Under Construction: VisiBroker 3.3 For Delphi 5, Part 2

by Bob Swart

Last month, I started my coverage of the new VisiBroker 3.3 for Delphi 5, which should be available for purchase by the time you read this. There were a few things that I didn't show last time, like support for special type constructs (`enums`, `unions` and `sequences`) and CORBA callbacks, that will be on the agenda for this second article about VisiBroker 3.3 for Delphi 5.

Interface Definitions

The IDL file contains the interface definition between the CORBA server and the CORBA clients. Last month, I constructed a somewhat artificial IDL file that covered most of the existing and new features and enhancements of VisiBroker 3.3 for Delphi 5, except for the support for sequences and enumerated types, which are the topic of the first half of this article. As an intermediate language, IDL is both cross-platform and cross-language. And yet IDL contains a lot of expression power, which means the IDL2*xxx* native language compilers (where *xxx* stands for your language of choice, such as C++, Java or, in our case, Pascal) must be resourceful enough to translate the IDL structure (see Listing 1) to a useful ObjectPascal structure.

Let's take a look at the interfaces and types defined in the IDL file. We already saw the interface `Rates` from last time, only this time I've added a method `SetRate` to allow us to change the interest rate (which will be useful for our callbacks example later in this article). The `Rates` interface is followed by a simple `typedef`. The `enum` type `AccountType` has two possible values: `normal` and `saving`, to be used in the `union` type definition `NormalOrSavingAccount` that follows later. In between these two, I've defined some regular types called `NormalAccount` (with only a balance) and `SavingAccount` (with a balance and an interest rate, obtained from the interface).

Apart from these special types, we can also define arrays or sequences (arrays with unspecified number of items). These can be found in the `AccountArray`, which is an array of three `NormalOrSavingAccount` items, and the `AccountSequence`, which is an unspecified sequence of `NormalOr-SavingAccount` items.

A question that may have come up now is 'how do I pass these types as arguments?'. Well, I'm glad you asked, since the last interface `Accounts` is especially constructed to show two methods that receive `AccountArray` and `AccountSequence` arguments.

IDL2Pas

Enough about the IDL file. Let's turn it into server skeletons and client stubs. Last time, we created the CORBA server as a console application (with the CORBA client as GUI application), and I promised they would trade places this time, so we'll use the IDL file from Listing 1 to create a CORBA server Windows application. This time, the CORBA server doesn't end in a waiting message loop (see last month's article), but should be started in an event handler. The code generated by IDL2Pas contains a main form with a method called `InitCorba`. This method already contains some example code to guide us in writing the correct code to create an instance of our CORBA server.

In this case, we have two interfaces defined (`Rates` and `Accounts`), so we need to add two fields (of type Rates and Accounts) and create two CORBA server objects here. As the comments in the `uses` clause of the main form indicate, we should also (by hand) add the generated `DrBob42_i`, `DrBob42_c` and `DrBob42_impl` units to the `uses` clause of the `interface` section.

The modified code of `InitCorba` and `Unit1` can be seen in Listing 2.

The last remaining step is calling the `InitCorba` method when the form is created (for example, in the `OnCreate` event handler, which also means that we can free the CORBA objects in the `OnDestroy` event handler).

➤ *Listing 1: New DrBob42.idl.*

```
module DrBob42
{
  interface Rates
  {
    float interest_rate();
    void SetRate(in float rate);
  };
  typedef float Money;
  enum AccountType
  {
    normal,
    saving
  };
  struct NormalAccount
  {
    Money balance;
  };
  struct SavingAccount
  {
    Money balance;
    Rates rates; // interface
  };
  union NormalOrSavingAccount switch (AccountType)
  {
    case normal:
      NormalAccount accountN;
    case saving:
      SavingAccount accountS;
  };
  const unsigned long ArraySize = 3;
  typedef NormalOrSavingAccount AccountArray[ArraySize];
  typedef sequence<NormalOrSavingAccount> AccountSequence;
  interface Accounts
  {
    void AccountArrayTest(in AccountArray Accounts);
    void AccountSequenceTest(in AccountSequence Accounts);
  };
};
```

CORBA Client

To test the CORBA server, we need to create a new project using the CORBA client application (a handy tip: put them both in the same project group). Last time we made a CORBA client Windows application, so this time I'm showing how to make a CORBA client console application. The console application should create instances to the CORBA server Accounts first, which is done as in Listing 3.

Note the last two source lines (in comments) that show how we plan to call the `Account.AccountArrayTest` and `Account.AccountSequenceTest` methods. We can find the definition of the `AccountArray` and `AccountSequence` in the DrBob42_i.pas interface unit:

```
AccountArray = array[0..2] of
  DrBob42_i.NormalOrSavingAccount;
AccountSequence = array of
  DrBob42_i.NormalOrSavingAccount;
```

As you see, the sequence is similar to an open array in ObjectPascal. If we start with the `AccountArray`, we need to create an instance of the `TNormalOrSavingAccount` union for every item in the array (ie we need to do that three times, from 0 to 2). We then need to actually decide whether to treat it as a `normal` or `saving` account, but we can't do that by assigning something to the `_discriminator` field (which is read-only). Instead, we just have to either assign a value to the `accountN` or to the `accountS` properties of the `NormalOrSaving Account` item. Assigning a value to either the `accountN` or the `accountS` property has a beneficial side-effect: the `_discriminator` field is correctly set as well.

The example source code in Listing 4 fills the three items in the `AccountArray`, making two normal accounts and one savings account.

Note that the constructor of the `TSavingAccount` needs the `Rates` interface as the second argument. But since this is a global CORBA object anyway, we can just pass the instance of `Rate`.

Considering the code in Listing 4, it shouldn't be much of a surprise that the code for a sequence looks a bit similar (see Listing 5). The main difference is that we need to make a call to `SetLength` to initialise the open array.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, Corba, DrBob42_i, DrBob42_s, DrBob42_impl;
type
  TForm1 = class(TForm)
  private
  protected
    Rate: Rates; // skeleton object
    Account: Accounts; // skeleton object
    procedure InitCorba;
  public
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.InitCorba;
begin
  CorbaInitialize;
  // Add CORBA server Code Here
  Rate := TRatesSkeleton.Create('Rates', TRates.Create);
  BOA.ObjIsReady(Rate as _Object);
  Account := TAccountsSkeleton.Create('Accounts', TAccounts.Create);
  BOA.ObjIsReady(Account as _Object)
end;
end.
```

➤ *Listing 2: CORBA server initialisation.*

```
program CClient;
{$APPTYPE CONSOLE}
uses
  SysUtils, CORBA,
  DrBob42_c in 'DrBob42_c.pas',
  DrBob42_i in 'DrBob42_i.pas';
var
  Rate: Rates; // skeleton object
  Account: Accounts; // skeleton object
begin
  CorbaInitialize;
  // Add CORBA client Code Here
  Rate := TRatesHelper.Bind;
  Account := TAccountsHelper.Bind;
  // AccountArrayTest;
  // AccountSequenceTest;
end.
```

➤ *Listing 3: CORBA client application.*

```
procedure AccountArrayTest;
var
  MyAccounts: AccountArray;
begin
  MyAccounts[0] := TNormalOrSavingAccount.Create;
  MyAccounts[0].accountN := TNormalAccount.Create(7); // normal
  MyAccounts[1] := TNormalOrSavingAccount.Create;
  MyAccounts[1].accountN := TNormalAccount.Create(42); // normal
  MyAccounts[2] := TNormalOrSavingAccount.Create;
  MyAccounts[2].accountS := TSavingAccount.Create(42,Rate); // saving
  Account.AccountArrayTest(MyAccounts);
end;
```

➤ *Listing 4: Testing array of accounts.*

```
procedure AccountSequenceTest;
var
  MyAccounts: AccountSequence;
begin
  SetLength(MyAccounts,2);
  MyAccounts[0] := TNormalOrSavingAccount.Create;
  MyAccounts[0].accountN := TNormalAccount.Create(7); // normal
  MyAccounts[1] := TNormalOrSavingAccount.Create;
  MyAccounts[1].accountS := TSavingAccount.Create(42,Rate); // saving
  Account.AccountSequenceTest(MyAccounts);
end;
```

➤ *Listing 5: Testing sequence of accounts.*

CORBA Implementation

The `AccountArrayTest` and `AccountSequenceTest` routines in Listings 4 and 5 call the implementation of the CORBA server which wasn't covered, yet. We've seen how to create instances of argument types, and now it's time to focus on using passed argument types. This turns out to be very easy: Delphi's own Code Insight will help us use the correct property. The only 'advanced' issue here is the `_discriminator` property of the array or sequence items, which has a value of either `normal` or `saving`, and based on that value we can access the `accountN` or `accountS` property.

In short, the code for the CORBA server implementation is given in Listing 6. I'm just using simple `ShowMessages` to display the individual items of the array and sequence.

Note that the example is not really useful, since all I do is walk through the array or sequence and determine what kind of account we find next, printing only the account information (for normal accounts) or the account information including the current interest rate (for saving accounts). However, it might become a bit more interesting, at least theoretically, once we realise that we never called the `SetRate` method of the `Rates` interface, yet. We could call this method, from the client, if we want to change the interest rate. Note that since there is but one CORBA server `Rates` object, all the clients will automatically be working with the new interest rate once it has been changed. However, having the interest rates change behind your back does not mean that a client is automatically notified that the interest rate has changed, which leads to our last topic of today...

CORBA Callbacks

So far we've seen CORBA servers that are being called from CORBA clients. That's one server with many clients, with the initiative always at the client (making a request), and the server merely responding to the client. But what if we had a CORBA server that needed to inform its clients that something was changing (say, the interest rate suddenly got changed from 4.0 to 4.2 per cent?), or what if the CORBA server simply had to contact the CORBA client to obtain a client-specific interest rate in the first place? In those cases, we would need to use CORBA callbacks to allow the server to talk to the client instead of the other way around.

The solution that I'm about to implement involves the definition and implementation of the `Rates` interface at the client level. We will do this by modifying the DrBob42_impl.pas file, taking out the `TRates` class definition and implementation, and placing it in a file called Rates_impl.pas. This new file will be included by the CORBA client, and not by the CORBA server any more. The easiest way to do this is just to copy the file DrBob42_impl.pas to Rates_impl.pas, remove all the `TRates` from DrBob42_impl.pas and remove everything but `TRates` from Rates_impl.pas, and don't forget to change the unit name inside Rates_impl.pas as well.

➤ *Listing 6: CORBA server implementation.*

```
unit DrBob42_impl;
{This file was generated on 31 Jan 2001 11:22:44 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 D:\usr\bob\magazine\DELPHI.MAG\#67\
  src\drbob42.idl.}
{Delphi Pascal unit      : DrBob42_impl
}
{derived from IDL module : DrBob42
}
interface
uses
  SysUtils,
  CORBA,
  DrBob42_i,
  DrBob42_c;
type
  TRates = class;
  TAccounts = class;
  TRates = class(TInterfacedObject, DrBob42_i.Rates)
  protected
    FRate: Single;
  public
    constructor Create;
    function  interest_rate: Single;
    procedure SetRate (const rate: Single);
  end;
  TAccounts = class(TInterfacedObject, DrBob42_i.Accounts)
  protected
  public
    constructor Create;
    procedure AccountArrayTest(const Accounts:
      DrBob42_i.AccountArray);
    procedure AccountSequenceTest(const Accounts:
      DrBob42_i.AccountSequence);
  end;
implementation
constructor TRates.Create;
begin
  inherited;
  FRate := 1
end;
function TRates.interest_rate: Single;
begin
  Result := FRate
end;
procedure TRates.SetRate(const rate: Single);
begin
  FRate := rate
end;
constructor TAccounts.Create;
begin
  inherited
end;
procedure TAccounts.AccountArrayTest(const Accounts:
  DrBob42_i.AccountArray);
var
  i: Integer;
begin
  writeln('OK');
  for i:=0 to 2 do
  begin
    if Accounts[i]._discriminator = normal then
      ShowMessage(Format('Normal Balance %d: %1.2f',
        [i+1,Accounts[i].accountN.balance]))
    else
      ShowMessage(Format('Savings Balance %d: %1.2f at
        %1.2f%%', [i+1,Accounts[i].accountS.balance,
        Accounts[i].accountS.rates.interest_rate]))
  end
end;
procedure TAccounts.AccountSequenceTest(const Accounts:
  DrBob42_i.AccountSequence);
var
  i: Integer;
begin
  for i:=0 to High(Accounts) do // use High on Open Array
  begin
    if Accounts[i]._discriminator = normal then
      ShowMessage(Format('Normal Balance %d: %1.2f',
        [i+1,Accounts[i].accountN.balance]))
    else
      ShowMessage(Format('Savings Balance %d: %1.2f at
        %1.2f%%', [i+1,Accounts[i].accountS.balance,
         Accounts[i].accountS.rates.interest_rate]))
  end
end;
initialization
end.
```

```
program CClient;
{$DEFINE CALLBACK}
{$APPTYPE CONSOLE}
uses
  SysUtils, CORBA,
  DrBob42_c in 'DrBob42_c.pas',
  DrBob42_i in 'DrBob42_i.pas',
  DrBob42_s, Rates_impl;
var
  Rate: Rates; // skeleton object
  Account: Accounts; // skeleton object
procedure AccountArrayTest;
var
  MyAccounts: AccountArray;
begin
  MyAccounts[0] := TNormalOrSavingAccount.Create;
  // normal
  MyAccounts[0].accountN := TNormalAccount.Create(7);
  MyAccounts[1] := TNormalOrSavingAccount.Create;
  // normal
  MyAccounts[1].accountN := TNormalAccount.Create(42);
  MyAccounts[2] := TNormalOrSavingAccount.Create;
  MyAccounts[2].accountS :=
    TSavingAccount.Create(42,Rate); // saving
  Account.AccountArrayTest(MyAccounts);
end;
proocedure AccountSequenceTest;
var
  MyAccounts: AccountSequence;
begin
  SetLength(MyAccounts,2);
  MyAccounts[0] := TNormalOrSavingAccount.Create;
  // normal
  MyAccounts[0].accountN := TNormalAccount.Create(7);
  MyAccounts[1] := TNormalOrSavingAccount.Create;
  MyAccounts[1].accountS :=
    TSavingAccount.Create(42,Rate); // saving
  Account.AccountSequenceTest(MyAccounts);
end;
var
  R: Integer;
begin
  CorbaInitialize;
{$IFDEF CALLBACK}
  Rate := TRatesSkeleton.Create('Rates', TRates.Create);
  BOA.ObjIsReady(Rate as _Object);
{$ENDIF}
  write('Rate: ');
  readln(R);
  try
    // Add CORBA client Code Here
  {$IFNDEF CALLBACK}
    Rate := TRatesHelper.Bind;
  {$ENDIF}
    Rate.SetRate(R);
    Account := TAccountsHelper.Bind;
    AccountArrayTest;
    AccountSequenceTest;
  except
    on E: Exception do
      writeln(E.Message)
  end;
  readln;
end.
```

➤ *Listing 7: Callback-enabled CORBA client.*

After you've split the implementation units, you'll notice that the CORBA client still compiles (nothing has changed to break it), but the CORBA server no longer compiles: it doesn't know the `TRates` class type any more, which is needed to create an instance of the `TRatesSkeleton` (see Listing 2 to refresh your memory). That's no big deal, since the CORBA server is no longer implementing the `Rates` anyway, so just remove these two lines from the CORBA server project file (the lines regarding `Rates`). In the source code on the disk, I've used a compiler conditional variable called `CALLBACK` to distinguish between the original and the callback version of the project.

Now that the CORBA server knows nothing about the `Rates` any more, let's put it in the CORBA client. For this, you need to open up the CORBA client project, add the `DrBob42_s` and `Rates_impl` units to the `uses` clause, and add a few lines of code (that were removed from the CORBA server just a minute ago) inside `{$IFDEF CALLBACK}` compiler conditionals, namely:

```
Rate := TRatesSkeleton.Create(
  'Rates', TRates.Create);
BOA.ObjIsReady(
  Rate as _Object);
```

We're almost done: in case we use the callback code as shown above, the CORBA client should not create the `Rate` object by calling the `TRatesHelper.Bind` (there's nothing to bind, the object is already here), so I have used similar `{$IFNDEF CALLBACK}` compiler conditionals to make sure this code isn't called when it's not needed (see the final listing).

The rest is already in place: in both cases we pass the `Rates` interface as a field member of the `SavingAccount` (which is a possible type of the `TNormalOrSavingAccount` type), and the CORBA server will need to call the `Rates` interface to obtain the rates (using the method `interest_rates`), which results in a callback to the client that implemented this server. Presto!

The final implementation of the callback-enabled CORBA client is shown in Listing 7.

If you compare Listing 3 to Listing 7 you will notice a lot of changes. One is the use of `IFDEF`s to change between the original version and the callback version of the code, and another change is the use of `try..except` blocks to catch any CORBA exceptions that may have been raised by the CORBA server.

In this article, unlike last month, I didn't use any special exceptions, but we also don't use `try..except` blocks in the CORBA server implementation (see the code in DrBob42_impl.pas), so anything that goes wrong on the CORBA server that raises an exception will result in an exception (usually the `BAD OPERATION` CORBA exception) to be raised on the client side.

Next Time

In this article, we've seen a CORBA Windows server application and a console client application. We have also experimented with enumerated types, sequences and arrays, and played with CORBA callbacks.

Next time, I'm going a little bit deeper into the topic of general interfaces, specifically an interface called `IDataAware`, which might just be a more convenient and easier way to manage and define data-aware components in Delphi.

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