CORBA Development

by David Sampson

CHAPTER

19

IN THIS CHAPTER

CORBA stands for *Common Object Request Broker Architecture*. Its purpose is to facilitate distributed object computing. Unlike a proprietary approach such as DCOM, CORBA is an open standard that isn't under the control of any single company. An organization called the Object Management Group (OMG), which is made up of more than 800 industry representatives, controls the CORBA specification. The OMG meets periodically to issue updates or amendments to the standard and to resolve any outstanding issues.

The OMG specifies what CORBA will do and to a certain degree, how it will do it. Beyond that, each CORBA vendor is free to come up with its own implementation and method of complying with the CORBA specification. This freedom has a price. For example, the OMG doesn't specify how different CORBA implementations locate objects when using two different *ORBs (Object Request Brokers)*. So in the past, it has been a struggle to get applications to bootstrap together when they were written with different vendor's products. This is one area that has received a lot of attention and is continuing to improve as the CORBA specification evolves.

More information on the OMG is available at its Web site (`www.omg.org`). You'll find a wealth of information about CORBA, including the latest specifications, tutorials, Web links to vendors, and so on.

One thing you'll discover is that many free CORBA implementations are available on the internet. This chapter deals with the Borland CORBA implementation bundled with Delphi 6 Enterprise edition. The CORBA product is called VisiBroker, and is arguably the most widely used ORB in the world. Delphi 6 contains all the runtime library files needed to use CORBA. In addition, wizards are integrated into the IDE that make application development relatively straightforward.

CORBA Features

CORBA has several features that make it beneficial for use in distributed enterprise environments:

- CORBA is an object-oriented approach. Each CORBA server publishes an interface that lists the methods and data types it supports. The implementation details are hidden from the caller.
- Location Transparency. The real power in CORBA is that objects can be located anywhere. When a CORBA client application calls a server object, it doesn't know where the server resides. In fact, CORBA presents the client application with an image of the server application. The client then operates as if the server object is running locally in its own process space. This will be discussed in more detail in the CORBA Architecture section.

- Programming Language Independence. A major benefit is that objects can be written in a variety of different languages. Java and C++ are the leaders, but Delphi is gaining much wider acceptance because of all the features available in the product. To make sure that these languages can interoperate, CORBA objects interact with each other through their published interfaces. Each server object must comply with its interface definition. Because of the differences in programming languages, clients cannot know about or compensate for any of the implementation details on the server side. Strict object-oriented design is enforced in the CORBA world.
- Multi Platform/Multi Operating Systems. CORBA implementations exist for different platforms and operating systems. It isn't unusual for a deployment to use Java on the back-end mainframe computer and Delphi on the middle tier or client side. Developers can write powerful applications that tap into legacy systems and present information to end users with feature-rich clients built in Delphi.

CORBA Architecture

Figure 19.1 shows a block diagram of the CORBA architecture. The common piece to both the client and the server is the ORB. The ORB handles all communications between objects. It does this using the *Internet Inter-ORB Protocol (IIOP)* that is layered on TCP/IP. This guarantees reliable end-to-end message delivery and usage anywhere TCP/IP is deployed. In addition to handling all message traffic, the ORB also corrects for platform variations.

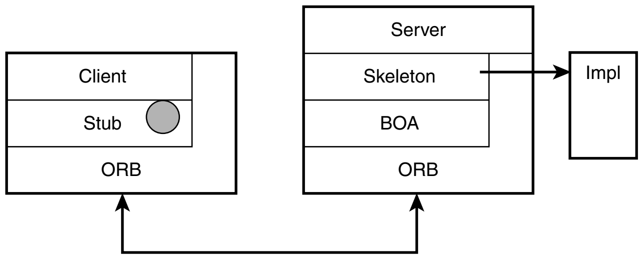


FIGURE 19.1
The CORBA Architecture.

For example, if the number `123` is originated on an Intel based machine and is sent to a Sun workstation, the number won't be processed correctly without some sort of intervention. This is because the two processors use different layouts for their registers.

This is referred to as the Big-Endian/Little-Endian problem. Because the ORB knows what platform it is running on, it will set a flag in the CORBA message to indicate whether it originated on a big-endian or little-endian machine. The receiving side will read this flag and automatically process the data correctly. This ensures that the number `123` is processed correctly on both ends.

NOTE

According to Rhu, Herron, and Klinker in *IIOP Complete* (Addison Wesley), p.65, "The terms *little endian* and *big endian* are an analogy drawn by Cohen from *Gulliver's Travels*, in which the islands of Lilliput and Blefescua feuded over which end of an egg to crack, the little end or the big end."

The client side consists of two additional layers. The client block is the application that is written by the developer. The more interesting piece is the stub. The stub is a file that is automatically generated by a tool that is included in the Delphi Enterprise edition. This tool is called the IDL2Pas compiler. Its purpose is to take files that describe the server interfaces and generate Delphi Pascal that can interact with the CORBA ORB. The IDL2Pas compiler is documented in a set of HTML files on the Delphi 6 CD-ROM (`Delphi6\Doc\Corba`).

The stub file contains one or more classes that "mirror" the CORBA server. The classes contain the same published interfaces and data types that are exposed by the server. The client calls the stub classes in order to communicate with the server. The stub classes act as a proxy for the server objects. The symbol in the stub block represents a connection to the server. The connection is established through a *bind* call that is issued by the client. The stub is said to have an *object reference* to the server (represented by the symbol). Once the connection is made to the server, the client invokes a method call on the stub class. The stub packs the request and any required method arguments into a buffer for transmission to the server. This is referred to as *marshaling* the data. The stub invokes the call through its server object reference via the ORB. When the server responds, the stub class receives the message from the ORB and hands the response back to the client.

The client can also call some utility type functions directly in the ORB. The block diagram shows this logical connection.

The server side contains an ORB interface called the *Basic Object Adaptor (BOA)*. The BOA is responsible for routing messages from the ORB to the skeleton interface (described next). In the future, Delphi will also provide a *Portable Object Adaptor (POA)*, which will offer more flexibility and customization of the server interface.

The skeleton is a class that is generated by the IDL2Pas compiler, just like the stub. The skeleton contains one or more classes that publish the server side CORBA interfaces. In the Delphi CORBA implementation, the skeleton doesn't contain any implementation details of the server-side interfaces. Instead, there is another file (also generated by IDL2Pas) that contains the classes which represent the functional details of the server. This is referred to as the IMPL file (short for Implementation).

The classes in the Impl file aren't tied to CORBA. The same implementation classes can be used to provide interfaces for CORBA, COM, or anything else.

When a message arrives on the server side, the ORB passes a message buffer to the BOA, which in turn, passes the buffer to the skeleton class. The skeleton un-marshals the data from the buffer and determines which method should be called in the IMPL file. After the IMPL file class method is called, the skeleton takes any return results and parameter values and marshals them into a buffer for transmission back to the client. The response buffer is handed back to the BOA and ORB, and is sent back to the client-side ORB.

OSAgent

CORBA objects need a way of locating one another. The OMG provides a solution for this with the Naming Service that is described in the CORBA specification. The Naming Service is a program that runs somewhere on the network. Server-side objects register with the Naming Service so that client applications have a method of locating specific objects. The Naming Service requires additional code on both the client and server sides. The location of the Naming Service process has to be known in advance before a client application can request a connection to a server object.

This is a fairly complicated way of letting clients and servers connect. VisiBroker has a utility that makes object location much easier than using the Naming Service. This program is called OSAgent. It isn't part of the CORBA specification. OSAgent is a proprietary utility that is only available with the Borland ORB. As long as the VisiBroker ORB is used within a CORBA implementation, the OSAgent is the preferred method of locating and binding to objects.

Before running CORBA applications created with VisiBroker, start OSAgent. When the server application starts, it will register itself with the agent. The client application will connect to the server by first contacting the OSAgent, requesting the address of the server, and then connecting directly to the server process.

Interfaces

All CORBA objects are described by their interfaces. This is pure object-oriented design. A server application will publish specific type declarations, interfaces, and methods that any client might call. Once these interfaces are published, they are immutable. That is, they should never change. To add additional features to an object, the best approach is to derive a new server from the old one and enhance the new object. That way, a new interface can be published without creating backward compatibility problems for deployed applications.

To describe interfaces, the OMG has published an *Interface Definition Language (IDL)*. IDL is programming language independent, but looks like C or Java. Each ORB vendor supplies an IDL compiler to translate IDL files into code for a specific language. The term *IDL compiler* is a misnomer. It doesn't actually compile the IDL file into an executable file. It is more of a code generator because the output is a set of source code files in the target language.

The OMG has specified language mappings for some languages like C++ and Java. A C++ ORB will have an IDL2CPP compiler. Java ORBs have an IDL2Java compiler.

The files generated by the IDL2 whatever compilers are the stub and skeleton class files that were discussed in the CORBA Architecture section. Delphi contains an IDL2Pas compiler that can be executed from the command line or launched through the IDE CORBA wizards.

Interface Definition Language (IDL)

IDL is an extensive subject. The Delphi 6 Enterprise CD-ROM contains a PDF document (`Delphi6\Doc\CORBA`) that describes the Object Pascal mapping for IDL. This document contains all the details about each data type, modules, inheritance, and user-defined types. This section highlights some of the notable aspects of IDL; however, to gain more insight into the details, refer to the mapping document.

There are a few rules that an IDL file must follow. The first is that an IDL file must have `.idl` as the file extension. This can be upper- or lowercase. Other file extensions won't be accepted.

The contents of an IDL file are relatively free flowing but do follow a certain structure. The interface descriptions are case sensitive. In C++ and Java, two interfaces named `Foo` and `foo` are considered different. However, in Delphi, they will create a problem because it will appear that the same interface has been created twice.

Comments in an IDL file are the same as C and C++. These are valid comments:

```
// This is a single line comment.
/* This is an example of a block comment
   that can be spread
   over several lines */
```

All IDL keywords must be written in lowercase, or the IDL2Pas compiler will reject them. Avoid using Delphi keywords if possible. The Delphi mapping specification states that all Delphi keywords will be prepended with the underscore (_) character. It is a good practice to avoid the use of Delphi reserved words.

An IDL file can include another IDL file with the `#include` pragma statement. This facilitates organizing large IDL files into smaller groups.

Basic Types

IDL has a number of basic types that can be used in interface descriptions. Table 19.1 shows a list of the basic types and shows how they are mapped to Object Pascal.

TABLE 19.1 Basic IDL Types

IDL Type	*Pascal Type*
boolean	Boolean
Char	Char
wchar	Wchar
octet	Byte
string	AnsiString
wstring	WideString
short	SmallInt
unsigned short	Word
long	Integer
unsigned long	Cardinal
long long	Int64
unsigned long long	Int64
float	Single
double	Double
long double	Extended
fixed	not implemented—no corresponding type

IDL doesn't have a type called int. Instead, the short, long, unsigned short, and unsigned long are used to specify the integer type. Characters correspond to the ISO Latin-1 type, which is equivalent to the ASCII table. The only exception is the NUL character (#0). C and C++ programmers asked the OMG to make that an illegal character because it represents the termination character in a string in those languages.

The implementation of Booleans is vendor specific. The Boolean mapping corresponds to a Boolean type in Delphi. The Any type is mapped to a Variant in Delphi.

User-Defined Types

You can define your own types in IDL. The syntax is similar to the way data structures are specified in C. Common user-defined types include aliases, enumerations, structures, arrays, and sequences.

Aliases

Aliases are used to give data types more meaningful names. For example, a year type can be created like this:

```
typedef short YearType;
```

Enumerations

IDL enumerations are mapped to an enumeration type in Delphi. An enumeration of some color values would look like this:

```
enum Color(red, white, blue, green, black);
```

Structures

Structures are similar to records in Pascal. Here's an example of a structure that represents a time value:

```
struct TimeOfDay {
   short hour;
   short minute;
   short seconds;
};
```

Arrays

Arrays can be single or multi dimensional. Specify an array with a `typedef`. Here are some examples:

```
typedef Color ColorArray[4];  // single dimensional array of the Color enum
typedef string StringArray[10][20];  //10 strings of max length 20
```

Sequences

Sequences are used heavily in IDL. They map to a variable length array in Delphi. Sequences can be bounded or unbounded.

```
typedef sequence<Color> Colors;
typedef sequence<long, 1000> NumSeq;
```

The first argument in the sequence specification is the base type of the variable array. The second argument is optional and specifies the length in a bounded sequence.

The most common use of sequences in CORBA programming is to pass database records between servers and clients. When the client application receives a sequence, it has to loop through it to extract all the fields in a record. Then it populates the user interface database controls with the information. MIDAS and CORBA can be used together to provide a friendlier approach.

Method Arguments

All arguments that are specified in a method have to be declared with one of three attributes. These attributes are `in`, `out`, or `inout`.

A parameter declared as an `in` type has its values set by the client. This is mapped as a `const` parameter in Delphi.

An `out` parameter has its value set by the server. It is mapped as a `var` parameter.

An `inout` parameter has its initial value set by the client. The server receives the data and changes it before returning the variable to the client. An `inout` parameter is mapped as a `var'` parameter in Delphi.

Modules

The keyword `module` is used to group interfaces and types. The module name will be used by IDL2Pas to name the Delphi unit. Interfaces and types defined within a module are local in scope. A module named `Foo` that contained an interface named `Bar` would be referenced outside the module with the module name and the interface name like this: `Foo::Bar`.

IDL doesn't support the idea of private or protected types and methods. All interfaces and methods are considered public. This makes sense when you consider that the IDL file represents the interfaces that the server exposes to the world. It wouldn't make sense to hide or protect something in this context.

One of the best ways to learn how to write IDL is to look at examples other people have written. The VisiBroker directory (`c:\Inprise`, by default) has a subdirectory called `IDL` that contains the IDL files for the various CORBA interfaces such as the ORB and the various services. These files are a good starting point and are full of examples of type declarations and interface definitions. These files contain examples of nested modules and references to outer scoped types.

With this basic understanding of IDL, several CORBA examples can be developed to demonstrate the power of distributed object computing. The remainder of the chapter covers the development of several CORBA servers and clients.

The Bank Example

CORBA has a traditional example that is the equivalent of `"Hello, world"` in C. It's known as the Bank Example and consists of a simple method call that returns a bank balance. We're going to add some additional capabilities such as a `deposit` and `withdraw` method. It would also be a good idea to prohibit overdrafts on the account, so an exception will be used to block drawing out more than the account contents. The IDL for this example is in Listing 19.1.

LISTING 19.1 `Bank.idl`

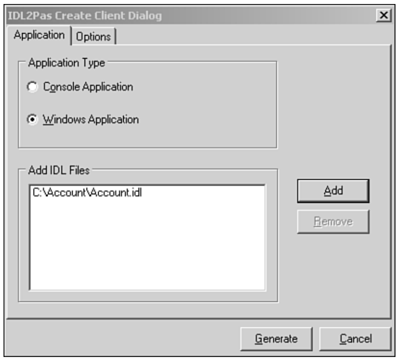
```
module Bank {
   exception WithdrawError {
      float current_balance;
   };

   interface Account {
      void deposit(in float amount);
      void withdraw(in float amount) raises (WithdrawError);
      float balance();
   };
};
```

The exception is declared with one data member. If the client attempts to withdraw more money than the account contains, the exception will be raised with the current account balance stored in the data member. Client applications can trap the exception and display a message to the user. In this case, a warning will be displayed along with the current balance.

The `deposit` and `withdraw` methods are equivalent to procedures, so they have a return type of void. Each takes one argument: the amount to add to or subtract from the account. The amount is a floating point number that will be mapped to a single in Delphi. Notice that the arguments for `deposit` and `withdraw` are declared as `in` parameters because the methods are passing the value from the client to the server. The `balance` method is a function that returns a floating point value that contains the current balance of the account.

The Delphi 6 IDE contains a set of wizards that make creating CORBA clients and servers pretty easy. We'll start by creating the server side of our application. To bring up the wizard, go to File, New, Other and select the CORBA tab in the dialog box. Then double-click on the CORBA server icon. The main wizard screen will appear as shown in Figure 19.2.



C:\Account\Account.idl

FIGURE 19.2
The CORBA Wizard in Delphi 6.

This window contains a list of all the IDL files that will be processed to generate the application. Initially, it is empty. To add one or more files, click the Add' button. This brings up a standard file open dialog. Change to the directory where the `Bank.idl` file is located, select that file, and then click OK. The `Bank.idl` file will be added to the list of files that will be processed by the IDL2Pas compiler. Because this is the only IDL file in the application, click on Generate to create the server application.

The IDL2Pas compiler will process the IDL file, and the wizard will create an application with the generated files. For server applications, four files are generated:

- `Bank_I.pas`—This file contains all the interfaces and type definitions.
- `Bank_C.pas`—This contains any user-defined types, exceptions, and client stub classes. In addition, all user-defined types and stub classes will have a helper class. The helper class assists in reading and writing data to the CORBA buffers.
- `Bank_S.pas`—This has the server-side skeleton class definitions.
- `Bank_Impl.pas`—This has a general class definition for an implementation on the server side. You can add code to the methods to perform the actions you want the server to complete. You don't have to use this file, but it's a handy starting point.

From this list of files, you can see that the client-side stub shown in the CORBA architecture is in the `Bank_C.pas` file, whereas the server-side skeleton is in `Bank_S.pas`. A sample implementation for the server side is stored in the `Bank_Impl.pas` file.

Listing 19.2 shows the interface definitions for the application. There is only one interface named `Account`, and it contains the three methods that were declared in the IDL file.

LISTING 19.2 `Bank_I.pas`

```
unit Bank_i;
 interface

uses
  CORBA;

type
  Account = interface;

  Account = interface
    ['{99FCA96D-77B2-4A99-7677-E1E0C32F8C67}']
    procedure deposit (const amount : Single);
    procedure withdraw (const amount : Single);
    function  balance : Single;
  end;

implementation

initialization

end.
```

Listing 19.3 shows the source for the `Bank_C.pas` file. This file contains the declaration of the `Overdrawn` exception. It is derived from a class called `UserException` that is also defined in that file.

LISTING 19.3 The `Bank_C.pas` File

```
unit Bank_c;

interface

uses
  CORBA, Bank_i;

type
  EWithdrawError = class;
  TAccountHelper = class;
  TAccountStub = class;

  EWithdrawError = class(UserException)
  private
    Fcurrent_balance : Single;
```

LISTING 19.3 Continued

```
  protected
    function  _get_current_balance : Single; virtual;
  public
    property  current_balance : Single read _get_current_balance;
    constructor Create; overload;
    constructor Create(const current_balance : Single); overload;
    procedure Copy(const _Input : InputStream); override;
    procedure WriteExceptionInfo(var _Output : OutputStream); override;
  end;

  TAccountHelper = class
    class procedure Insert (var _A: CORBA.Any; const _Value : Bank_i.Account);
    class function  Extract(var _A: CORBA.Any) : Bank_i.Account;
    class function  TypeCode     : CORBA.TypeCode;
    class function  RepositoryId : string;
    class function  Read (const _Input  : CORBA.InputStream) : Bank_i.Account;
    class procedure Write(const _Output : CORBA.OutputStream;
➥          const _Value : Bank_i.Account);
    class function  Narrow(const _Obj   : CORBA.CORBAObject; _
➥          IsA : Boolean = False) : Bank_i.Account;
    class function  Bind(const _InstanceName : string = ''; _
➥          HostName : string = '') : Bank_i.Account; overload;
    class function  Bind(_Options : BindOptions;
➥          const _InstanceName : string = ''; _HostName: string = '') :
➥          Bank_i.Account; overload;
  end;

  TAccountStub = class(CORBA.TCORBAObject, Bank_i.Account)
  public
    procedure deposit ( const amount : Single); virtual;
    procedure withdraw ( const amount : Single); virtual;
    function  balance : Single; virtual;
  end;

implementation

var

  WithdrawErrorDesc : PExceptionDescription;

function  EWithdrawError._get_current_balance : Single;
begin
  Result := Fcurrent_balance;
end;
```

Listing 19.3 Continued

```
constructor EWithdrawError.Create;
begin
  inherited Create;
end;

constructor EWithdrawError.Create(const current_balance : Single);
begin
  inherited Create;
  Fcurrent_balance := current_balance;
end;

procedure EWithdrawError.Copy(const _Input: InputStream);
begin
  _Input.ReadFloat(Fcurrent_balance);
end;

procedure EWithdrawError.WriteExceptionInfo(var _Output : OutputStream);
begin
  _Output.WriteString('IDL:Bank/WithdrawError:1.0');
  _Output.WriteFloat(Fcurrent_balance);
end;

function  WithdrawError_Factory: PExceptionProxy; cdecl;
begin
  with Bank_c.EWithdrawError.Create() do Result := Proxy;
end;

class procedure TAccountHelper.Insert(var _A : CORBA.Any;
➥    const _Value : Bank_i.Account);
begin
  _A := Orb.MakeObjectRef( TAccountHelper.TypeCode, _
➥        Value as CORBA.CORBAObject);
end;

class function TAccountHelper.Extract(var _A : CORBA.Any): Bank_i.Account;
var
  _obj : Corba.CorbaObject;
begin
  _obj := Orb.GetObjectRef(_A);
  Result := TAccountHelper.Narrow(_obj, True);
end;

class function TAccountHelper.TypeCode : CORBA.TypeCode;
```

LISTING 19.3 Continued

```
begin
  Result := ORB.CreateInterfaceTC(RepositoryId, 'Account');
end;

class function TAccountHelper.RepositoryId : string;
begin
  Result := 'IDL:Bank/Account:1.0';
end;

class function TAccountHelper.Read(const _Input : CORBA.InputStream)
➥    : Bank_i.Account;
var
  _Obj : CORBA.CORBAObject;
begin
  _Input.ReadObject(_Obj);
  Result := Narrow(_Obj, True)
end;

class procedure TAccountHelper.Write(const _Output : CORBA.OutputStream;
➥    const _Value : Bank_i.Account);
begin
  _Output.WriteObject(_Value as CORBA.CORBAObject);
end;

class function TAccountHelper.Narrow(const _Obj : CORBA.CORBAObject; _
➥    IsA : Boolean) : Bank_i.Account;
begin
  Result := nil;
  if (_Obj = nil) or (_Obj.QueryInterface(Bank_i.Account, Result) = 0) then
    exit;
  if _IsA and _Obj._IsA(RepositoryId) then
    Result := TAccountStub.Create(_Obj);
end;

class function TAccountHelper.Bind(const _InstanceName : string = ''; _
➥    HostName: string = '') : Bank_i.Account;
begin
  Result := Narrow(ORB.bind(RepositoryId, _InstanceName, _HostName), True);
end;

class function TAccountHelper.Bind(_Options : BindOptions;
➥   const_InstanceName : string = ''; HostName : string = '') :
➥    Bank_i.Account;
```

LISTING 19.3 Continued

```
begin
  Result := Narrow(ORB.bind(RepositoryId, _Options, _InstanceName, _
➥      HostName), True);
end;

procedure TAccountStub.deposit ( const amount : Single);
var
  _Output: CORBA.OutputStream;
  _Input : CORBA.InputStream;
begin
  inherited _CreateRequest('deposit',True, _Output);
  _Output.WriteFloat(amount);
  inherited _Invoke(_Output, _Input);
end;

procedure TAccountStub.withdraw ( const amount : Single);
var
  _Output: CORBA.OutputStream;
  _Input : CORBA.InputStream;
begin
  inherited _CreateRequest('withdraw',True, _Output);
  _Output.WriteFloat(amount);
  inherited _Invoke(_Output, _Input);
end;

function  TAccountStub.balance : Single;
var
  _Output: CORBA.OutputStream;
  _Input : CORBA.InputStream;
begin
  inherited _CreateRequest('balance',True, _Output);
  inherited _Invoke(_Output, _Input);
  _Input.ReadFloat(Result);
end;

initialization

Bank_c.WithdrawErrorDesc := RegisterUserException('WithdrawError',
➥    'IDL:Bank/WithdrawError:1.0', @Bank_c.WithdrawError_Factory);

finalization

UnRegisterUserException(Bank_c.WithdrawErrorDesc);

end.
```

Listing 19.4 shows the definition for the `Account` implementation class. This class isn't tied to CORBA, so it can be reused for other applications or interfaces. The `Account` class contains the methods that were declared in the `Bank.idl` file. Code has been added to the `TAccount` methods to implement the full server.

LISTING 19.4 The Implementation Class for the Bank Server

```
unit Bank_impl;

interface

uses
  SysUtils, CORBA, Bank_i, Bank_c;

type
  TAccount = class;

 unit Bank_impl;

interface

uses
  SysUtils, CORBA, Bank_i, Bank_c;

type

TAccount = class(TInterfacedObject, Bank_i.Account)
  protected
    _balance : Single;
  public
    constructor Create;
    procedure deposit (const amount : Single);
    procedure withdraw (const amount : Single);
    function  balance : Single;
  end;


implementation

constructor TAccount.Create;
begin
  inherited;
  _balance := random(10000);
end;
```

LISTING 19.4 Continued

```
procedure TAccount.deposit(const amount : Single);
begin
  if amount > 0 then
    _balance := _balance + amount;
end;

procedure TAccount.withdraw(const amount : Single);
begin
  if amount < _balance then
    _balance := _balance - amount
  else
    raise EWithdrawError.Create(_balance);
end;

function  TAccount.balance : Single;
begin
  result := _balance;
end;

initialization
   randomize;

end.
```

The `TAccount` object is derived from `TInterfacedObject`, so it will be reference counted automatically. It implements the `Account` interface that was contained in the `Bank_I.pas` file. The deposit method does a simple check to make sure that the user hasn't passed a negative number to the application. The `withdraw` method performs a check on the amount passed by the client. If it is less than the balance, the exception is raised with the current account balance as the exception argument. The client can process the exception to display information to the end user. The `balance` method returns the current balance on the server.

Listing 19.5 shows the stub class that is used as the proxy object for the client application. Like the server skeleton, it has the three methods defined in the `Account` interface in the IDL file.

LISTING 19.5 Client-Side Stub Class

```
  TAccountStub = class(CORBA.TCORBAObject, Bank_i.Account)  public

  public

  public
```

LISTING 19.5 Continued

```
    procedure deposit ( const amount : Single); virtual;
    procedure withdraw ( const amount : Single); virtual;
    function  balance : Single; virtual;
  end;
```

Listing 19.6 shows the `deposit()` method in detail. Two CORBA buffer streams are declared as local variables. The `CreateRequest()` method is a call into the ORB that asks for a valid output buffer so that information can be written into it. The stub passes the name of the method that will be called on the server side and specifies whether to wait for the server to complete its task before continuing. This is referred to as a one-way call or a two-way call.

LISTING 19.6 The Stub Class Deposit Method

```
procedure TAccountStub.deposit(const amount : Single);
var
  _Output: CORBA.OutputStream;
  _Input : CORBA.InputStream;
begin
  inherited _CreateRequest('deposit', True, Output);
  _Output.WriteFloat(amount);
  inherited _Invoke(Output, Input);
end;
```

The next step is to write any data values that need to be passed to the server into the output buffer. In this case, the amount to deposit is stored in the buffer. The final call is the `Invoke` method. This is another call to the ORB that sends the request and output buffer to the server side. After the server has finished processing, execution continues on the client side. In situations where the method call is a function (such as the `balance` method), the input buffer contains the returned result. IDL2Pas would have generated the code to read the values from the input buffer. However, in this case it was a call to a procedure, so no return value is present.

All of the stub code is generated automatically by IDL2Pas, so you should never have to edit it yourself. However, it is helpful to understand what this generated code does.

The final part of the code for the application is in the client GUI. The client will contain three push buttons, two edit controls, and one label control as shown in Figure 19.3. All CORBA interface variables are declared as interface types. In this case, the Account interface is declared as type `Account`. This establishes a variable from the type defined in the `Bank_i.pas` file that has the three methods defined in the `Bank.idl` file. The other benefit to having interface type variables is the automatic reference counting that takes place behind the scenes. All the CORBA objects should be reference counted. The IDL2Pas compiler automatically generates the code to facilitate this.

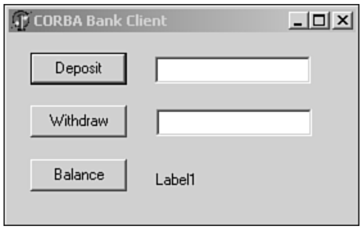


FIGURE 19.3
The CORBA Client Application.

The most interesting part of the code is the Withdraw `OnClick` event. Listing 19.7 contains the client-side source. The call to the `Withdraw()` method checks to make sure that the client isn't attempting to take more than the account holds. If this is the case, an exception is raised. Notice that raising an exception in CORBA is identical to raising an exception in Delphi. The Delphi exception gets translated to a CORBA exception automatically.

LISTING 19.7 The Client Source

```
unit ClientMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Corba, Bank_c, Bank_i, StdCtrls;

type
  TForm1 = class(TForm)
    btnDeposit: TButton;
    btnWithdraw: TButton;
    btnBalance: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    procedure btnDepositClick(Sender: TObject);
    procedure btnWithdrawClick(Sender: TObject);
    procedure btnBalanceClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
  { private declarations }
  protected
    Acct : Account;
    procedure InitCorba;
  { protected declarations }
```

LISTING 19.7 Continued

```
  public
  { public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.InitCorba;
begin
  CorbaInitialize;
  // Bind to the Corba server
  Acct := TAccountHelper.bind;
end;

procedure TForm1.btnDepositClick(Sender: TObject);
begin
  Acct.deposit(StrToFloat(Edit1.text));
end;

procedure TForm1.btnWithdrawClick(Sender: TObject);
begin
  try
    Acct.withdraw(StrToFloat(Edit2.Text));
  except
    on e: EWithdrawError do
      ShowMessage('Withdraw Error. The balance = ' +
      FormatFloat('$##,##0.00', E.current_balance));
  end;
end;

procedure TForm1.btnBalanceClick(Sender: TObject);
begin
  label1.caption := FormatFloat('Balance = $##,##0.00', acct.balance);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 InitCorba;
end;

end.
```

After the client and server applications are compiled, OSAgent needs to be started. On a Windows NT machine, the VisiBroker OSAgent can be installed as a service. On other operating systems, it has to be started manually. To start OSAgent manually on any MS Windows platform, choose the Start, Run menu and type **`OSAgent -C`**. This starts OSAgent in console mode. The agent will appear as an icon on the taskbar.

The server application is started next, followed by the client. The client GUI is shown in Figure 19.3. It has three buttons, two edit boxes, and a label control to display the balance. Click the Balance button to get the initial value from the server. Then add some money. Click Balance again to refresh the value on the client side. (Balance can also be called as part of the `deposit` and `withdraw` methods to automatically update the client.) After trying a few values, try to withdraw more than the balance. You should see the exception message.

Complex Data Types

This next example won't do much as a practical application. However, it illustrates how to use some of the complex data types that are available in CORBA IDL. Listing 19.8 shows the IDL for the *Advanced Data Types (ADTs)*.

LISTING 19.8 `ADT.idl`

```
// ADT IDL file
//
// Demonstrates various data structures in IDL
//

// use an alias for string types

typedef string Identifier;

enum EnumType
{
  first,
  second,
  third
};

struct StructType
{
  short s;
  long l;
  Identifier i;
};
```

LISTING 19.8 Continued

```
const unsigned long ArraySize = 3;

typedef StructType StructArray[ArraySize];

typedef sequence<StructType> StructSequence;


interface ADT
{

  void Test1(in Identifier st, in EnumType myEnum, inout StructType myStruct);

  void Test2(out StructType myStruct, in StructArray myStructArray,
➥       out StructSequence myStructSeq);

};
```

The first data type shows the use of an alias to remap the string type. All strings in this example will be of type `Identifier`. The `EnumType` consists of three values: `first`, `second`, and `third`.

The `StructType` is similar to a record in Pascal. This data structure consists of a short, a long, and a string (mapped to the Identifier alias). The `ArraySize` is mapped as a constant.

The next two items in the IDL file declare types based on the previous definitions. The `StructArray` is declared as an array of three elements maximum (zero based). A sequence is a dynamic array. The last `typedef` declares a sequence of `StructTypes`.

Finally, the ADT interface is defined with two methods: `Test1` and `Test2`. The arguments to these methods are designed to show the different directions data can take. `In` parameters are created and initialized on the client side. `Out` parameters are created and initialized on the server side. `InOut` parameters are created and initialized on the client side, but typically are modified on the server side and returned to the client with new values in the data members.

Listing 19.9 shows the `ADT_I.pas` interface file. Notice that the `typedefs` are defined in this file. Also an interface is created for the `StructType`. All complex types are mapped to objects in Object Pascal with the appropriate `get` and `set` methods and a Helper class to facilitate marshaling the data in a CORBA buffer.

LISTING 19.9 The `ADT_I.pas` File

```
unit adt_i;

interface
```

LISTING 19.9 Continued

```
uses
  CORBA;

type

  EnumType = (first, second, third);

const
  { (Do not edit the values assigned to these constants.) }

  ArraySize : Cardinal = 3;

type
  StructType = interface;
  ADT = interface;

  Identifier = AnsiString;

  StructArray = array[0..2] of adt_i.StructType;

  StructSequence = array of adt_i.StructType;

  StructType = interface
    ['{B4A1845D-4DB0-9B2E-A2E3-001F2D6B8C81}']
    function  _get_s : SmallInt;
    procedure _set_s (const s : SmallInt);
    function  _get_l : Integer;
    procedure _set_l (const l : Integer);
    function  _get_i : adt_i.Identifier;
    procedure _set_i (const i : adt_i.Identifier);
    property  s : SmallInt read _get_s write _set_s;
    property  l : Integer read _get_l write _set_l;
    property  i : adt_i.Identifier read _get_i write _set_i;
  end;

  ADT = interface
    ['{203B9E07-735F-2980-CB02-353A7C6A5B68}']
    procedure Test1 (const st : adt_i.Identifier;
                     const myEnum : adt_i.EnumType;
                     var   myStruct : adt_i.StructType);
    procedure Test2 (out   myStruct : adt_i.StructType;
                     const myStructArray : adt_i.StructArray;
                     out   myStructSeq : adt_i.StructSequence);
  end;
```

LISTING 19.9 Continued

```
implementation

initialization

end.
```

Listing 19.10 shows the implementation of the server side. When you read the method parameter lists in IDL, the direction is applicable to the server, or receiving side. So an `out` parameter means that it is out relative to the server. An `in` parameter is in relative to the server, and so on.

All the `out` parameters on the server side need to have their data structures created and initialized before the data can be passed back to the client. Any parameter defined as a `const` or `var` parameter will have an existing data structure associated with it.

LISTING 19.10 The ADT Implementation File for the Server

```
unit adt_impl;

interface

uses
  SysUtils, CORBA, adt_i, adt_c;

type
  TADT = class;

  TADT = class(TInterfacedObject, adt_i.ADT)
  public
    constructor Create;
    procedure Test1 ( const st : adt_i.Identifier;
                      const myEnum : adt_i.EnumType;
                      var   myStruct : adt_i.StructType);
    procedure Test2 ( out   myStruct : adt_i.StructType;
                      const myStructArray : adt_i.StructArray;
                      out   myStructSeq : adt_i.StructSequence);
  end;

implementation

uses ServerMain;

constructor TADT.Create;
```

LISTING 19.10 Continued

```
begin
  inherited;
end;

procedure TADT.Test1 ( const st : adt_i.Identifier;
                       const myEnum : adt_i.EnumType;
                       var   myStruct : adt_i.StructType);
begin
  Form1.Memo1.Lines.Add('String from Client : ' + st);

  case myEnum of
    first : Form1.Memo1.Lines.Add('Enum value is "first"');
    second: Form1.Memo1.Lines.Add('Enum value is "second"');
    third:  Form1.Memo1.Lines.Add('Enum value is "third"');
  end;

  Form1.Memo1.Lines.Add(Format('myStruct.s = %d', [myStruct.s]));
  Form1.Memo1.Lines.Add(Format('myStruct.l = %d', [myStruct.l]));
  Form1.Memo1.Lines.Add(Format('myStruct.i = %s', [myStruct.i]));

  myStruct.s := 10;
  myStruct.l := 1000;
  myStruct.i := 'This is the return string from the Server';
end;

procedure TADT.Test2 ( out   myStruct : adt_i.StructType;
                       const myStructArray : adt_i.StructArray;
                       out   myStructSeq : adt_i.StructSequence);
var
  k : integer;
  tempSeq : StructSequence;
begin
  myStruct := TStructType.Create(20, 2000,
➥      'Hello from the server structType Test 2');

  for k := 0 to ArraySize - 1 do
    With Form1.Memo1.Lines do
    begin
      Add(Format('myStructArray[%d].s = %d', [k, myStructArray[k].s]));
      Add(Format('myStructArray[%d].l = %d', [k, myStructArray[k].l]));
      Add(Format('myStructArray[%d].i = %s', [k, myStructArray[k].i]));
    end;

  SetLength(tempSeq, 2);
```

LISTING 19.10 Continued

```
  for k := 0 to 1 do
    tempSeq[k] := TStructType.Create(k + 100, k + 1000, Format('k = %d', [k]));
  myStructSeq := tempSeq;
end;


initialization

end.
```

The client application user interface has two buttons and a memo control. Each button is mapped to one of the ADT test methods. The results of the calls are written to the memo control. Listing 19.11 shows the client file.

LISTING 19.11 The ADT Client Side

```
unit ClientMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Corba, adt_c, adt_i, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
  { private declarations }
  protected
    myADT : ADT;
    procedure InitCorba;
  { protected declarations }
  public
  { public declarations }
  end;

var
  Form1: TForm1;
```

LISTING 19.11 Continued

```
implementation

{$R *.DFM}

procedure TForm1.InitCorba;
begin
  CorbaInitialize;
  myADT := TADTHelper.bind;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  initCorba;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  temp : StructType;
begin
  temp := TStructType.Create(50, 500, 'This is the client struct in Test 1');
  myADT.Test1('Hello from the Test1 Client', first, temp);

  with Memo1.Lines do
  begin
    Add('Response from server inout struc var:');
    Add(Format('myStruct.s = %d', [temp.s]));
    Add(Format('myStruct.l = %d', [temp.l]));
    Add(Format('myStruct.i = %s', [temp.i]));
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  I: Integer;
  temp : StructType;
  tempSeq : StructSequence;
  tempArray : StructArray;
begin
   temp := TStructType.Create(0,0,'test');
   SetLength(tempSeq, 2);

   for I := 0 to ArraySize -1 do
     tempArray[I] := TStructType.Create(200 + I, 2000 + I,
➥         Format('Stuct %d in Array', [I]) );
```

LISTING 19.11 Continued

```
   myADT.Test2(temp, tempArray, tempSeq);

   with Memo1.Lines do
   begin
     Add(Format('struct.s = %d', [temp.s]) );
     Add(Format('struct.l = %d', [temp.l]) );
     Add(Format('struct.i = %s', [temp.i]) );
   end;

   for I := 0 to 1 do
   with Memo1.Lines do
   begin
     Add( Format('tempSeq[%d].s = %d', [I, tempSeq[I].s]) );
     Add( Format('tempSeq[%d].l = %d', [I, tempSeq[I].l]) );
     Add( Format('tempSeq[%d].i = %s', [I, tempSeq[I].i]) );
   end;
end;

end.
```

To run this example, compile the code and make sure that the OSAgent is running. When you click on either of the test buttons on the data, structures are exchanged between the client and server. The data that is received on each side is written to the respective memo control in the application's window.

Delphi, CORBA, and Enterprise Java Beans (EJBs)

This section shows you how to make a Delphi CORBA application connect to EJBs that are deployed under the Borland Application Server. To construct and deploy the EJB for this demo, you'll need Borland JBuilder 5 and Borland Application Server 4.51. Both of these products are available as a free trial download edition from the Borland Web site (`www.borland.com`).

A Crash Course in EJBs for Delphi Programmers

Several years ago, Sun Microsystems came out with their J2EE platform. This was an enhancement to the Java environment to add enterprise level distributed object computing. The specification for J2EE is fairly complex, but from an application developer's point of view, it can be broken down into a few straightforward concepts.

One key piece of the J2EE platform is an *Enterprise Java Bean (EJB)*. An EJB is (usually) a small, portable and scaleable object that is designed to do a specific job. The idea is that many

EJBs can be scattered around the enterprise to perform various functions. At some central point, an application is deployed that will contact an EJB only when it needs the functionality that EJB provides.

An EJB Is a Specialized Component

In terms of Delphi, think of an EJB as a component. One example of an EJB would be a component that connects to a database and provides records to any application that requests them. Another EJB might perform a calculation based on information supplied to it, such as calculating sales tax for a purchase.

EJBs Live Within a Container

In Delphi, components are put into a package and are installed into the IDE. The IDE manages the components on the palette. The hooks are there to create the component when you drop one on a form. If you delete the component from the form, it is destroyed within the IDE.

A similar approach is taken for EJBs only not through the Delphi IDE. The J2EE specification describes an entity known as the *EJB container*. The container is the host for all EJBs. This is a similar concept to the way the IDE manages components within Delphi. The container manages the process of creating and destroying EJBs.

EJBs Have Predefined APIs

Borland AppServer has an EJB container embedded within itself. Just like the Delphi IDE and its components, the EJB container and all EJBs must have a predefined set of APIs that let the EJB live within the container. The EJB developer adds additional methods to the EJB to give it specific functionality. However, the predefined APIs must be there for the EJB to be managed correctly by the container.

In addition to creating and destroying the EJB, there are specific APIs for message routing and callbacks to an EJB. The container also performs numerous other features described in the specification, but those details are beyond the scope of this book.

The Home and Remote Interfaces

As part of the predefined API set, all EJBs must have two interfaces. One is called the `Home` interface, and the other is the `Remote` interface. The `Home` interface is the initial method that an application calls to get an instance of the EJB. The `Home` interface is a factory that creates instances of the `Remote` interface and hands them back to the calling application.

The `Remote` interface has all the methods that the calling program wants to use. These are equivalent to the interfaces declared in an IDL file. So the process is that a client application will call the EJB `Home` interface to get an instance of the `Remote` interface. Once it has the `Remote` interface, the client can call any method published by that interface.

Types of EJBs

All EJBs can be divided into one of two groups:

- Session beans
- Entity beans

A session bean is (typically) a stateless EJB. *Stateless* means that between calls, the session bean doesn't store any information about the calling application. It is said to be nonpersistent. It doesn't keep track of where the client might be in a calling sequence, so it doesn't have the equivalent of a state machine. It is possible to have a stateful session bean, but the developer has to write all the logic to implement that.

An entity bean generally wraps a database record. This type of bean is said to have state because the information it processes (the database record) is stored between calls.

There is another key difference between a session and entity bean. When a client connects to a session bean, one instance of the session bean is created specifically for the calling client. If another client calls the session bean, another instance is created. So each client will get its own instance of a session bean.

When a client calls an entity bean, one instance of the entity bean is created. If another client calls the entity bean, it will share the same entity bean instance. Each entity bean is managed by the container in a connection pool.

Configuring JBuilder 5 for EJB Development

The easiest way to create EJBs is with Borland's JBuilder 5. To connect Delphi to an EJB, the EJB needs to be deployed with Borland's Application Server (version 4.51 or higher). Both JBuilder 5 and Borland AppServer are available on the Borland Web site as trial edition downloads. Typically, AppServer should be installed, followed by JBuilder.

Before starting JBuilder, it is a good idea to set up a projects directory for all your JBuilder applications. Typically, this will be something similar to `c:\MyProjects`.

When you start JBuilder 5 and try to create an EJB using the wizards built into the IDE, you might see them grayed out. If this is the case, it is because JBuilder needs to be configured to point to the AppServer. This is a configuration setting in a JBuilder dialog box.

To configure JBuilder 5 for EJB support,

1. Start JBuilder 5, go to the Tools menu, and select Enterprise Setup. This opens a dialog box to set the CORBA configuration.
2. On the CORBA tab, select VisiBroker. JBuilder ships with VisiBroker for Java.
3. Click on the Edit button. The Edit Configuration dialog box appears. Enter the path to the ORB. This is where IDL2Java resides (typically, `c:\Borland\AppServer\bin`).
4. Select that Application server tab. This is used to tell JBuilder which Application Server to use. Select BAS 4.5 and make sure that it is pointing to the AppServer directory (that is, `c:\Borland\AppServer`).
5. Under Projects, Default Project Proprieties, select the Servers tab. Then make sure that the Borland Application Server is selected. If not, click on the ellipse button and add it to the configuration.

That should configure JBuilder for EJBs. Now we can create our first EJB.

Building a Simple `"Hello, world"` EJB

Borrowing from the C world, the EJB application we'll build is one that returns the string `"Hello, world"`. This example guides you through the process of making an EJB. More complex EJBs can be developed by following the same process and adding more methods to the Remote interface.

First start Borland AppServer, and then JBuilder. When you become more proficient in Java and JBuilder, you can eventually develop and test your EJBs totally within the JBuilder environment. In this section, we'll develop the EJB and deploy it to the Borland AppServer. Then we'll make a Delphi client that will connect to the EJB. This is a more realistic scenario for real-world development and deployment.

To build the `"Hello, world"` EJB,

1. Close down all projects within JBuilder. Then choose File, New Project. Give this project the name `"HelloWorld"`. This will also be the name of the project file when JBuilder saves it to disk.
2. Next we need to add an EJB Group. Choose File, New, Enterprise and select the Empty EJB Group icon. When prompted for a name, give it **`HelloGroup`**. Notice that there is an edit control that specifies the name of the jar file that this application will be built into. A jar file is similar to a zip file. You archive all the Java byte code files into a jar file so that you only have one file to deploy. In this case, rename the jar file to `HelloWorld.jar`.
3. Next add a new EJB by selecting File, New, Enterprise and select Enterprise Java Bean. When prompted for a name, enter **`HelloBean`**. JBuilder 5 will automatically create the bean and all the required interfaces.

4. Select the `HelloBean.java` file in the project window and click on the Source tab. Make sure that your source code resembles Listing 19.12.

LISTING 19.12 The JavaBean Source

```
package helloworld;
import java.rmi.*;
import javax.ejb.*;
import java.lang.String;
public class HelloBean implements SessionBean
{
   private SessionContext sessionContext;
   public void ejbCreate()
   {
   }
   public void ejbRemove() throws RemoteException
   {
   }
   public void ejbActivate() throws RemoteException
   {
   }
   public void ejbPassivate() throws RemoteException
   {
   }
   public void setSessionContext(SessionContext sessionContext) throws
   RemoteException
   {
      this.sessionContext = sessionContext;
   }
   public String sayHello() {
      return "Hello, world";
   }
}
```

5. You need to enter the last method in the code block (`sayHello()`). This method returns a string, so it must include the `java.lang.String` package as shown near the top of the file.
6. Now we need to expose the `sayHello()` method through the `Remote` interface. To do that, select the Bean tab at the bottom of the code window. Then select the Methods tab at the bottom of the Bean tab window. You'll see the `sayHello()` method listed with an unchecked check box next to it. Check the box. That exposes the method through the Remote interface. To verify this, double-click on the `Hello.java` file in the project window. This brings up the source for the Remote interface. Notice that `sayHello()` is there now.
7. Save your work and build the project. You should have no errors.

Building a Client Test Application in JBuilder

JBuilder lets you build a client-side Java application to test your EJB. To do this,

1. Choose File, New, Enterprise and select EJB Test Client. Name this `HelloTestClient1.java`.
2. JBuilder will automatically create the file. Go to the bottom of it, and you will see a main program. Make yours resemble Listing 19.13.

LISTING 19.13 EJB Java Test Client Application

```
public static void main(String[] args) {
   HelloTestClient1 client = new HelloTestClient1();
   client.create();   //add these two lines
   client.sayHello();
   // Use the client object to call one of the Home interface wrappers
   // above, to create a Remote interface reference to the bean.
   // If the return value is of the Remote interface type, you can use it
   // to access the remote interface methods. You can also just use the
   // client object to call the Remote interface wrappers.
}
```

You add the `create()` and `sayHello()` methods to the main program.

Building the Client and Testing the EJB

Now build the test client. Follow these steps:

1. Run the EJB by selecting the HelloGroup entity in the project window and right-clicking on it. Then choose Run. Soon you should see messages in the JBuilder message pane indicating that the EJB is running. This might take 20 or 30 seconds depending on the speed and memory of your machine. Remember, Java is a resource hog.
2. Select the client application and right-click on it. Choose Run from the menu. You'll see it start up and eventually print `"Hello, world"` to the message window. This means that our EJB works correctly.
3. You can stop the EJB group by clicking on the red Stop button at the bottom of the message window.

Deploying the EJB to AppServer

To deploy the EJB to AppServer, follow these steps:

1. Select Tools, EJB Deployment. Follow the wizard, and it will deploy the EJB.
2. Once the EJB is deployed, AppServer will automatically start it. Click on the Next button until you reach step 4.

3. In step 4 of the wizard, you must select an EJB container. Make sure that Borland AppServer is running. Then click on the Add EJB Container button, and you'll see the AppServer container. Select it and click OK. Then continue with the wizard until it completes.

Generating the SIDL File

Borland has developed a proprietary technique to remap EJBs to an AppServer interface called *Simplified IDL (SIDL)*. This remapping makes sure that older CORBA applications can call EJBs by using the CORBA 2.1 standard (or higher). There is a tool that ships with AppServer called the SIDL compiler that can take a Remote interface and generate conventional IDL files.

Borland provides a free plug-in for JBuilder that facilitates converting the EJB interfaces to IDL with the SIDL compiler. The plug-in can be found on the CD-ROM in the same directory as the source code for this chapter. The plug-in is a Java jar file called `otSIDL.jar`. Copy it to the `c:\ JBuilder5\lib\ext` directory (or the equivalent path where you installed JBuilder). You'll have to restart JBuilder to activate the tool.

When the JBuilder IDE comes up, choose Tools, IDE Options, and you'll see a dialog box with a tab for SIDL. Select the SIDL tab and specify an output directory. In this case, enter **`c:\MyProjects\HelloWorld`** (or where ever you stored the HelloWorld project).

This tool adds a pop-up menu to the EJB Remote interface.

Select the `HelloHome.java` file from the project list and right-click on it. You'll see a menu item called Generate Simplified IDL. Choose that, and it will run the SIDL compiler. The output will be in the `classes` directory for the project.

Developing the EJB Client in Delphi

Now that we have a complete EJB, we can take the IDL file generated by the SIDL compiler and create a Delphi CORBA client to talk to the EJB. If you look under the EJB project's `classes` directory, you'll see a file called `HelloHome.idl`. You will need this and a copy of the `sidl.idl` file that is found on the `Delphi6\Demos\Corba\Idl2pas\EJB\EuroConverter` directory. Put those two files in a new directory. Then, follow these steps:

1. Start Delphi and choose File, New, Other, CORBA, CORBA Client Application.
2. Add the `HelloHome.idl` file to the list of files that will be processed. You don't need to add the `sidl.idl` file because it is included automatically in `HelloHome.idl` via the `include` pragma.
3. The wizard will create a new CORBA client. Save the project and call it `HelloClient`. Many file windows will be displayed after the wizard runs. Other than the `Unit1.pas` file, the only two that need to be displayed are `HelloHome_HelloWorld_i.pas` and `HelloHome_HelloWorld_c.pas`. All the rest can be closed.

4. On the main form, drop a button and label control. Make your application resemble Figure 19.4.

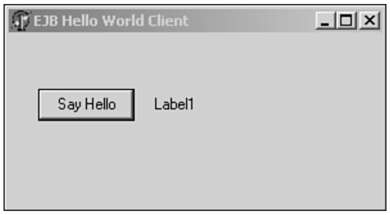


FIGURE 19.4
The EJB Delphi Client.

5. In the form's `OnCreate()` method, enter **`initCorba`**.
6. Modify the `initCorba()` method to resemble the code block shown in Listing 19.14. You'll have to add two variables to the class definition. One is for the `Home` interface, and the other one is for the `Remote` interface. The `Home` interface is a factory that creates `Remote` interfaces. Once you have an instance of the `Remote` interface, you can call the methods on the EJB. So the `initCorba()` method will contain the code that binds to the `Home` interface and generates a `Remote` interface object.
7. Add a button `OnClick` event and make it resemble the button `OnClick` code block shown in Listing 19.14.
8. Build the client application. See the note if you get errors.

LISTING 19.14 The EJB Client Main Application File

```
unit ClientMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Corba, HelloHome_c, HelloHome_helloworld_c, HelloHome_helloworld_i,
  HelloHome_i, HelloHome_sidl_javax_ejb_c, HelloHome_sidl_javax_ejb_i,
  HelloHome_sidl_java_lang_c, HelloHome_sidl_java_lang_i,
  HelloHome_sidl_java_math_c, HelloHome_sidl_java_math_i,
  HelloHome_sidl_java_sql_c, HelloHome_sidl_java_sql_i,
  HelloHome_sidl_java_util_c, HelloHome_sidl_java_util_i,
  StdCtrls;

type
  TForm1 = class(TForm)
```

LISTING 19.14 Continued

```
    Button1: TButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
  { private declarations }
  protected
    myHome : HelloHome;
    myRemote : Hello;
    procedure InitCorba;
  { protected declarations }
  public
  { public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.InitCorba;
begin
   CorbaInitialize;

   myHome := THelloHomeHelper.Bind;
   myRemote := myHome._create;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
   initCorba;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
   Label1.Caption := myRemote.sayHello;
end;

end.
```

NOTE

You'll encounter two possible errors with the original Delphi 6 distribution. These errors were fixed in the Service Pack 1 update. The first error is that the compiler will complain about a unit not being included. It will point you to the particular unit name. When it does, just add the unit name to the uses clause in the file with the error.

The second error occurs at runtime. This is related to the `Home` interface `create()` method. *Create* was originally on the reserved word list for IDL2Pas. So when it encounters that word, it puts an underscore in front of it. When the EJB gets a request for a method called `_create()`, it generates an exception because it doesn't publish that method. It publishes a method called `create()`.

To fix this, go to the `THelloHomeStub._create()` method in `HelloHome_helloworld_c.pas` (the code snippet follows). The first argument in the `_CreateRequest()` method tells the CORBA server which method will be called. If you see `_create` as the first argument, change it to `create`:

```
function  THelloHomeStub._create : HelloHome_helloworld_i.Hello;
var
  _Output: CORBA.OutputStream;
  _Input : CORBA.InputStream;
begin
  inherited _CreateRequest('create', True, _Output);
  inherited _Invoke(_Output, _Input);
  Result := HelloHome_helloworld_c.THelloHelper.Read(_Input);
end;
```

Both of these errors were fixed in the Delphi 6 update 1, so you should upgrade to that to get around these errors.

Running the Application

To run the application, follow these steps:

1. Start the OSAgent.
2. Make sure that Borland AppServer has started.
3. Start the `HelloClient`. Click on the button, and you should see the label text change to `"Hello, world"`.

More complex EJBs can be developed by following a similar development process. In Java, you just add more method interfaces to increase the capabilities of the EJB. The client-side process will essentially remain the same as this example. Delphi CORBA clients will be built by gathering the SIDL produced IDL files and processing them through the Delphi CORBA IDE wizard.

CORBA and Web Services

It is fairly straightforward to extend a CORBA application through the Web Services architecture. The SOAP specification doesn't allow object references to be passed between applications, so a little work needs to be done on the middle-tier level to isolate SOAP clients from the details of CORBA applications.

This next example will publish the EJB that was created in the last section so that it can be used by SOAP clients. Figure 19.5 shows the architecture of the application.

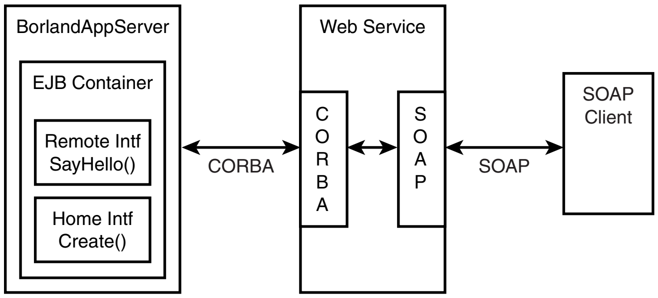


FIGURE 19.5
The CORBA/Web Services example architecture.

The EJB is deployed under the Borland AppServer. It has a published CORBA interface that any CORBA client can call. The Web Service application is both a SOAP server and a CORBA client. The SOAP server portion of the application will wrap the calls the SOAP client makes around the CORBA client interface.

By using this technique, you can harness the power of EJBs and publish the results as SOAP interfaces to clients that only have that capability. This means that a SOAP enabled application can access applications that manipulate EJBs. This brings an enormous capability to the client desktop and gets rid of CORBA ORB deployment issues on the client side as well.

In this example, the EJB will remain exactly as it was in the last section. No modifications to its interface or functionally have to be made. That portion of the application is complete.

Creating the Web Service

To create the Web Service, you need the IDL files from the last section. IDL2Pas can be run in a command window to generate the client-side files. Create a directory for the project and copy the `SIDL.idl` and `HelloHome.idl` files into the new directory. Then open a command window and type the following:

```
IDL2Pas HelloHome.idl
```

The IDL2Pas compiler will generate the files for the application. We only need the `HelloHome_I.pas` and `HelloHome_C.pas` files for the CORBA client side.

In order to build this project, you need to install the Invokamatic Wizard, which lets you create a SOAP application in a couple of minutes. When you register Delphi 6, you get access to the Delphi 6 Registered Users' Web site, which contains the Delphi eXtreme Toys downloads. This contains additional tools and freebies for Delphi 6, including the Invokamatic Wizard. Download this wizard from the Borland Web site and install it into the IDE.

Now, to create the Web Service application, follow these steps:

1. Close down all projects in Delphi and select File, New, Other, Web Services. Then choose the Soap Server Application icon.
2. Choose the Web App Debugger Executable for the target application Web server and give it a name such as `coHelloWorld` for the `coClass` name. Make sure that you use a unique name every time you repeat this exercise, or clean the registry by unregistering your application when you are done with it.
3. The wizard will generate a bare bones application. Save it to the project directory that contains the IDL files. Name the module file `ServerMod.pas`, the main form file `ServerMain.pas`, and the application `Server.dpr`.
4. Now select the Files, New, Other, Web Services and choose the Invokamatic Wizard.
5. The dialog box prompts you for a name to use. Name it HelloWorldSoap. This automatically names the interface and files. In the Invokable Class drop-down box, choose TInvokable Class. When you select OK, it creates two new units in your project. One is an interface unit, and the other is an implementation unit.
6. Select the interface unit and add the following method to the `IHelloWorldSoapIntf` interface:

   ```
   function sayHello: string; stdcall;
   ```

   The `stdcall` call tag is required so that the correct calling convention is established.
7. Now copy this method declaration to the implementation unit `THelloWorldSoapIntf` class and make it a public method.
8. Put the cursor anywhere on the method line and press Shift+Control+C to activate the class completion. Just for test purposes, we'll have this method return a hard-wired string to test the client. So type

   ```
   result := 'Hello, world';
   ```
9. Save the program, compile it, and run it. Running it registers the method interface. Make sure you start the Web App Debugger from the Delphi Tools menu. You can start the server and check the interfaces it knows about by clicking on the URL in the Web App Debugger UI.

Creating the SOAP Client Application

To create a SOAP client application,

1. Close down the project and choose File, New, Application.
2. Add one label and one edit control to the main form. Also add the interface files `SoapHTTPClient` and `HelloWorldSoapIntf` to the `uses` clause.
3. Declare a variable called `mySoap` to be of type `IHelloWorldSoap`.
4. On the button `OnClick` method, make the code resemble Listing 19.15.
5. Save the program and compile the files.
6. Run the application and click on the Say Hello button. After a small delay while the server application is loaded, you should see `"Hello, world"` in the label caption.

This is a preliminary test to show that the SOAP client and server work together. Now we can add the CORBA client to the server project to complete the full application.

LISTING 19.15 SOAP Client Main Form Class

```
unit ClientMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, SoapHTTPClient, HelloWorldSoapIntf;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
    mySoap : IHelloWorldSoap;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
```

LISTING 19.15 Continued

```
procedure TForm1.Button1Click(Sender: TObject);
var x : THTTPRio;
begin
   x := THTTPRio.Create(nil);
   x.URL := 'http://localhost:1024/Server.exe/SOAP/';
   mySoap := x as IHelloWorldSoap;
   Label1.Caption := mySoap.sayHello;
end;

end.
```

Adding the CORBA Client Code to the Web Service

To add the CORBA client files to the Web Server project,

1. Copy the `*_i.pas` and `*_c.pas` files from the EJB client application developed in the last section. The interface file is shown in Listing 19.16.

LISTING 19.16 SOAP Interface File

```
{ Invokable interface declaration unit for IHelloWorldSoap }

unit HelloWorldSoapIntf;

interface

uses
  Types, XSBuiltIns;

type
  IHelloWorldSoap = interface(IInvokable)
    ['{CA738F7B-B111-4F12-BEBD-C2ADDD80C3E2}']
    // Declare your invokable logic here using standard Object Pascal code
    // Remember to include a calling convention! (usually stdcall)
    // For example:
    // function Add(const First, Second: double): double; stdcall;
    // function Subtract(const First, Second: double): double; stdcall;
    // function Multiply(const First, Second: double): double; stdcall;
    // function Divide(const First, Second: double): double; stdcall;
    function sayHello : String; stdcall;
  end;

implementation
```

LISTING 19.16 Continued

```
uses
  InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(IHelloWorldSoap), '', '');

end.
```

2. Add two public variables to the `Form1` class as shown in Listing 19.17. The variables will be public so that they are exposed to other units in the application.
3. Add an `OnCreate()` method to the main form and make it look like the one in Listing 19.17.
4. Finally, change the `HelloWorldSoapImpl.pas` file `sayHello()` method to look like Listing 19.18.
5. Now save the project and compile the server.
6. Make sure OSAgent, and Borland AppServer are running, then run the client. When you click on the Say Hello button you should see “Hello, world”.

This was not a difficult example. However, you can now see the process of exposing EJBs to SOAP clients. This opens up new possibilities for bringing J2EE applications to the desktop and other Delphi application types.

LISTING 19.17 SOAP Server Main Form Class

```
unit ServerMain;

interface

uses
  SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Corba,
  HelloHome_helloworld_i, HelloHome_helloworld_c;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    myHome :  HelloHome;
    myRemote : Hello;
  end;
```

Listing 19.17 Continued

```
var
  Form1: TForm1;

implementation

uses ComApp;

{$R *.DFM}

const
  CLASS_ComWebApp: TGUID = '{63859D3A-005F-43BB-8E64-85A466D9C364}';

procedure TForm1.FormCreate(Sender: TObject);
begin
  CorbaInitialize;
  myHome := THelloHomeHelper.bind;
  myRemote := myHome._create;
end;

initialization
  TWebAppAutoObjectFactory.Create(Class_ComWebApp,
    'coHelloWorld', 'coHelloWorld Object');

end.
```

Listing 19.18 SOAP Server Implementation Class

```
{ Invokable implementation declaration unit for THelloWorldSoap,
  which implements IHelloWorldSoap }

unit HelloWorldSoapImpl;

interface

uses
  HelloWorldSoapIntf, InvokeRegistry, ServerMain;

type
  THelloWorldSoap = class(TInvokableClass, IHelloWorldSoap)
    // Make sure you have your invokable logic implemented in IHelloWorldSoap
    // first, save the file, then use CodeInsight(tm) to fill in this
    // implementation section by pressing Ctrl+Space, marking all the interface
    // declarations for IHelloWorldSoap, and pressing Enter.
```

LISTING 19.18 Continued

```
    // Once the declarations are inserted here, use ClassCompletion(tm)
    // to write the implementation stubs by pressing Ctrl+Shift+C
    function sayHello : String; stdcall;
  end;

implementation

{ THelloWorldSoap }

function THelloWorldSoap.sayHello: String;
begin
//  result := 'Hello, world';  //test for soap client
  result := ServerMain.Form1.myRemote.sayHello;
end;

initialization
  InvRegistry.RegisterInvokableClass(THelloWorldSoap);

end.
```

Summary

This chapter provides an introduction to developing CORBA applications in Delphi. We started with the basics of the CORBA architecture and developed a fairly simple Bank application. From there, we looked at more complex data structures.

Then we got into an area that is generating a lot of interest from enterprise developers. We learned how to develop an EJB in JBuilder 5, deploy it to the Borland AppServer, and connect a Delphi CORBA client to the EJB.

From there, we extended the EJB through a combination of CORBA and Web Services using the SOAP protocol. Using this approach, you can now take any EJB, connect it to a Web Service, and expose it to any client that can use the SOAP protocol. The client doesn't have to be aware that CORBA is deployed on the back end. This opens up new possibilities for extending corporate legacy applications.