Borland®

CORBA® with Delphi™

by Bob Swart

Table of Contents

Introduction	1
CORBA®	2
Delphi™ 5 Enterprise	2
Delphi 5 for CORBA Server	3
CORBA Clients with Delphi 5	4
VisiBroker 3.3 for Delphi 5	7
Conclusion	11

VisiBroker® 3.3 for Delphi™ 5

An introduction to CORBA® is followed by the design of a practical, real-world CORBA application implemented in Delphi™ 5 Enterprise. The CORBA application acts as a multi-user electronic personal diary that can be used to check other people's diaries, make appointments, and so forth. This can be useful in corporate environments where employees are not always working in the same (physical) office.

We'll implement the CORBA Server in Delphi, followed by a CORBA Client in Delphi using three different communication techniques: the Type Library, Dynamic Interface Invocation (DII) and the IDL2PAS that came with VisiBroker® 3.3 for Delphi 5.

VisiBroker®

white paper

CORBA®

CORBA stands for Common Object Request Broker Architecture. CORBA is platform-independent like Java™ , and language-independent like COM - truly the best of the best. CORBA is a communication protocol between a client and a server. Communication between these two is handled by an ORB (Object Request Broker) and IIOP (Internet InterORB Protocol).
Before a client and a server can communicate with each other, a "contract" must be made that specifies the functionality that will be implemented by the server and available (to call) for the client. Such a contract is defined using IDL (Interface Definition Language). With IDL, you can specify a module, which consists of one or more interfaces, each of which can contain methods and exceptions, but more about exceptions later.
Once the IDL contains the interface contract, it still consists of a platform-independent and language-independent specification. The IDL definition can then be translated into platform-specific and language-specific parts, using for example, an IDL2JAVA, IDL2CPP or IDL2PAS.
The IDL is translated into Server Skeletons and Client Stubs. The Server Skeleton defines the module/interface and methods that the server needs to implement, while the client stubs define the module/interface methods that the client can call.
The best feature of CORBA is, again, the fact that it is a cross-platform language-independent communication protocol. Specifically, the client and server do not need to run on the same platform, nor do they need to be implemented in the same language. What makes all this work is the ORB which passes a request for a method call on from the client to the server. The real work is done when passing the arguments (and values) from one place to another. In order to make sure that arguments values are correctly passed on, they are transferred in a platform-independent, language-independent format. When the client calls a method passing an argument, then the argument is converted to an ORB-specific format (this is called marshalling). When the server receives a call, the ORB-specific format is converted back to the correct values which are native to the platform and language of the server (this is called unmarshalling). The same thing happens when the server passes the results back to the client.
All in all, CORBA is a powerful cross-platform and cross-language communication protocol that can be used to connect many clients to a single server. In this paper we'll explore the way in which Delphi 5 supports CORBA, using VisiBroker 3.3 for Delphi 5.

Delphi™ 5 Enterprise

Using Delphi 5 Enterprise, you can create a CORBA Data Module or a plain CORBA Object (both Wizards can be found in the Multi-tier tab of the Object Repository). In this paper, I'll focus on the plain CORBA Object only, which is the one that can act as a CORBA server (object) for cross-platform cross-language CORBA clients.
The example that I've announced in the introduction of this paper is based on a real-world, multi-user personal diary (a long description for an "on-line diary", wouldn't you say?). We've been using an application similar to the one I'm describing in this paper for over a year now. Unfortunately, I have neither the space, nor the time to cover all the implementation details here, so I'll stick to the CORBA communication details only (I'm sure you'll understand, since VisiBroker 3.3 for Delphi 5 is the topic for this paper).
The on-line diary contains information for employees and their daily activities (on an hourly basis for example). The central server would contain (a database with) all on-line diaries, and the remote clients can access this central server to make appointments with each other. The power of CORBA ensures that nobody really needs to care how or where the server is implemented (as long as they can

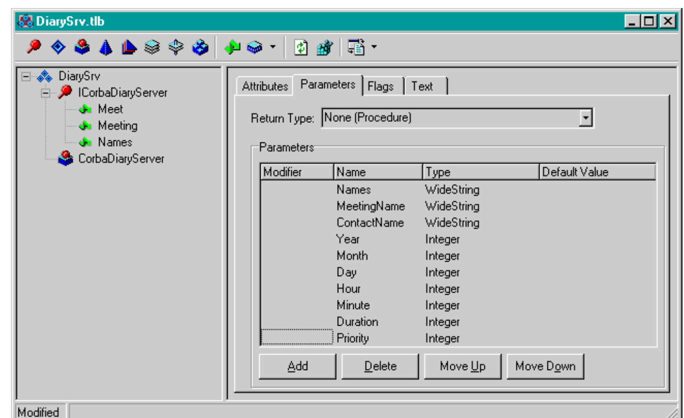
connect to it), nor does the server need to care where or how the clients are implemented. In this case the server is implemented in Delphi 5 Enterprise. Since this is a Delphi paper, the client will too, but you'll see a number of techniques that will also (or especially) be useful on "foreign" servers or "foreign" clients.

Delphi 5 for CORBA Server

To create a Delphi 5 for CORBA Server application, you need to start a new application (with a main form that can act as "identifier" to show that the server is up and running). To this application, which I've called DiarySrv, you need to add a Corba Object (from the Multi-tier page of the Object Repository).
Inside the CORBA Object Wizard, you need to specify the Class Name of the CORBA object. This will be the name of the interface (with an I-prefix), as well as the TCorbaImplementation descendant class (with the T-prefix), that your CORBA application will create and use. The interface is the important part here: this is where you can specify the methods that the server needs to implement, and that the client can call. In Delphi 5, specifying these interface methods is done using the Type Library. This gives CORBA a bit of a COM taste, but it's not truly COM (it's only using interfaces behind the scene - the actual communication is still done using CORBA). Specify a class name here (such as CorbaDiaryServer), leave the Instancing and Threading-model options at their default values, and hit OK.

Type Library Editor

After you've created the new CORBA Object, you must start the Type Library Editor to specify the interface definition of the CORBA server. In this case, I need to specify a single method called Meeting, with four arguments: Names, Date, Time and Duration. Names is a PChar (LPSTR) that contains the (unique specified) names of the people that need to be present at that meeting (a hence it would be nice if they were to be automatically notified). Date and Time are in the YYYYMMDD and HHMM formats, which makes them sortable! Finally, Duration specifies the number of minutes the meeting is supposed to last (note: 24*60 = 1440 minutes make up an entire day).



You should hit the "Refresh Implementation" button to make sure the Delphi unit for the CorbaDiaryServer contains this (still empty) method Meeting. Once the method Meeting is available, you can implement it anyway you want (in the paper, I'll just e-mail it to the office manager, who then has to make sure the appointments are made - but you're free to make a more sophisticated implementation, of course).
Type Library Editor Limitations
While the Type Library Editor is a convenient way for Delphi developers to specify the interface of a (CORBA or COM) object, there are a number of limitations when using the Type Library Editor. It just doesn't support everything CORBA has to offer, like structured types

(records) or CORBA exceptions. This may not be a big deal for the average Delphi developer, but for a CORBA developer it is a big deal. Later in this paper, I'll show you VisiBroker 3.3 for Delphi 5, which does not have these limitations.

Exporting IDL

Apart from implementing the interface definition, you should also export, it in order to make sure that potential CORBA clients know which method(s) are available, and which arguments direct or point to these methods.
Although you've used the Type Library Editor so far, you haven't really seen IDL, yet (remember the theory part at the beginning of this paper?).
While still in the Type Library Editor, you must also export the type library definition as CORBA IDL, so that any CORBA client can use this IDL to generate client stubs. Simply select the "Export to CORBA IDL" using the last button on the right of the Type Library Editor. This resulting IDL file can be used by CORBA clients, for example, implemented using JBuilder™ , C++Builder™ or Delphi.
The IDL file for the module DiarySrv with example interface ICorbaDiaryServer (and method Meeting) should look similar to this:

```
module DiarySrv
{
  interface ICorbaDiaryServer
  {
    void Meeting(in string Names,
                 in long Date, // yyyymmdd
                 in long Time, // hhmm
                 in long Duration);
  };

  interface CorbaDiaryServerFactory
  {
    ICorbaDiaryServer CreateInstance(in string
InstanceName);
  };
};
```

Now, close the Type Library Editor and return to the CORBA object implementation unit. This is the time to implement the Meeting method inside your TCorbaDiaryServer class. Once you're done that, you can compile and test the CORBA Server.
Running CORBA Server
Before you can run the CORBA Server, you must first make sure the (VisiBroker) Smart Agent is running. You can find the Smart Agent in your Delphi program group, or as osagent.exe in your VBROKER\BIN directory. The Smart Agent can be seen as the telephone service; without it, you cannot reach the ORB or send CORBA messages from one place to another.
When the Smart Agent runs (on at least one machine in the subnet), you can run the CORBA Server. The server will "register" itself with the Smart Agent, and from that moment on will wait until you terminate it again, or a call from CORBA client comes in (dispatched by the Smart Agent). It is time to leave the CORBA Server alone, and start working on CORBA clients to connect to it.

CORBA Clients with Delphi 5

There are three different ways to create a CORBA Client using Delphi 5. The oldest way is using the Type Library import unit (provided a CORBA Server was written in Delphi 5 that used a Type Library), and use that to create a (CORBA) object instance and call its methods. Note that this only works when both the CORBA client and server are written in Delphi, so I don't consider this "pure" CORBA (but you'll see how it works anyway, because you do get a great deal of design-time support).
The second way to create a CORBA client, when you have an IDL file and no Type Library at your disposal, is by registering the IDL file inside the Interface Repository,

and using Dynamic Interface Invocation to create an object and call its methods. This technique is very flexible (when the interface changes, your client can dynamically adapt to it), but also harder to implement. Apart from that, due to the dynamic nature of the interface, there is hardly any design-time support.

The third and without a doubt best way to create CORBA clients in Delphi, is by using an add-on tool available for Delphi 5 Enterprise called VisiBroker 3.3 for Delphi 5. This add-on tool, which can be downloaded from the Borland website, www.borland.com, contains an IDL2PAS compiler that produces client stubs from IDL files. During the session, I've been using a prerelease version of VisiBroker 3.3 for Kelphi5, which is now available the release edition). However, the techniques shown will be quite useful to CORBA developers...

1. CORBA Client using Type Library

The easiest way to connect a Delphi for CORBA Client to an existing Delphi for CORBA Server (that was made using the Type Library Editor), is by using the Type Library Import unit. Start a new project and add the Type Library import unit from the CORBA Server to your CORBA Client project. The import unit contains the factory constructor that you need to call in order to create an instance of the CORBA server. It also contains the full definitions of the methods that belong to the interfaces (so you get full design-time support including CodeInsight™ help for arguments). After you've created the CORBA server, you can call its methods as if they were local methods. For example, to call the Meeting method, you need to write the following code:

```
procedure TCorbaDiaryClient.TypeLibrary(Sender:
TObject);
var
  CorbaDiaryServer: ICorbaDiaryServer;
begin
  CorbaDiaryServer :=
TCorbaDiaryServerCorbaFactory.CreateInstance('CorbaDia
ryServer');
  CorbaDiaryServer.Meeting('Bob
Swart',20000925,1715,75);
  CorbaDiaryServer := nil
end;
```

Before you can test this, you must make sure the Smart Agent is running, then start the CORBA server, followed by your CORBA client. As long as both the CORBA Server and Client are written in Delphi, you can rely on the Type Library (and import unit) to connect them to each other. However, the earlier mentioned limitations still apply: no support for records or CORBA exceptions.

If the CORBA Server is written in another environment (like C++ or Java), then there's no Type Library import unit, and you need to use one of the following techniques to write a CORBA Client in Delphi.

2. CORBA Client using Dynamic Interface Invocation (DII)

Although I will only be using Delphi 5 Enterprise during this paper, let's assume just for the sake of argument that the CORBA Server is written in some "foreign" environment and/or you do not have the Type Library (import unit) available when you start to write the CORBA Client in Delphi 5. Instead, you start with an IDL file that defines the interface that the CORBA Server implements and is available for your CORBA Client to call. Some people have imported an IDL definition in the "text" page of the Type Library and hit the "refresh" button to generate the corresponding Delphi interface, but I won't be doing that, since I've already explained that the Delphi Type Library does not contain full support for the CORBA IDL (so this might or might not work, depending on the IDL features that are used).

The Interface Repository

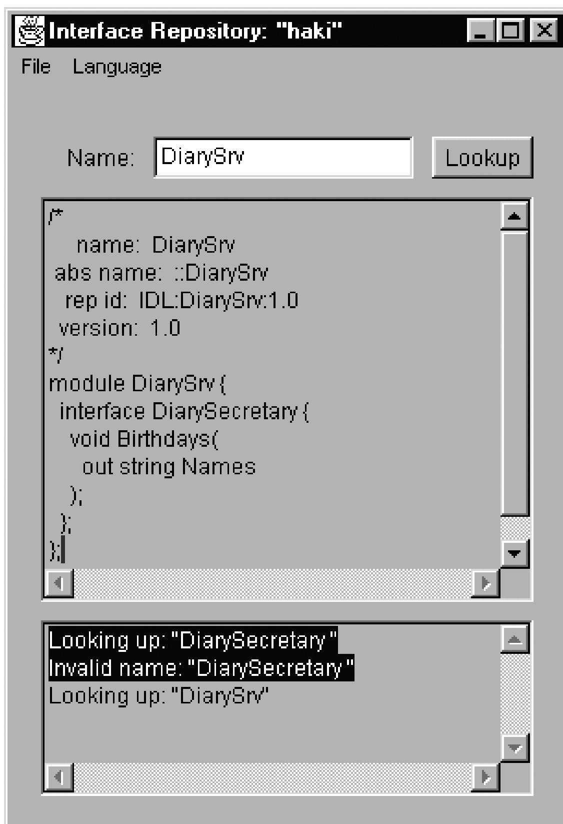
Since Delphi (out-of-the-box) can't directly use the IDL we've just written, you must use a different technique. The CORBA client will dynamically communicate with a

repository that stores available interfaces. This repository is called the Interface Repository. It needs to be connected to an ORB, so it can connect the CORBA Server that implements this interface (method) with the CORBA client that requests this interface (method). The technique of connecting like this is also known as "late binding". If you "load" an interface definition (IDL file) inside the Object Repositor, then it is available for use by all CORBA clients that can talk to the Interface Repository (which isn't difficult, as you'll see in a minute).

First, let's start the Interface Repository and load the "DiarySrv.idl" file in it. This can be done by the following command-line:

```
start irep drbob42 DiarySrv.idl
```

Keep in mind that the VisiBroker Smart Agent must be running on the same machine on which the Interface Repository is started. This will start a GUI application where you can even do some simple searching on the available IDLs.



The Delphi client is now able to access the IDL and invoke methods dynamically.

Dynamic Interface Invocation (DII)

The code needed to dynamically obtain an object reference uses the OrbBind method, after which you can call any method from this CORBA server object reference. Note that method calls are now case-sensitive (and you won't get any design-time support using CodeInsight):

```
procedure
TCorbaDiaryClient.DynamicInterfaceInvocation(Sender:
TObject);
var
  Factory: TAny;
  CorbaDiaryServer: TAny;
begin
  Factory :=
Orb.Bind('IDL:DiarySrv/CorbaDiaryServerFactory:1.0');
  CorbaDiaryServer :=
Factory.CreateInstance('CorbaDiaryServer');
  CorbaDiaryServer.Meeting('Bob
Swart',20000925,1715,75);
  CorbaDiaryServer := unassigned;
  Factory := unassigned
end;
```

DII is a powerful way to talk to CORBA servers if interfaces change often and the names of the methods remain the same. You don't need to recompile the client, or distribute the IDL file to be able to use the interfaces - as long as the Interface Repository can be used. However, the downside for Delphi developers is the lack of design-time support (you won't get error messages until it's too late: when the application is running). For that and more, you need to look at the third way to write CORBA Clients in Delphi, using IDL2PAS.

3. CORBA Client using IDL2PAS

The first public edition of VisiBroker 3.3 for Delphi 5 Enterprise shipped at the end of 1999. It contained an IDL2PAS batch file, some examples and documentation (including a VisiBroker 3.3 for Pascal Reference Guide in PDF format).

The IDL2PAS batch file "compiles" a CORBA IDL file, like DiarySrv.idl, into DiarySrv_i.pas and DirarySrv_c.pas. The DiarySrv_i.pas file will get the interface definition, while the DiarySrv_c.pas file will get the client stub definitions. As you may have noted, there was no support for server skeletons, which would have been ended up in a _s or _impl file. During the paper in London, I'll show the latest version of IDL2PAS that also generates server skeletons.

Using the generated DiarySrv_i.pas and DiarySrv_c.pas files with the new CORBA and OrbPas30 units (that also come with VisiBroker 3.3 for Delphi 5), you can now statically bind to any CORBA server. To do so, you must first create the Factory of type CorbaDiaryServerFactory (defined in the DiarySrv_c.pas unit), by calling TCorbaDiaryServerFactoryHelper.Bind with CorbaDiaryServer as argument. With this Factory instance, you can create the CORBA server itself by calling the CreateInstance method with CorbaDiaryServer as argument.

```
  uses
    CORBA, OrbPas30, DiarySrv_i, DiarySrv_c;

  procedure TCorbaDiaryClient.IDL2PAS(Sender: TObject);
  var
    Factory: CorbaDiaryServerFactory;
    CorbaDiaryServer: ICorbaDiaryServer;
  begin
    Factory :=
TCorbaDiaryServerFactoryHelper.Bind('CorbaDiaryServer')
;
    CorbaDiaryServer :=
Factory.CreateInstance('CorbaDiaryServer');
    CorbaDiaryServer.Meeting('Bob
Swart',20000925,1715,75);
    CorbaDiaryServer := nil;
    Factory := nil
  end;
```

All three approaches (Type Library import unit, dynamic interface invocation and IDL-2-PAS binding) have the same CorbaDiaryServerMeeting call. The differences are in the way that you create an instance of the CORBA server itself. One big difference is the fact that the Type Library import unit method only works with CORBA Servers for Delphi. A second big difference is the fact that when using Dynamic Interface Invocation (DII), you lack design-time support (because the CORBA Server itself is of type TAny). Finally, the IDL-2-PAS offers the ability to bind to any CORBA server, yet gives you design-time support as well. Add to that the the other enhancements that you find in VisiBroker 3.3 for Delphi 5, and you'll understand why this is my favorite technique.

VisiBroker 3.3 for Delphi 5

During the session, I've been using a prerelease version of VisiBroker 3.3 for Delphi 5 (with special permission from Borland's Ben Riga), which is now available (the release edition). However, the techniques shown will be quite useful to CORBA developers, even if they still use the first (original) public edition of VisiBroker 3.3 for Delphi 5.

CORBA structs

Let's start with a a CORBA feature that is supported by the original IDL-2-PAS - and not available when using the Type Library import unit. I'm talking about the support for CORBA structs, which are a bit similar to plain ObjectPascal record types (to be honest, CORBA structs are a lot more powerful since they can contain recursive definitions, but that's a story for another day). To extend your DiarySrv example, you can combine the Date and

Time integer values in a single structure DateTime defined as follows in IDL (note that I merged the Date and Time arguments to the Meeting method as well):

```
module DiarySrv
{
  struct DateTime {
    long Date;
    long Time;
  };

  interface ICorbaDiaryServer
  {
    void Meeting(in string Names,
                 in DateTime DayTime,
                 in long Duration);
  };

  interface CorbaDiaryServerFactory
  {
    ICorbaDiaryServer CreateInstance(in string
InstanceName);
  };
};
```

Even using the first edition of IDL2PAS, the resulting struct DateTime can be found inside both the DiarySrv_i.pas and the DiarySrv_c.pas generated files. The generated DiarySrv_i.pas file contains the interface definition of DateTime:

```
type
  DateTime = interface
    ['{5070DDE7-BD49-0C52-68CE-EF2214199CB8}']

    { Accessor and mutator methods for IDL Structure
Elements. }
    function   _get_Date : Integer;
    procedure  _set_Date (const Date : Integer);
    function   _get_Time : Integer;
    procedure  _set_Time (const Time : Integer);

    { Properties representing IDL Structure Elements. }
    property  Date : Integer read _get_Date write
_set_Date;
    property  Time : Integer read _get_Time write
_set_Time;
  end;
```

As with all Delphi interfaces, this is only a definition of functionality; the implementation must be made someplace else (inside, the generated DiarySrv_c.pas file):

```
type
  TDateTime = class (TInterfacedObject,
DiarySrv_i.DateTime)
  private
    Date : Integer;
    Time : Integer;
    constructor Create; overload;
  public
    function   _get_Date : Integer; virtual;
    procedure  _set_Date ( const _value : Integer );
virtual;
    function   _get_Time : Integer; virtual;
    procedure  _set_Time ( const _value : Integer );
virtual;
    constructor Create (const Date : Integer;
                        const Time : Integer); overload;
  end;
```

In order to use the TDataTime type, you only have to create an instance of it, at which time you can pass the Date and Time values to the Create constuctor, or set their values using the _set_Date and _set_Time methods later. Clients receiving a TDataTime argument can obviously use the _get_Date and _get_Time functions to obtains their values again. It sounds really easy, and it is. For those of you interested in where the actual marshalling is happening, take a look at the TDateTimeHelper class (defined in the same DiarySrv_c.pas file) that is doing a lot of the hard work behind the scenes (including marshalling the individual record fields).

CORBA Exceptions

It's nice to have support for CORBA structs (this was available in the original edition of VisiBroker 3.3 for Delphi 5), but what about error handling and recovery? Delphi 5 has no built-in support for CORBA exceptions. And even the original VisiBroker 3.3 for Delphi 5 only supports client-side CORBA exceptions. *Only client-side?* Yes, and that means that the CORBA client is able to "catch" (or handle) exceptions using a try-except block, but the CORBA server is not (yet) able to raise any CORBA exceptions. This includes standard as well as custom CORBA exceptions. The next edition of VisiBroker 3.3 for Delphi 5 actually supports CORBA server-side exceptions as well (great news!), so let's examine CORBA exceptions in a bit more detail.

A CORBA exception is like a struct definition in IDL. You can add data (fields), but no functionality (no methods). Apart from that, you cannot built an exception hierarchy, as structs cannot be derived from other structs (for me, this is one of the few minor disappointments when using CORBA).

To extend the DiarySrv example with CORBA exceptions, consider a situation where I try to setup a meeting using the Meeting method, only to find that there was no way to signal that a meeting is impossible to schedule. It would be nice to get a notification of such an event by raising an exception (containing useful feedback information, if possible). You can define an "MeetingImpossible" exception, which can be raised if one or more of the people can't attend the meeting (or if the Date/Time is invalid). Using IDL, you can define an exception MeetingImpossible, including a property called Reason, and extend the DiarySrv.idl until we get the final version:

```
module DiarySrv
{
  exception MeetingImpossible
  {
    string Reason;
  };

  struct DateTime {
    long Date;
    long Time;
  };

  interface ICorbaDiaryServer
  {
    void Meeting(in string Names,
                 in DateTime DayTime,
                 in long Duration)
      raises (MeetingImpossible);
  };

  interface CorbaDiaryServerFactory
  {
    ICorbaDiaryServer CreateInstance(in string
InstanceName);
  };
};
```

In the above example, the custom CORBA exception MeetingImpossible is defined outside the ICorbaDiaryServer interface. It could also have been defined inside the ICorbaDiaryServer interface (resulting in a "local" exception - visible to the ICorbaDiaryServer only - which results in a slightly different generated class name EICorbaDiaryServer_MeetingImpossible).

VisiBroker 3.3 for Delphi 5 supports all CORBA standard exceptions already. What we've been working on just now is known as a CORBA custom exception. Both built-in and custom CORBA exceptions are mapped by the IDL2PAS compiler to an ObjectPascal class type derived from SystemException (for built-in CORBA exceptions) or UserException (for the custom exception types) - both are derived from the standard ObjectPascal Exception type, by the way. For the above definition, the original

IDL2PAS generated the following CORBA custom exception class inside the DiarySrv_c.pas file:

```
  type
    EMeetingImpossible = class(UserException)
    private
      FReason : AnsiString;
    protected
      function  _get_Reason : AnsiString; virtual;
    public
      property  Reason : AnsiString read _get_Reason;
      procedure Copy(const _Input : InputStream);
override;
    end;
```

While the above definition is useful for "catching" exceptions in a try-except block, it offers no help to actually create an exception and raise it (which shouldn't be a surprise, as I've already stated over and over again that the original VisiBroker 3.3 for Delphi 5 did not support server-side exceptions). In contrast, the new and just released version of VisiBroker 3.3 for Delphi 5 generates the following class for the custom CORBA exception:

```
  type
    EMeetingImpossible = class(UserException)
    private
      FReason : AnsiString;
    protected
      function  _get_Reason : AnsiString; virtual;
    public
      property  Reason : AnsiString read _get_Reason;
      constructor Create; overload;
      constructor Create(const Reason : AnsiString);
overload;
      procedure Copy(const _Input : InputStream);
override;
      procedure WriteExceptionInfo(var _Output :
OutputStream); override;
    end;
```

The most obvious new additions are the two (overloaded) constructors Create, which enable us to create this EMeetingImpossible custom exception with or without an initial Reason string. Once an EMeetingImpossible exception instance has been created, we can raise it - at the CORBA server side - just like you'd raise a regular Delphi exception:

```
  raise DiarySrv.EMeetingImpossible('Exception raised by
Delphi 5');
```

Once the CORBA server raises this exception, it will not be handled by the CORBA server application itself. Instead, the VisiBroker 3.3 ORB will receive this exception, marshall it into a native CORBA exception (including the fields inside the exception struct), and pass it onto the CORBA network. At the client side, the CORBA exception will be marshalled back into an EMeetingImpossible CORBA custom exception and actually raised (at the point where the client made the call to the server method that caused the exception to be raised in the first place).

In short, handling CORBA exceptions is hardly any different from handling regular ObjectPascal exceptions:

```
  uses
    CORBA, OrbPas30, DiarySrv_i, DiarySrv_c;

  procedure TCorbaDiaryClient.IDL2PAS(Sender:
TObject);
  var
    CorbaDiaryServer: ICorbaDiaryServer;
    DateTime: TDateTime;
  begin
    Server :=  TICorbaDiaryServerHelper.Bind;
    DateTime := TDateTime.Create(200009,13);
    try
      Server.Meeting('Micha, Rick, Arnim',DateTime,75);
    except
      on E: EMeetingImpossible do
        ShowMessage(E.Reason)
    end;
```

```
    DateTime.Free;
    CorbaDiaryServer := nil
  end;
```

CORBA Server Skeletons

Apart from generating exception types that can be used for server-side exception raising, the IDL2PAS that comes with new edition of VisiBroker 3.3 for Delphi 5 also generates true Server Skeleton code, generating both an DiarySrv_s.pas and DiarySrv_impl.pas unit (as we'll explore in detail during the paper). This last feature truly makes the Type Library Editor and Import Unit obsolete for CORBA develoment using Delphi (once the new VisiBroker 3.3 for Delphi 5 is officially available, that is).

Conclusions

Delphi supports CORBA servers with a Type Library Editor/Import Unit interface as well as using Dynamic Interface Invocation (DII) or by a true IDL2PAS as part of VisiBroker 3.3 for Delphi 5. The latest edition of VisiBroker 3.3 for Delphi 5 supports CORBA structs and client-side as well as server-side exceptions as a few ofthe more interesting CORBA features - apart from client stubs and server skeletons.

And now only one personal wish remains: support for VisiBroker 4 in Delphi 6 (as the current VisiBroker 3.3 for Delphi 5 is still version 3.3).

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 the Inprise/Borland Conferences since 1993. He is a free-lance technical author for The Delphi Magazine, UK-BUG Developer's Magazine, Delphi Developer and wrote chapters for The Revolutionary Guide to Delphi 2 (WROX), Delphi 4 Unleashed, C++Builder 4 Unleashed, and C++Builder 5 Developer's Guide (SAMS).

©2000 Informant Communications Group, Inc. All Rights Reserved. Used by permission. Unauthorized distribution or duplication is strictly prohibited. For more information on Informant Communications Group, please visit http://www.DelphiZine.com

Borland®

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

Made in Borland®. Copyright © 2001 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. CORBA is a trademark or registered trademark of Oject Management Group, Inc. in the U.S. and other countries. All other marks are the property of their respective owners. 11856