Delphi Magazine and UK-BUG Developer's Magazine, Delphi Developer and has written chapters for The Revolutionary Guide to Delphi 2 (WROX), Delphi 4 Unleashed, C++Builder 4 Unleashed, and C++Builder 5 Developer's Guide (SAMS).

This white paper is an extended version of an article which originally appeared in The Delphi Magazine (www.TheDelphiMagazine.com) and is used with the permission of the publishers, iTec.

Borland

100 Enterprise Way
Scotts Valley, CA 95066-3249
www.borland.com | 831-431-1000