CORBA Development with Delphi

CHAPTER

27

IN THIS CHAPTER

The acronym CORBA stands for *Common Object Request Broker Architecture*. CORBA is a specification, developed by the Object Management Group (OMG), that defines a standards-based architecture for building language- and platform-neutral object implementations. The OMG is an independent consortium of companies and industry experts who adhere strictly to the goal of developing standards for open, platform-neutral, distributed object architectures. Unlike some competing standards (such as Microsoft's COM/DCOM), the OMG does not offer any implementations of the standards it defines.

Object Request Brokers

The workhorse of the CORBA architecture is the Object Request Broker (ORB). The ORB provides the implementation of the CORBA specification and is the glue (or *middleware*) that holds the entire solution together. If you're familiar with Microsoft's COM/DCOM technology, you'll notice that the ORB provides runtime, security, and transport layers similar to that of the COM/DCOM library. All communication between client and server passes through the ORB so that method calls and parameters can be resolved into the address space of the caller or callee (marshaling). The ORB also provides many helper routines that can be called directly from a client or server, similar to the functionality that `oleaut32.dll` provides for COM/DCOM. As previously mentioned, the CORBA specification provides no default implementation of an ORB library. Because building an ORB is no trivial task, CORBA developers are dependent on third parties to supply CORBA-compliant ORB implementations. The good news is that ORB implementations are currently available from many vendors and for all the major platforms (such as Windows and UNIX) as well as some more obscure operating systems. Currently, the two most widely recognized CORBA implementations are the Inprise VisiBroker ORB and the IONA Orbix ORB.

Interfaces

A single CORBA solution can be comprised of various objects, developed in a heterogeneous mix of development languages and executing on a variety of different platforms. Therefore, there needs to be some standard way for the objects to represent themselves to other objects, clients, and the ORB. This representation is accomplished using an interface. An *interface* defines a list of available methods and their parameters but does not serve to implement any functionality of these routines. When a CORBA object implements an interface, it's guaranteeing that it implements all the methods defined by the interface. At its lowest level, an interface is simply a function table or list of entry points into specific methods. Because this construct can be represented on any hardware platform and by any serious development tool, interfaces become the common tongue of the CORBA world. Because the syntax of development languages can differ widely, the OMG has defined the Interface Definition Language (IDL),

which is used for defining CORBA interfaces. IDL is the standard language for defining CORBA interfaces, and many development tools are able to translate IDL into their native syntax in order to allow developers to easily construct CORBA-compliant interfaces. With Delphi, we won't need to manually write IDL; instead, the type library editor will allow us to visually define our interfaces and optionally export the corresponding IDL code.

Stubs and Skeletons

The CORBA mechanism works using proxies. The use of proxies is currently the leading design pattern for solving the complex problems associated with passing data between distributed objects. A proxy sits on both the client and server side and makes it appear to the client or server that it is communicating with a local process. The ORB then handles all the messy details that need to occur between the proxies (for example, marshaling, network communication, and so on). This architecture, as shown in Figure 27.1, affords the developers of a CORBA client or server some protection from low-level transport details and allows them to focus on correctly implementing their specific business solution. In CORBA terms, the proxy that represents the server that a client communicates with is called a *stub*, and a proxy that represents a client on the server side is called a *skeleton*. When you're creating a CORBA server object using the Delphi wizard, a unit containing interface definitions for the stub and skeleton will be automatically generated.

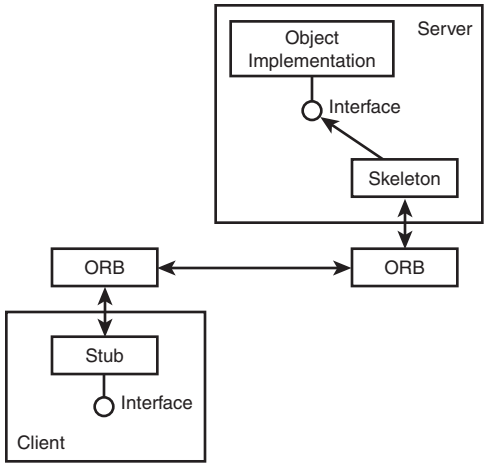

FIGURE 27.1
A simplified schematic of the CORBA architecture.

The VisiBroker ORB

As mentioned previously, CORBA is a standard that requires some third party to actually implement the ORB services. The CORBA support offered in Delphi 4 and 5 uses the VisiBroker ORB from Inprise to implement the CORBA specification. The VisiBroker product provides full support for the CORBA specification as well as many VisiBroker extensions such as naming and event services. Because full coverage of the VisiBroker product is outside the scope of this chapter, we'll focus on the parts of VisiBroker that are most pertinent to Delphi's CORBA implementation. More information on VisiBroker, including product documentation, can be found at `www.borland.com/visibroker`.

VisiBroker Runtime Support Services

Included with the VisiBroker ORB libraries are various runtime support services that function to hold the whole CORBA/VisiBroker distributed architecture together. We'll discuss each of these.

Smart Agent (osagent)

The VisiBroker Smart Agent provides *object location* services to CORBA applications. Use of the Smart Agent provides the CORBA environment with location transparency. That is, clients are not concerned with the location of the servers themselves; clients simply need to be able to locate the Smart Agent and it will handle the details of finding an appropriate server. A Smart Agent must be running somewhere on your local network. Multiple Smart Agents in a single network can be configured to listen on different ports, in effect providing multiple *ORB domains*. This might be useful for providing a production ORB environment and a development ORB environment. Smart Agents can also be configured to communicate with Smart Agents residing on different local networks, thus extending the range of your CORBA infrastructure.

Object Activation Daemon

The VisiBroker Object Activation Daemon (OAD) provides services for dynamically launching servers when their services are needed. The Smart Agent can only bind clients to implementations of objects that are already running. However, if a CORBA object implementation is registered with the Object Activation Daemon, the Smart Agent and the OAD can cooperate and launch the server process if there isn't one available.

The Interface Repository

The Interface Repository (IREP) is an online database of object type information. This repository is necessary for clients who wish to dynamically bind (late-bind) to CORBA interfaces. The ORB can use the type information in the interface repository for correctly marshaling late-bound method invocations. In order for dynamic binding to be used, the Interface Repository

must be running somewhere on the network that's accessible to clients, and the interface to be used must be registered with the repository.

VisiBroker Administration Tools

In order to configure and administer the aforementioned runtime support tools, the Delphi VisiBroker package ships with an assortment of GUI and command-line administration utilities. We list them in Table 27.1 for completeness but defer the details on their usage until they're needed later in this chapter.

TABLE 27.1 VisiBroker Administration Tools

Tool	*Purpose*
`osagent`	Used for administering the Smart Agent
`osfind`	Enumerates object implementations available on the network
`oad`	Used for administering the OAD
`oadutil`	Used for registering, unregistering, and listing interfaces with the OAD
`irep`	Used to administer the Interface Repository
`idl2ir`	Utility for registering IDL with the Interface Repository
`vregedit`	Allows for easy Registry (Windows) changes to Smart Agent defaults
`vbver`	Reports version numbers of the VisiBroker services

Delphi CORBA Support

The CORBA support introduced in Delphi (starting with version 4) has been often criticized. Although there are limitations, many of the rumors are exaggerated or simply wrong. To begin with, the support in Delphi is indeed a "true" CORBA implementation. The VisiBroker ORB for C++ (`orb_br.dll`) is used underneath and is wrapped by a dynamic link library (`orbpas50.dll`) in order to allow Pascal and Delphi interface definitions and data types to work with the VisiBroker ORB.

One area that usually causes CORBA purists to cringe is when they look at Delphi-generated stub and skeleton code and see references to GUIDs and `IUnknown` and `IDispatch` interfaces. These constructs reek of COM/DCOM, and most CORBA supporters wish to have them far from their beloved CORBA implementations. Many myths have been circulated surrounding the existence of these COM beasts, including that CORBA calls go through COM, or that parameters are marshaled twice (once through COM and once through CORBA). Before running amok with all kinds of crazy assumptions, let's examine why these COM definitions exist in a Delphi-generated CORBA server:

- To begin with, when interfaces were added to Delphi, it was done with COM in mind. All Delphi interfaces "inherit" from the base COM interface (`IUnknown`). This means that when you define an interface in Delphi that's to be used with CORBA, the three additional methods of `IUnknown` (`QueryInterface`, `AddRef`, and `Release`) must be implemented. This is true even for a CORBA interface; the base implementation of the `TCorbaImplementation` class implements these methods for the Delphi developer.
- Second, when creating a CORBA object using the Delphi wizard, you'll notice that a COM "dual" interface is created by default. By examining the generated stub and skeleton unit, you see that the CORBA interface inherits from `IDispatch` and defines a dispinterface. Although this is unnecessary for CORBA (and you can alter the definition to inherit from `IUnknown`), the object implementation must define the additional methods of `IDispatch` in order for these objects to compile properly. The class declarations of `TCorbaDispatchStub` and `TCorbaImplementation` implement the four additional methods of `IDispatch`. Careful inspection of this code will show that the implementations do not really do anything; they are present so that the type library editor can be used with CORBA objects.
- Finally, the interfaces that are generated by the wizard contain GUIDs (or IIDs). These are unique identifiers that COM uses to identify interfaces. Although CORBA does not use GUIDs itself in order to identify objects or interfaces, some internal VCL routines use these GUIDs in order to uniquely identify the CORBA interfaces. For this reason, GUIDs should not be removed from the interfaces generated by the CORBA Object Wizard.

As you can see from this discussion, the COM entities that are generated by the Delphi CORBA wizard may be less of a cost than some developers think. One beneficial side effect of this—a feature that's unique to Delphi—is that it becomes very easy to build classes that can be exposed through both COM/DCOM and CORBA at the same time!

At the time of writing, the most glaring weakness of Delphi's CORBA implementation is the lack of a utility for converting IDL to Pascal (`Idl2Pas`), a tool that's currently available from Inprise for both Java and C++. It's a common misconception that Delphi does not have the ability to early-bind to a CORBA server written in a different language. A more correct statement would be that a Delphi developer cannot *easily* early-bind to a CORBA server written in another language. Delphi clients can perform static (early) binding or dynamic (late) binding to CORBA servers written in Delphi or any other language. However, the inability of Delphi to import an IDL file and generate Pascal code that the compiler can understand makes it much more difficult to early-bind to CORBA servers that are written in other languages. Due to this, a developer must manually code CORBA stub classes when desiring to early-bind a Delphi client to a CORBA object implemented in C++ or Java. Inprise is currently working on an

Idl2Pas converter that will simplify Delphi/CORBA development and should soon be available as an add-on to Delphi 5. Later in this chapter we will provide an introductory look at this new technology.

CORBA Support Classes

The Delphi CORBA framework uses a mixture of interface and implementation inheritance in order to enable developers to create CORBA clients and servers. CORBA work is accomplished primarily by implementing interfaces for objects, stubs, and skeletons. Because interfaces do not support the concept of inheriting implementation code, this task could become quite tedious because all interfaces would need to reimplement common calls to the CORBA ORB. To address this, Delphi provides a group of VCL base classes that implement the methods of the primary CORBA interfaces (for example, ICorbaObject, ISkeletonObject, and IStubObject). The primary base classes are shown in Figure 27.2 and are described in the following list.

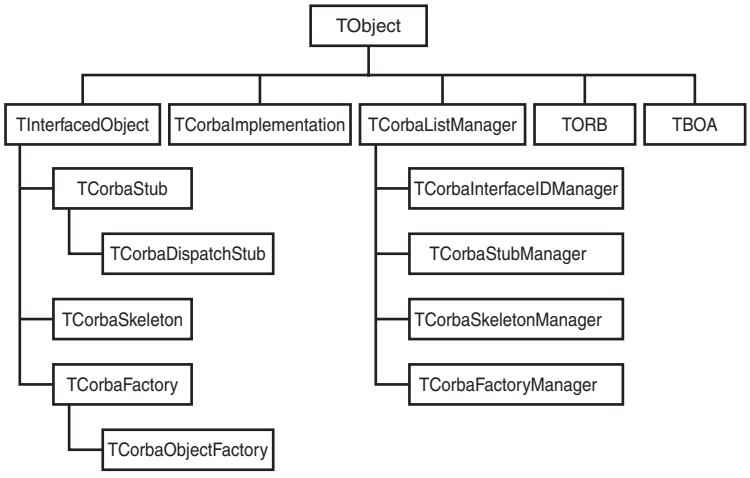

FIGURE 27.2
The VCL's CORBA support hierarchy.

- TCorbaImplementation. This class supports IUnknown (interfaces) and provides interface-querying and reference-counting capabilities. The methods of IDispatch are also stubbed out on this class so that dual interfaces added from the type library editor are supported. Delphi CORBA objects will descend from this class.
- TCorbaStub. This class implements the ICorbaObject and IStubObject interfaces. TCorbaStub is the base class for all stubs generated by the Delphi Type Library Editor.

A stub is used to marshal interface calls for a CORBA client. Developers wishing (or having) to provide their own marshaling will create `TCorbaStub` descendants.

- `TCorbaDispatchStub`. This class inherits from `TCorbaStub` and implements (stubs out) the methods of the COM interface `IDispatch`. This is so those interfaces that are created with the Delphi Type Library Editor that inherit from `IDispatch` can be used with CORBA.
- `TCorbaSkeleton`. This class implements the `ISkeletonObject` interface and is responsible for communicating with the ORB and passing calls on the server object. Unlike the stub, the skeleton class does not actually implement the interface of the server. Instead, the skeleton holds a reference to the server and invokes methods on this reference.
- `TCorbaFactory` *and* `TCorbaObjectFactory`. `TCorbaFactory` is the base class for objects that can create CORBA object instances. `TCorbaObjectFactory` can instantiate any descendants of `TCorbaImplementation`.
- `TCorbaListManager` *(and subclasses)*. The Delphi CORBA framework must keep track of various entities at runtime, such as skeletons, stubs, factories, and interface IDs. `TCorbaListManager` is a base class that provides support for thread synchronization. This allows the VCL to provide internal housekeeping in a thread-safe manner. Typically, a developer will not need to do much with these list manager classes except for occasionally registering a custom stub object.
- `TBOA`. This is the Delphi class that represents the Basic Object Adapter (BOA), a CORBA mechanism for communication between the ORB and the skeleton. The `TBOA` class is a "singleton" object and never needs to be instantiated directly.
- `TORB`. The `TORB` class is how the Delphi VCL communicates with the VisiBroker ORB. Like the `TBOA` class, the TORB class is a "singleton" and should never be instantiated directly. The implementations of many of `TORB`'s methods call functions in `orbpas50.dll`, which in turn calls routines in the VisiBroker C++ ORB (`orb_br.dll`).

CORBA Object Wizard

The classes just listed are relatively straightforward and represent just about all the VCL CORBA classes that a Delphi developer should have to deal with. However, you may be happy to know that there's a Delphi wizard that helps you properly implement your CORBA objects. Use the File, New menu to invoke Delphi's Object Repository, as shown in Figure 27.3, and select the Multitier tab.

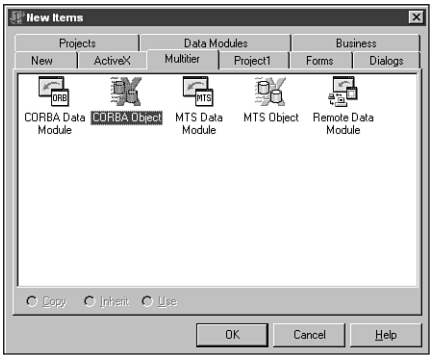

FIGURE 27.3
The Delphi Object Repository/CORBA Wizard.

Now click CORBA Object and you'll see the CORBA Object Wizard pictured in Figure 27.4.

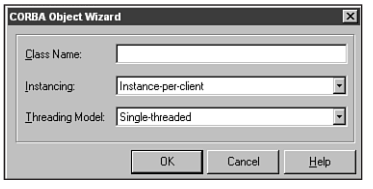

FIGURE 27.4
The CORBA Object Wizard.

Fill in the class name with the desired name for your CORBA object and interface. Note that you should probably not use the standard Delphi convention of starting your class name with a *T*, because this will be automatically added for you. For example, if you enter *MyObject*, a Delphi class named `TMyObject` will be generated that implements the interface `IMyObject`.

The Instancing option determines how object instances are handed out to clients. One of the following two options may be chosen:

- *Shared Instance*. This model is normally used for CORBA development. Each client uses a single shared instance of the object implementation. CORBA servers that use this model should be built as "stateless" servers. Because many clients may be sharing a single instance, any particular client is not guaranteed to find the server in the exact same state that it was in after the last call.
- *Instance-per-client*. The instance-per-client model constructs a unique instance of an object for each client that requests an object's service. This model allows for the construction of "state" objects that maintain a consistent state across client calls. However,

this model can be more resource-intensive because it requires servers to track the state of connected clients so that objects can be freed when clients are finished with them.

The Threading Model option specifies how your CORBA objects will be called. Two options are available here:

- *Single-threaded*. Each object instance will be called from a single thread; therefore, the object itself does not need to be made thread-safe. Note that the CORBA server application may contain multiple objects or instances; therefore, global or shared data must still be made thread-safe.
- *Multithreaded*. Although each client connection will make calls on a dedicated client thread, objects can receive concurrent calls from multiple threads. In this scenario, global as well as object data must be made thread-safe. The most difficult scenario to implement (regarding threading concerns) is you're when using a shared object instance with the multithreaded model. The simplest would be the single-threaded, instance-per-client model.

Keep in mind that simply selecting a threading option does not serve to implement your servers or objects in a thread-safe manner. These options are purely for specifying the threading model your object supports. It remains your responsibility to implement your CORBA servers in a thread-safe manner, based on the threading model desired.

After you've successfully completed the CORBA wizard, two Pascal code units will be generated. A stub/skeleton unit will be generated that follows the naming pattern *YourProject*_TLB.pas. This file will contain the definition of the main interface of your object, a stub and skeleton class, a CORBA class factory class, and code to register the stub, skeleton, and interface with the appropriate Delphi mechanisms. Listing 27.1 shows the code generated for a class named "MyFirstCORBAServer."

LISTING 27.1 A Delphi-Generated Stub and Skeleton Unit

```
unit FirstCorbaServer_TLB;

// ************************************************************************ //
// WARNING
// -------
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or the
// 'Refresh' command of the Type Library Editor activated while editing the
// Type Library, the contents of this file will be regenerated and all
// manual modifications will be lost.
// ************************************************************************ //
```

```
// PASTLWTR : $Revision:   1.88  $
// File generated on 11/02/1999 4:01:10 PM from Type Library described below.

// ************************************************************************ //
// Type Lib: C:\ICON99\FirstCORBAServer\FirstCorbaServer.tlb (1)
// IID\LCID: {CE8DB340-913A-11D3-9706-0000861F6726}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINDOWS\SYSTEM\STDOLE2.TLB)
//   (2) v4.0 StdVCL, (C:\WINDOWS\SYSTEM\STDVCL40.DLL)
// ************************************************************************ //
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked pointers.
interface

uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL,
   SysUtils, CORBAObj, OrbPas, CorbaStd;

// *********************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// *********************************************************************//
const
  // TypeLibrary Major and minor versions
  FirstCorbaServerMajorVersion = 1;
  FirstCorbaServerMinorVersion = 0;

  LIBID_FirstCorbaServer: TGUID = '{CE8DB340-913A-11D3-9706-0000861F6726}';

  IID_IMyFirstCorbaServer: TGUID = '{CE8DB341-913A-11D3-9706-0000861F6726}';
  CLASS_MyFirstCorbaServer: TGUID = '{CE8DB343-913A-11D3-9706-0000861F6726}';
type

// *********************************************************************//
// Forward declaration of types defined in TypeLibrary
// *********************************************************************//
  IMyFirstCorbaServer = interface;
  IMyFirstCorbaServerDisp = dispinterface;

// *********************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// *********************************************************************//
```

continues

LISTING 27.1 Continued

```
  MyFirstCorbaServer = IMyFirstCorbaServer;


// ************************************************************************//
// Interface: IMyFirstCorbaServer
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {CE8DB341-913A-11D3-9706-0000861F6726}
// ************************************************************************//
  IMyFirstCorbaServer = interface(IDispatch)
    ['{CE8DB341-913A-11D3-9706-0000861F6726}']
    procedure SayHelloWorld; safecall;
  end;

// ************************************************************************//
// DispIntf:  IMyFirstCorbaServerDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {CE8DB341-913A-11D3-9706-0000861F6726}
// ************************************************************************//
  IMyFirstCorbaServerDisp = dispinterface
    ['{CE8DB341-913A-11D3-9706-0000861F6726}']
    procedure SayHelloWorld; dispid 1;
  end;

  TMyFirstCorbaServerStub = class(TCorbaDispatchStub, IMyFirstCorbaServer)
  public
    procedure SayHelloWorld; safecall;
  end;

  TMyFirstCorbaServerSkeleton = class(TCorbaSkeleton)
  private
    FIntf: IMyFirstCorbaServer;
  public
    constructor Create(const InstanceName: string; const Impl: IUnknown);
      override;
    procedure GetImplementation(out Impl: IUnknown); override; stdcall;
  published
    procedure SayHelloWorld(const InBuf: IMarshalInBuffer; Cookie: Pointer);
  end;

// ************************************************************************//
// The Class CoMyFirstCorbaServer provides a Create and CreateRemote method to
// create instances of the default interface IMyFirstCorbaServer exposed by
// the CoClass MyFirstCorbaServer. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this typelibrary.
```

```
// ********************************************************************//
  CoMyFirstCorbaServer = class
    class function Create: IMyFirstCorbaServer;
    class function CreateRemote(const MachineName: string):
    ➥IMyFirstCorbaServer;
  end;

  TMyFirstCorbaServerCorbaFactory = class
    class function CreateInstance(const InstanceName: string):
      IMyFirstCorbaServer;
  end;

implementation

uses ComObj;

{ TMyFirstCorbaServerStub }

procedure TMyFirstCorbaServerStub.SayHelloWorld;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('SayHelloWorld', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
end;

{ TMyFirstCorbaServerSkeleton }

constructor TMyFirstCorbaServerSkeleton.Create(const InstanceName: string;
  const Impl: IUnknown);
begin
  inherited;
  inherited InitSkeleton('MyFirstCorbaServer', InstanceName,
    'IDL:FirstCorbaServer/IMyFirstCorbaServer:1.0', tmMultiThreaded, True);
  FIntf := Impl as IMyFirstCorbaServer;
end;

procedure TMyFirstCorbaServerSkeleton.GetImplementation(out Impl: IUnknown);
begin
  Impl := FIntf;
end;

procedure TMyFirstCorbaServerSkeleton.SayHelloWorld(
  const InBuf: IMarshalInBuffer; Cookie: Pointer);
var
```

continues

LISTING 27.1 Continued

```
  OutBuf: IMarshalOutBuffer;
begin
  FIntf.SayHelloWorld;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

class function CoMyFirstCorbaServer.Create: IMyFirstCorbaServer;
begin
  Result := CreateComObject(CLASS_MyFirstCorbaServer) as IMyFirstCorbaServer;
end;

class function CoMyFirstCorbaServer.CreateRemote(const MachineName: string):
  IMyFirstCorbaServer;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_MyFirstCorbaServer) as
    IMyFirstCorbaServer;
end;

class function TMyFirstCorbaServerCorbaFactory.CreateInstance(
  const InstanceName: string): IMyFirstCorbaServer;
begin
  Result := CorbaFactoryCreateStub(
    'IDL:FirstCorbaServer/MyFirstCorbaServerFactory:1.0', 'MyFirstCorbaServer',
    InstanceName, '', IMyFirstCorbaServer) as IMyFirstCorbaServer;
end;

initialization
  CorbaStubManager.RegisterStub(IMyFirstCorbaServer, TMyFirstCorbaServerStub);
  CorbaInterfaceIDManager.RegisterInterface(IMyFirstCorbaServer,
    'IDL:FirstCorbaServer/IMyFirstCorbaServer:1.0');
  CorbaSkeletonManager.RegisterSkeleton(IMyFirstCorbaServer,
    TMyFirstCorbaServerSkeleton);

end.
```

Upon examination of this stub and skeleton unit, one interesting point to note is that the skeleton class does not actually implement the `IMyFirstCorbaServer` interface. The skeleton will have the same methods as the supported interface, but you'll notice that the parameters are different. The methods of the skeleton will receive raw, marshaled information and must then unmarshal the parameters and pass them to the appropriate interface. For this reason, the skeleton does not implement the interface directly. Instead, the skeleton will hold an internal reference to the supported interface and delegate its calls to this internal reference.

The second unit generated will contain the framework for implementing your object. A Delphi class that descends from `TCorbaImplementation` and implements your main interface will be generated. This unit will also create an instance of the factory that's responsible for creating the CORBA object. A typical CORBA object implementation unit would look like the code shown in Listing 27.2.

LISTING 27.2 A Delphi-Generated CORBA Object Implementation

```
unit uMyFirstCorbaServer;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, ComObj, StdVcl,
  CorbaObj, FirstCorbaServer_TLB;

type

  TMyFirstCorbaServer = class(TCorbaImplementation, IMyFirstCorbaServer)
  private
    { Private declarations }
  public
    { Public declarations }
  protected
    procedure SayHelloWorld; safecall;
  end;

implementation

uses CorbInit;

procedure TMyFirstCorbaServer.SayHelloWorld;
begin
  //Implement method here.
end;

initialization
  TCorbaObjectFactory.Create('MyFirstCorbaServerFactory', 'MyFirstCorbaServer',
    'IDL:FirstCorbaServer/MyFirstCorbaServerFactory:1.0', IMyFirstCorbaServer,
    TMyFirstCorbaServer, iMultiInstance, tmSingleThread);
end.
```

This unit will eventually contain the code that implements all the methods of the `IMyFirstCORBAServer` interface as well as any internal functionality of the `TMyFirstCORBAServer` class. By using classical implementation inheritance descending from `TCorbaImplementation`, the implementation will automatically be able to become a CORBA object. By supporting the `IMyFirstCorbaServer` interface, the object guarantees it will satisfy the contract of this interface. In lieu of manually declaring the object's interface and implementation, we'll now turn to the Delphi Visual Type Library Editor.

The Delphi Type Library Editor

To fully implement this custom CORBA object, code must be added to both the stub and skeleton unit and the object implementation unit listed previously. Although at first this may appear to be a somewhat daunting task, the Delphi Type Library Editor is available to help you with the process. Proceed to the Delphi main menu and select View, Type Library. You'll see the window shown in Figure 27.5, which visually represents the interfaces and other entities defined in the stub and skeleton unit.

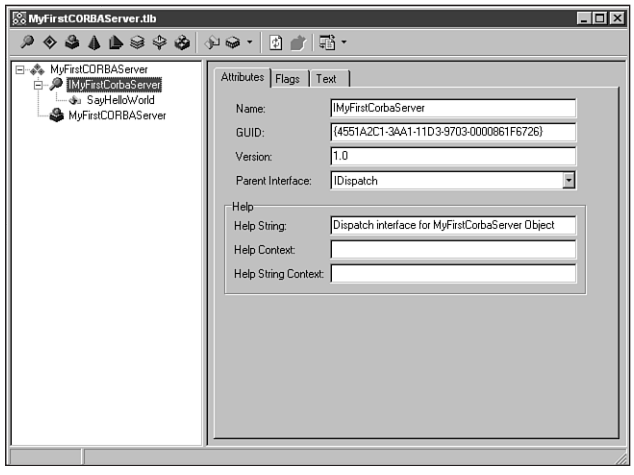

FIGURE 27.5
The Delphi Visual Type Library Editor.

At this point, you can select the `IMyFirstCorbaServer` interface in the editor and click the speedbutton to add a new method. Once the method has been added, you can use the editor's visual interface to define parameters, return types, and so on. Note that not all the data types shown as possible parameter types in the Type Library Editor are valid for CORBA objects. Because the Type Library Editor currently is a dual-purpose tool for both COM and CORBA, many of the data types are valid for COM/Automation objects only. Consult the Delphi help

files for exhaustive lists of valid CORBA (IDL) data types. Once you've used the Type Library Editor to add the methods of your interface, clicking the Refresh speedbutton will regenerate the code in your project. The stub and skeleton unit will be refreshed, and empty implementation methods will be added to your implementation unit. All that's left for you to do is to fill in the code and implement the empty methods that the Type Library Editor generates.

NOTE

Delphi 5 contains a new feature that will generate component wrappers for CoClasses contained in a type library. Unfortunately, wrappers are generated whether you are importing an existing type library or creating one of your own. These component wrappers are not appropriate for a CORBA object, so you should perform the following steps to prevent generation of this additional code. From the Delphi menu, select Project, Import Type Library. When the dialog box appears, clear the "Generate Component Wrapper" check box and close the dialog box by clicking Close in the upper right corner. Finally, click the Refresh speedbutton in the Type Library Editor. The extraneous code will be eliminated from your application.

Creating CORBA Solutions with Delphi 5

Now that we have discussed the basic CORBA framework and IDE tools in Delphi, we are going to apply our knowledge by creating a CORBA server. Then we'll finish by building a client that will use our custom CORBA server.

Building a CORBA Server

Having examined the basics of creating a CORBA server, we'll now go into detail and construct a CORBA server from start to finish. Our objective is to create a middle-tier CORBA object that can accept SQL queries from a client, query a database, and send results back to the calling client. Our implementation will use the Borland Database Engine (BDE) in order to easily retrieve data from a database server. Keep in mind that this dependency is only a consideration from the standpoint of the server object. The client application needs no knowledge (or deployment) of the BDE, and the server could easily be adapted to retrieve data using other mechanisms such as Delphi 5's new ADO datasets or even a custom `TDataset`.

Invoking the CORBA Object Wizard

Create a new Delphi application and then invoke the CORBA Object Wizard as described earlier. The name of our object will be `QueryServer`; this will produce an interface named `IQueryServer` and an implementation class with the name `TQueryServer`. Choose Instance-Per-Client for the Instancing option because our object will support data navigation (for example,

First, Next, and so on) and therefore is not a stateless object. In order to avoid the complexities of writing thread-safe code at this point, select Single-Threaded for the Threading Model option. After you click OK, the stub and skeleton unit as well as the object's implementation unit will be added to the project.

You may notice that the default Delphi application contains a form by default. A Delphi GUI application must have a form in order to remain in the main Windows message loop. Most CORBA server applications have no need for a visual form; therefore, we could solve this by entering

```
Application.ShowMainForm := False;
```

in the project file of the application. For this example, we would like to verify that the server is running, so we'll leave the form visible and provide a `TLabel` to inform us that our CORBA server is active. This form is shown in Figure 27.6.

FIGURE 27.6
Our CORBA server's main form.

Be aware that this form should be considered global data. Even though we've created the CORBA object with a single-threaded model, the CORBA server application could contain other objects that are servicing calls on other threads. Therefore, accessing this form from the code of the object would not be considered thread-safe.

Using the Type Library Editor

Now that we've generated the necessary code to implement our CORBA object, we're going to use the Type Library Editor to add support methods to our interface. We're going to add functionality to our `IQueryServer` interface to allow clients to log in to a database and send SQL statements, navigate the data, and retrieve a row at a time of the result set. This is accomplished by selecting the `IQueryServer` interface and clicking the New Method speedbutton. As each of our new methods are added, we can name them using the Name edit box on the Attributes tab. For each new method, you may also need to use the grid on the Parameters tab of the Type Library Editor in order to supply parameter types and return values. After adding several methods to provide our desired functionality, the Type Library Editor will look like Figure 27.7.

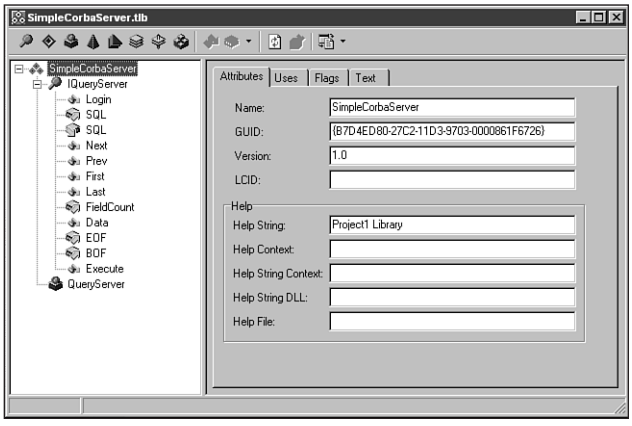

FIGURE 27.7
`IQueryServer` *methods in the Type Library Editor.*

Implementing the Methods of IQueryServer

Now that we've defined the interface of our CORBA object, what remains to be done is to implement the code to make the exposed methods work. Our implementation class will encapsulate a `TDatabase` and a `TQuery` in order to provide access to the BDE and server data. The remainder of the work is trivial—the interface methods will simply call the provided functionality of the `TDatabase` and `TQuery` VCL components.

The only method that will be a little more involved to implement is the `Data` method (function). This method will retrieve the entire row of data that's currently positioned in the query results. Because we're returning multiple values, we need some type of structure to be returned that represents these values appropriately. In IDL, this would usually involve the use of a *sequence*, which is a varying array of some data type. The Type Library Editor does not currently allow us to define an IDL sequence, so we'll make the return type of the `Data` method be an `OLEVariant`. This `OLEVariant` will actually be an array that holds the column values for the positioned row in each of its elements. We can use an `OLEVariant` for this task because IDL has a similar construct called an `Any` that can hold any valid IDL data type. The IDL that Delphi generates (shown later) will recognize an `OLEVariant` as an IDL `Any`, and the Delphi CORBA framework will allow this value to be converted to an `Any` and correctly marshaled to and from the ORB. In fact, there's a type declared in the Delphi VCL called `TAny` that maps directly to a `Variant`. All we'll need to do is create an array of `Variant` types and pass this as the return value of our `Data` function, as follows:

```
function TQueryServer.Data: OleVariant;
var
  i : integer;
```

```
begin
  //Pack and send data.
  Result := VarArrayCreate([0,FQuery.FieldCount-1],varOLEStr);
  for i := 0 to FQuery.FieldCount - 1 do
  begin
    Result[i] := FQuery.Fields[i].AsString;
  end;
end;
```

Once we implement the remainder of our methods, we'll have a stub and skeleton unit, as shown in Listing 27.3.

LISTING 27.3 The Stub and Skeleton Unit for `IQueryServer`

```
unit SimpleCorbaServer_TLB;

// ************************************************************************ //
// WARNING
// -------
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or the
// 'Refresh' command of the Type Library Editor activated while editing the
// Type Library, the contents of this file will be regenerated and all
// manual modifications will be lost.
// ************************************************************************ //

// PASTLWTR : $Revision:   1.88  $
// File generated on 11/02/1999 6:01:08 PM from Type Library described below.

// ************************************************************************ //
// Type Lib: C:\ICON99\CORBA Server\SimpleCorbaServer.tlb (1)
// IID\LCID: {B7D4ED80-27C2-11D3-9703-0000861F6726}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINDOWS\SYSTEM\STDOLE2.TLB)
//   (2) v4.0 StdVCL, (C:\WINDOWS\SYSTEM\STDVCL40.DLL)
// ************************************************************************ //
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked pointers.
interface

uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL,
   SysUtils, CORBAObj, OrbPas, CorbaStd;

// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
```

```
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// *********************************************************************//
const
  // TypeLibrary Major and minor versions
  SimpleCorbaServerMajorVersion = 1;
  SimpleCorbaServerMinorVersion = 0;

  LIBID_SimpleCorbaServer: TGUID = '{B7D4ED80-27C2-11D3-9703-0000861F6726}';

  IID_IQueryServer: TGUID = '{B7D4ED81-27C2-11D3-9703-0000861F6726}';
  CLASS_QueryServer: TGUID = '{B7D4ED83-27C2-11D3-9703-0000861F6726}';
type

// *********************************************************************//
// Forward declaration of types defined in TypeLibrary
// *********************************************************************//
  IQueryServer = interface;
  IQueryServerDisp = dispinterface;

// *********************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// *********************************************************************//
  QueryServer = IQueryServer;


// *********************************************************************//
// Interface: IQueryServer
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {B7D4ED81-27C2-11D3-9703-0000861F6726}
// *********************************************************************//
  IQueryServer = interface(IDispatch)
    ['{B7D4ED81-27C2-11D3-9703-0000861F6726}']
    function  Login(const Db: WideString; const User: WideString;
      const Password: WideString): WordBool; safecall;
    function  Get_SQL: WideString; safecall;
    procedure Set_SQL(const Value: WideString); safecall;
    procedure Next; safecall;
    procedure Prev; safecall;
    procedure First; safecall;
    procedure Last; safecall;
    function  Get_FieldCount: Integer; safecall;
    function  Data: OleVariant; safecall;
```

continues

LISTING 27.3 Continued

```
    function  Get_EOF: WordBool; safecall;
    function  Get_BOF: WordBool; safecall;
    function  Execute: WordBool; safecall;
    property SQL: WideString read Get_SQL write Set_SQL;
    property FieldCount: Integer read Get_FieldCount;
    property EOF: WordBool read Get_EOF;
    property BOF: WordBool read Get_BOF;
  end;

// ******************************************************************//
// DispIntf:  IQueryServerDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {B7D4ED81-27C2-11D3-9703-0000861F6726}
// ******************************************************************//
  IQueryServerDisp = dispinterface
    ['{B7D4ED81-27C2-11D3-9703-0000861F6726}']
    function  Login(const Db: WideString; const User: WideString;
      const Password: WideString): WordBool; dispid 1;
    property SQL: WideString dispid 2;
    procedure Next; dispid 3;
    procedure Prev; dispid 4;
    procedure First; dispid 5;
    procedure Last; dispid 6;
    property FieldCount: Integer readonly dispid 7;
    function  Data: OleVariant; dispid 8;
    property EOF: WordBool readonly dispid 9;
    property BOF: WordBool readonly dispid 11;
    function  Execute: WordBool; dispid 12;
  end;

  TQueryServerStub = class(TCorbaDispatchStub, IQueryServer)
  public
    function Login(const Db: WideString; const User: WideString;
      const Password: WideString): WordBool; safecall;
    function Get_SQL: WideString; safecall;
    procedure Set_SQL(const Value: WideString); safecall;
    procedure Next; safecall;
    procedure Prev; safecall;
    procedure First; safecall;
    procedure Last; safecall;
    function Get_FieldCount: Integer; safecall;
    function Data: OleVariant; safecall;
    function Get_EOF: WordBool; safecall;
    function Get_BOF: WordBool; safecall;
    function Execute: WordBool; safecall;
```

```
  end;

  TQueryServerSkeleton = class(TCorbaSkeleton)
  private
    FIntf: IQueryServer;
  public
    constructor Create(const InstanceName: string; const Impl: IUnknown);
      override;
    procedure GetImplementation(out Impl: IUnknown); override; stdcall;
  published
    procedure Login(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Get_SQL(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Set_SQL(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Next(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Prev(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure First(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Last(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Get_FieldCount(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Data(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Get_EOF(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Get_BOF(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Execute(const InBuf: IMarshalInBuffer; Cookie: Pointer);
  end;

// *********************************************************************//
// The Class CoQueryServer provides a Create and CreateRemote method to
// create instances of the default interface IQueryServer exposed by
// the CoClass QueryServer. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this typelibrary.
// *********************************************************************//
  CoQueryServer = class
    class function Create: IQueryServer;
    class function CreateRemote(const MachineName: string): IQueryServer;
  end;

  TQueryServerCorbaFactory = class
    class function CreateInstance(const InstanceName: string): IQueryServer;
  end;

implementation

uses ComObj;

{ TQueryServerStub }
```

continues

LISTING 27.3 Continued

```
function TQueryServerStub.Login(const Db: WideString; const User: WideString;
                                const Password: WideString): WordBool;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Login', True, OutBuf);
  OutBuf.PutWideText(PWideChar(Pointer(Db)));
  OutBuf.PutWideText(PWideChar(Pointer(User)));
  OutBuf.PutWideText(PWideChar(Pointer(Password)));
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalWordBool(InBuf);
end;

function TQueryServerStub.Get_SQL: WideString;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Get_SQL', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalWideText(InBuf);
end;

procedure TQueryServerStub.Set_SQL(const Value: WideString);
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Set_SQL', True, OutBuf);
  OutBuf.PutWideText(PWideChar(Pointer(Value)));
  FStub.Invoke(OutBuf, InBuf);
end;

procedure TQueryServerStub.Next;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Next', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
end;

procedure TQueryServerStub.Prev;
var
```

```
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Prev', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
end;

procedure TQueryServerStub.First;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('First', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
end;

procedure TQueryServerStub.Last;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Last', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
end;

function TQueryServerStub.Get_FieldCount: Integer;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Get_FieldCount', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := InBuf.GetLong;
end;

function TQueryServerStub.Data: OleVariant;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Data', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalAny(InBuf);
end;

function TQueryServerStub.Get_EOF: WordBool;
```

continues

LISTING 27.3 Continued

```
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Get_EOF', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalWordBool(InBuf);
end;

function TQueryServerStub.Get_BOF: WordBool;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Get_BOF', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalWordBool(InBuf);
end;

function TQueryServerStub.Execute: WordBool;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Execute', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalWordBool(InBuf);
end;

{ TQueryServerSkeleton }

constructor TQueryServerSkeleton.Create(const InstanceName: string;
  const Impl: IUnknown);
begin
  inherited;
  inherited InitSkeleton('QueryServer', InstanceName,
    'IDL:SimpleCorbaServer/IQueryServer:1.0', tmMultiThreaded, True);
  FIntf := Impl as IQueryServer;
end;

procedure TQueryServerSkeleton.GetImplementation(out Impl: IUnknown);
begin
  Impl := FIntf;
end;
```

```
procedure TQueryServerSkeleton.Login(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: WordBool;
  Db: WideString;
  User: WideString;
  Password: WideString;
begin
  Db := UnmarshalWideText(InBuf);
  User := UnmarshalWideText(InBuf);
  Password := UnmarshalWideText(InBuf);
  Retval := FIntf.Login(Db, User, Password);
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  MarshalWordBool(OutBuf, Retval);
end;

procedure TQueryServerSkeleton.Get_SQL(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: WideString;
begin
  Retval := FIntf.Get_SQL;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  OutBuf.PutWideText(PWideChar(Pointer(Retval)));
end;

procedure TQueryServerSkeleton.Set_SQL(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Value: WideString;
begin
  Value := UnmarshalWideText(InBuf);
  FIntf.Set_SQL(Value);
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

procedure TQueryServerSkeleton.Next(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
begin
  FIntf.Next;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
```

continues

LISTING 27.3 Continued

```
end;

procedure TQueryServerSkeleton.Prev(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
begin
  FIntf.Prev;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

procedure TQueryServerSkeleton.First(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
begin
  FIntf.First;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

procedure TQueryServerSkeleton.Last(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
begin
  FIntf.Last;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

procedure TQueryServerSkeleton.Get_FieldCount(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: Integer;
begin
  Retval := FIntf.Get_FieldCount;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  OutBuf.PutLong(Retval);
end;

procedure TQueryServerSkeleton.Data(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: OleVariant;
```

```
begin
  Retval := FIntf.Data;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  MarshalAny(OutBuf, Retval);
end;

procedure TQueryServerSkeleton.Get_EOF(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: WordBool;
begin
  Retval := FIntf.Get_EOF;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  MarshalWordBool(OutBuf, Retval);
end;

procedure TQueryServerSkeleton.Get_BOF(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: WordBool;
begin
  Retval := FIntf.Get_BOF;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  MarshalWordBool(OutBuf, Retval);
end;

procedure TQueryServerSkeleton.Execute(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: WordBool;
begin
  Retval := FIntf.Execute;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  MarshalWordBool(OutBuf, Retval);
end;

class function CoQueryServer.Create: IQueryServer;
begin
  Result := CreateComObject(CLASS_QueryServer) as IQueryServer;
end;

class function CoQueryServer.CreateRemote(const MachineName: string):
  IQueryServer;
```

continues

LISTING 27.3 Continued

```
begin
  Result := CreateRemoteComObject(MachineName, CLASS_QueryServer) as
    IQueryServer;
end;

class function TQueryServerCorbaFactory.CreateInstance(
  const InstanceName: string): IQueryServer;
begin
  Result := CorbaFactoryCreateStub(
    'IDL:SimpleCorbaServer/QueryServerFactory:1.0', 'QueryServer',
    InstanceName, '', IQueryServer) as IQueryServer;
end;

initialization
  CorbaStubManager.RegisterStub(IQueryServer, TQueryServerStub);
  CorbaInterfaceIDManager.RegisterInterface(IQueryServer,
    'IDL:SimpleCorbaServer/IQueryServer:1.0');
  CorbaSkeletonManager.RegisterSkeleton(IQueryServer, TQueryServerSkeleton);

end.
```

Notice that the Type Library Editor in conjunction with the Delphi wizards have generated all the necessary code to correctly marshal parameters. Parameters are marshaled from the stub to the ORB and are unmarshaled from the skeleton to the actual object implementation.

The only code we'll have to write is shown in Listing 27.4. You can see we only have to deal with correctly implementing our object's behavior; we don't have to get into the messy details of CORBA and parameter marshaling.

LISTING 27.4 The Implementation Unit for `TQueryServer`

```
unit uQueryServer;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, ComObj, StdVcl,
  CorbaObj, db, dbtables, orbpas, SimpleCorbaServer_TLB, frmqueryserver;

type

  TQueryServer = class(TCorbaImplementation, IQueryServer)
  private
    { Private declarations }
```

```
    FDatabase: TDatabase;
    FQuery: TQuery;
  public
    { Public declarations }
    constructor Create(Controller: IObject; AFactory: TCorbaFactory); override;
    destructor Destroy; override;
  protected
    function Data: OleVariant; safecall;
    function Get_BOF: WordBool; safecall;
    function Get_EOF: WordBool; safecall;
    function Get_FieldCount: Integer; safecall;
    function Get_SQL: WideString; safecall;
    function Login(const Db, User, Password: WideString): WordBool; safecall;
    procedure First; safecall;
    procedure Last; safecall;
    procedure Next; safecall;
    procedure Prev; safecall;
    procedure Set_SQL(const Value: WideString); safecall;
    function Execute: WordBool; safecall;
  end;

implementation

uses CorbInit;

function TQueryServer.Data: OleVariant;
var
  i : integer;
begin
  //Pack and send data.
  Result := VarArrayCreate([0,FQuery.FieldCount-1],varOLEStr);
  for i := 0 to FQuery.FieldCount - 1 do
  begin
    Result[i] := FQuery.Fields[i].AsString;
  end;
end;

function TQueryServer.Get_BOF: WordBool;
begin
  Result := FQuery.BOF;
end;

function TQueryServer.Get_EOF: WordBool;
begin
  Result := FQuery.EOF;
end;
```

continues

LISTING 27.4 Continued

```
function TQueryServer.Get_FieldCount: Integer;
begin
  Result := FQuery.FieldCount;
end;

function TQueryServer.Get_SQL: WideString;
begin
  Result := FQuery.SQL.Text;
end;

function TQueryServer.Login(const Db, User,
  Password: WideString): WordBool;
begin
  if FDatabase.Connected then FDatabase.Close;
  FDatabase.AliasName := Db;
  FDatabase.Params.Clear;
  FDatabase.Params.Add('USER NAME=' + User);
  FDatabase.Params.Add('PASSWORD=' + Password);
  FDatabase.Open;
end;

procedure TQueryServer.First;
begin
  FQuery.First;
end;

procedure TQueryServer.Last;
begin
  FQuery.Last;
end;

procedure TQueryServer.Next;
begin
  FQuery.Next;
end;

procedure TQueryServer.Prev;
begin
  FQuery.Prior;
end;

procedure TQueryServer.Set_SQL(const Value: WideString);
begin
  FQuery.SQL.Clear;
  FQuery.SQL.Add(Value);
```

```
end;

constructor TQueryServer.Create(Controller: IObject;
  AFactory: TCorbaFactory);
begin
  inherited Create(Controller,AFactory);
  FDatabase := TDatabase.Create(nil);
  FDatabase.LoginPrompt := false;
  FDatabase.DatabaseName := 'CorbaDb';
  FDatabase.HandleShared := true;
  FQuery := TQuery.Create(nil);
  FQuery.DatabaseName := 'CorbaDb';
end;

destructor TQueryServer.Destroy;
begin
  FQuery.Free;
  FDatabase.Free;
  inherited Destroy;
end;

function TQueryServer.Execute: WordBool;
begin
  FQuery.Close;
  FQuery.Open;
end;

initialization
  TCorbaObjectFactory.Create('QueryServerFactory', 'QueryServer',
    'IDL:SimpleCorbaServer/QueryServerFactory:1.0', IQueryServer,
    TQueryServer, iMultiInstance, tmSingleThread);
end.
```


One VCL detail you should note in the code in Listing 27.4 is the correct handling of the `TDatabase` object. The BDE namespace only allows for one uniquely named database within the same session. Because we could have multiple `TQueryServer` objects within this CORBA server that are sharing a single `TSession` object, we must set the `HandleShared` property of the `TDatabase` to True. If we don't do this, the next client that causes a new `TQueryServer` to be created will not be able to connect.

From the Type Library Editor, you can view the IDL that represents our interface. Click the drop-down arrow on the Export to IDL speedbutton in the Type Library Editor and select Export to CORBA IDL (note that this is similar but yet different from Microsoft IDL, or *MIDL*). You'll see the IDL code in the Delphi editor, as shown in Listing 27.5.

LISTING 27.5 The CORBA IDL for `IQueryServer`

```
module SimpleCorbaServer
{
  interface IQueryServer;


  interface IQueryServer
  {
    boolean Login(in wstring Db, in wstring User, in wstring Password);
    wstring Get_SQL();
    wstring Set_SQL(in wstring Value);
    void Next();
    void Prev();
    void First();
    void Last();
    long Get_FieldCount();
    any Data();
    boolean Get_EOF();
    boolean Get_BOF();
    boolean Execute();
  };

  interface QueryServerFactory
  {
    IQueryServer CreateInstance(in string InstanceName);
  };

};
```

Notice that the COM data types we selected in the Type Library Editor have all been properly converted to their IDL equivalents. This IDL can be imported into any other tools that support CORBA. Development tools such as CBuilder and JBuilder will generate wrapper classes so that clients written in these languages can easily use the functionality of our Delphi CORBA object.

NOTE

The IDL generated by Delphi, shown in Listing 27.5, is actually slightly incorrect. The `Set_SQL` function should not be returning a value. Although Delphi should correctly handle this, the problem stems from the fact that we added a property (`SQL`) in the Type Library Editor. Properties are recognized by COM but are not a construct normally found in CORBA. Delphi has created the read and write methods for the

property, but has not exported the write method to IDL correctly. This problem can be avoided by only declaring methods on your CORBA interfaces, or by manually editing the generated IDL to correct the declaration as follows:

```
void Set_SQL(in wstring Value);
```

Running the CORBA Server

The construction of our query server is finally complete. Now it's time to run the CORBA server application and let the VisiBroker ORB know that our object is available to clients. In order for clients to locate and connect to CORBA object implementations using the VisiBroker ORB, the VisiBroker Smart Agent must be running somewhere on your local network. The agent does not have to be running on the same computer as the client or the server. The Smart Agent can be launched from the command line (on Windows NT the Smart Agent can be run as a service) by typing

```
OSAGENT [-options]
```

from the command prompt, where valid options are as follows:

- `-p`. Sets a port for the agent to listen on
- `-v`. Prints debugging information to `osagent.log`
- `-?`. Prints usage information to `osagent.log`
- `-C`. Runs `osagent` in console mode (only on NT; default for 95/98)

If you are manually starting the Smart Agent on Windows NT, it is important to launch `osagent` using the `-c` switch. This will allow an `osagent` that has been installed as an NT service to run as a console application. An example of starting the Smart Agent on Windows NT as a console application to listen for requests on port 14005 would look as follows:

```
osagent -c  -p 14005.
```

Once the Smart Agent is running on the network, you can run the project we've just built and it will register itself with the Smart Agent and become available for client connections. Note that at this point you must actually run the server application; there is no built-in facility for launching a server (as in DCOM) unless you use the OAD.

Building an Early-Bound CORBA Client

Now that we have an available CORBA server serving objects, we can proceed to the next step and create a CORBA client with Delphi. We're going to build a simple client that uses the `IQueryServer` interface prepared earlier to read data from the server and populate a string grid with the retrieved data. It's important to realize that we're reaping the benefits of a multitier

architecture here. Our client only needs access to the VisiBroker ORB software and does not need any knowledge whatsoever of Delphi datasets or the Borland Database Engine (BDE).

A CORBA client can communicate via two ways with a CORBA object: early binding and late binding. *Early binding* means that the compiler can compile direct calls to the v-table of the stub. This not only offers performance benefits, but the compiler can provide type checking to ensure that you're passing correct parameter data types. In a late-binding scenario, all remote calls are made through the `Any` data type. Calls are slower because parameter information must be obtained from the VisiBroker Interface Repository and the incorrect parameter types are not detected until runtime. In order for Delphi to early-bind to a stub, the compiler must be supplied with some Pascal representation on the stub interface. With objects built in other languages, this becomes more difficult because Delphi 5 currently does not ship with a utility to convert IDL files into Pascal. In our case, we've built the server in Delphi and the wizards have generated a Pascal version of the stub interface. Therefore, we can early-bind to our server by simply including the stub and skeleton unit from the preceding example in the `uses` clause of our client.

Creating the CORBA Client

We'll first create a simple Delphi GUI application that will serve to view the results we obtain from the `IQueryServer` interface, as shown in Figure 27.8.

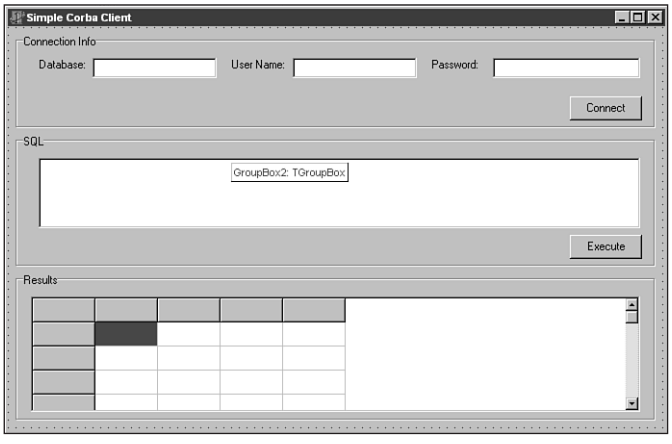

FIGURE 27.8
Our CORBA client GUI.

Having done this, we'll add the stub and skeleton unit from the server example to the `uses` clause of our form's unit (`SimpleCorbaServer_TLB.pas`).

Connecting to the CORBA Server

All that remains is to connect to our server and begin to make method calls against the remote interface. The used stub and skeleton unit defines a class factory for `IQueryServer` (named `TQueryServerCorbaFactory`). This class provides a class function (so we don't need to create an instance of `TQueryServerCorbaFactory`) named `CreateInstance` that will create the appropriate stub object and return the `IQueryServer` interface to us. We can then proceed to make early-bound calls to the remote `IQueryServer` interface. The only other nontrivial work in this client is to call the `Data` method of `IQueryServer` and unbundle the `OLEVariant` array in order to populate our string grid. This is done in the `ExecuteClick` event of our client. The complete implementation of our CORBA client is shown in Listing 27.6.

LISTING 27.6 The Implementation of `SimpleCorbaClient`

```
unit ufrmCorbaClient;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, SimpleCorbaServer_TLB, corbaObj, Grids;

type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    Label2: TLabel;
    edtDatabase: TEdit;
    Label3: TLabel;
    edtUserName: TEdit;
    Label4: TLabel;
    edtPassword: TEdit;
    Button5: TButton;
    GroupBox2: TGroupBox;
    memoSQL: TMemo;
    GroupBox3: TGroupBox;
    Button6: TButton;
    grdCorbaData: TStringGrid;
    procedure ConnectClick(Sender: TObject);
    procedure ExecuteClick(Sender: TObject);
  private
    { Private declarations }
    FQueryServer: IQueryServer;
  public
    { Public declarations }
  end;
```

continues

LISTING 27.6 Continued

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.ConnectClick(Sender: TObject);
begin
  if not(assigned(FQueryServer)) then
    FQueryServer := TQueryServerCorbaFactory.CreateInstance('SimpleServer');
  FQueryServer.Login(edtDatabase.Text,edtUserName.Text,edtPassword.Text);
end;

procedure TForm1.ExecuteClick(Sender: TObject);
var
  i,j: integer;
  CorbaData : OLEVariant;
begin
  FQueryServer.SQL := memoSQL.Text;
  FQueryServer.Execute;

  grdCorbaData.ColCount := FQueryServer.FieldCount;
  grdCorbaData.RowCount := 0;
  j := 0;

  while not(FQueryServer.EOF) do
  begin
    inc(j);
    grdCorbaData.RowCount := j;
    CorbaData := (FQueryServer.Data);
    for i := 0 to FQueryServer.FieldCount - 1 do
    begin
      grdCorbaData.Cells[i + 1,j-1] := CorbaData[i];
    end;
    FQueryServer.Next;
  end;
end;

end.
```

Provided that you've launched the Smart Agent and the server is running where the Smart Agent can see it, you can now run this application and retrieve data from our CORBA server!

Building a Late-Bound CORBA Client

We're now going to modify our CORBA client so that it uses late binding to communicate with the remote interface. In CORBA we use what's called the *Dynamic Invocation Interface* (DII). Late binding is not necessary here because both the server and client were developed with Delphi. However, it's a useful technique to learn if you wish to easily use CORBA servers developed in other languages.

First, we can remove the stub and skeleton unit from the `uses` clause of our form's unit. Remember that if the server had been written in Java (for example), this would not be available for you to use anyway.

Second, our client now has no knowledge of the interface `IQueryServer`. Therefore, we'll change the data type of the encapsulated `FQueryServer` field from type `IQueryServer` to type `TAny`.

Third, we need to acquire a generic CORBA stub in a different manner than before. We can call the global Pascal `CorbaBind` method (from the `CorbaObj` unit) and pass the repository ID of the factory we're requesting. After we've acquired the factory, we can call the `CreateInstance` method of the factory that will return a generic interface. We can keep this interface in an `Any` and call late-bound methods from the reference. The complete source for the late-bound client is shown in Listing 27.7.

LISTING 27.7 The Late-Bound Query Server Client

```
unit ufrmCorbaClientLate;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, corbaObj, Grids;

type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    Label2: TLabel;
    edtDatabase: TEdit;
    Label3: TLabel;
    edtUserName: TEdit;
    Label4: TLabel;
    edtPassword: TEdit;
    Button5: TButton;
    GroupBox2: TGroupBox;
    memoSQL: TMemo;
```

continues

LISTING 27.7 Continued

```
    GroupBox3: TGroupBox;
    Button6: TButton;
    grdCorbaData: TStringGrid;
    procedure ConnectClick(Sender: TObject);
    procedure ExecuteClick(Sender: TObject);
  private
    { Private declarations }
    FQueryServer: TAny;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.ConnectClick(Sender: TObject);
var
  Factory: TAny;
  User, Pass: WideString;
begin
  Factory := CorbaBind('IDL:SimpleCorbaServer/QueryServerFactory:1.0');
  FQueryServer := Factory.CreateInstance('');
  User := WideString(edtUserName.Text);
  Pass := WideString(edtPassword.Text);
  FQueryServer.Login(WideString(edtDatabase.Text),User,Pass);
end;

procedure TForm1.ExecuteClick(Sender: TObject);
var
  i,j: integer;
  CorbaData : OLEVariant;
begin
  FQueryServer.Set_SQL((memoSQL.Text));
  FQueryServer.Execute;

  grdCorbaData.ColCount := FQueryServer.Get_FieldCount;
  grdCorbaData.RowCount := 0;
  j := 0;

  while not(FQueryServer.Get_EOF) do
```

```
  begin
    inc(j);
    grdCorbaData.RowCount := j;
    CorbaData := FQueryServer.Data;
    for i := 0 to FQueryServer.Get_FieldCount - 1 do
    begin
      grdCorbaData.Cells[i + 1,j-1] := CorbaData[i];
    end;
    FQueryServer.Next;
  end;
end;

end.
```


You'll notice a couple other changes in the source code for the late-bound client.

IDL does not support the notion of "properties" as in COM. When we use early binding, we can get away with this because the compiler simply resolves to the address of the getter or setter method for the property. When we use late binding, the DII does not know about a property so we must call the getter or setter method explicitly. For example, instead of reading `FieldCount`, we would call `Get_FieldCount`.

All DII parameters are passed as `Any` types that store the data type as well. Some values need to be explicitly cast in order for the data type of the `Any` to be set correctly. For example, sending a string value to the `Login` method's `Db` parameter will cause the `Any`'s type to be set to `varString`. This will result in a bad parameter error unless the string is cast to a `WideString` so that the `Any`'s type is set to `varOleStr` (a `WideString`).

Finally, in addition to the Smart Agent, the VisiBroker Interface Repository must be running somewhere on your network and the `IQueryServer` interface must be registered with the Interface Repository. The Interface Repository is like an online database that allows the ORB to look up interface information for use with DII. The VisiBroker Interface Repository can be launched from the command line using the command

```
IREP [-console] IRname [file.idl]
```

The only required argument here is `IRname`. Because multiple Interface Repository instances can be running, this one needs to be identified somehow. The `-console` argument specifies whether the Interface Repository runs in console mode (the default is GUI mode), and the `file.idl` argument can specify an initial IDL file to be loaded when the repository starts. Additional IDL files can be loaded using the menu option (if running as GUI) or by running the `idl2ir` utility.

Cross-Language CORBA

At the time of writing, an Inprise-supplied `Idl2Pas` compiler is still not present in Delphi; however, a pre-release version of such a tool does currently exist. In this section, we will discuss the steps required to manually early-bind to a CORBA server written in another language as well as take an introductory look at the forthcoming `Idl2Pas` compiler.

Hand-Marshaling a Java CORBA Server

The following example uses a very simple CORBA server constructed in Java (JBuilder) that's to be called from a Delphi application. The IDL for the CORBA server is shown in Listing 27.8.

LISTING 27.8 The IDL for a Simple Java Server

```
module CorbaServer {
  interface SimpleText {
    string setText(in string txt);
  };
};
```

Provided the CORBA server has been registered with the Interface Repository, Delphi can easily access the server using DII (this code is shown in Listing 27.9 in the `btnDelphiTextEarly` method).

In order to early-bind without an `Idl2Pas` compiler, we must hand-code our own stub class to perform the marshaling code. Although this is not exactly rocket science, it can be quite tedious and error-prone for a large number of methods. We must also register the stub class and the interface for the stub class with the proper Delphi mechanisms. Listing 27.9 contains the entire code.

LISTING 27.9 The Code for Accessing a Java Server from the Delphi Client (Early and Late Bound)

```
unit uDelphiClient;

interface

uses
  Windows, Messages, SysUtils, CorbInit, CorbaObj, orbpas, Classes,
    Graphics, Controls, Forms, Dialogs,  StdCtrls;

type

  ISimpleText = interface
  ['{49F25940-3C3C-11D3-9703-0000861F6726}']
```

```
    function SetText(const txt: String): String;
  end;

  TSimpleTextStub = class(TCorbaStub, ISimpleText)
  public
    function SetText(const txt: String): String;
  end;

  TForm1 = class(TForm)
    edtDelphiText: TEdit;
    btnDelphiTextLate: TButton;
    btnDelphiTextEarlyClick: TButton;
    edtResult: TEdit;
    procedure btnDelphiTextLateClick(Sender: TObject);
    procedure btnDelphiTextEarlyClickClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.btnDelphiTextLateClick(Sender: TObject);
var
  JavaServer: TAny;
begin
  JavaServer := ORB.Bind('IDL:CorbaServer/SimpleText:1.0');
  edtResult.Text := JavaServer.setText(edtDelphiText.text);
end;

{ TSimpleTextStub }

function TSimpleTextStub.SetText(const txt: String): String;
var
  InBuf: IMarshalInBuffer;
  OutBuf: IMarshalOutBuffer;
begin
  FStub.CreateRequest('setText',True,OutBuf);
  OutBuf.PutText(pchar(txt));
  FStub.Invoke(OutBuf, InBuf);
```

continues

LISTING 27.9 Continued

```
  Result := UnmarshalText(InBuf);
end;

procedure TForm1.btnDelphiTextEarlyClickClick(Sender: TObject);
var
  JavaServer: ISimpleText;
begin
  JavaServer := CorbaBind(ISimpleText) as ISimpleText;
  edtResult.Text := JavaServer.SetText(edtDelphiText.text);
end;

initialization
  CorbaStubManager.RegisterStub(ISimpleText, TSimpleTextStub);
  CorbaInterfaceIDManager.RegisterInterface(ISimpleText,
    'IDL:CorbaServer/SimpleText:1.0');

end.
```

You will notice that the above code looks very similar to the code generated by the Type Library Editor when we create a CORBA object within Delphi. We have added our own descendant of `TCorbaStub` that will serve to provide client-side marshaling. Note that it is not necessary to descend from `TCorbaDispatchStub` because the Type Library Editor is not involved here. Next we implement our custom stub to marshal the parameters to and from the CORBA marshaling buffer interfaces: `IMarshalInBuffer` and `IMarshalOutBuffer`. These interfaces contain convenient methods for reading and writing various data types to the buffers. Consult the Delphi 5 online help for more information on using these methods. Finally, we need to register our custom stub and our interface with the Delphi CORBA framework. This code is shown in the `initialization` part of our unit.

The Inprise Idl2Pas Compiler

As evident from the code in Listing 27.9, hand-marshaling a large CORBA object would require a great deal of work. The solution to this problem lies in the availability of an `Idl2Pas` compiler that can automatically generate the appropriate marshaling code for our stub. By the time you read this chapter, such a tool should be available from Inprise. We will conclude this section by taking a brief look at the current per-release version of `Idl2Pas`.

The Inprise `Idl2Pas` compiler is implemented in Java and therefore requires a Java VM to be installed on your development machine. A suitable Java Runtime Environment (JRE) is provided when you install Delphi 5. The current pre-release `Idl2Pas` compiler is not yet integrated into the Delphi IDE, so we must invoke the compiler from the command line using the supplied `Idl2Pas.bat` batch file. The command necessary to invoke `Idl2Pas` on `SimpleText.idl` and store the generated files in `c:\idl` would look as follows:

```
IDL2PAS -root_dir c:\idl SimpleText.idl
```

The `Idl2Pas` compiler will generate two files in the specified directory, named after the module name included in the `idl` file. For our example, `CorbaServer_i.pas` will contain the Pascal declarations for the `idl` interfaces and is shown in Listing 27.10.

LISTING 27.10 Interface Definitions Generated from `Idl2Pas`

```
unit CorbaServer_i;

// This file was generated on 4 Nov 1999 17:58:12 GMT by version
// 01.09.00.A2.032c of the Inprise VisiBroker idl2pas CORBA IDL compiler.

//Delphi Pascal unit CorbaServer_i for the CorbaServer IDL module.
// The purpose of this file is to declare the interfaces and variables used in
// the associated client (CorbaServer_c)
// and/or server         (CorbaServer_s) units.

//This unit contains the pascal interface code for IDL module CorbaServer.

(* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                      SimpleText.idl", line 1
** IDL Name       : module
** Repository Id  : IDL:CorbaServer:1.0
** IDL definition :
*)

interface

uses
  CORBA;

type
  //These forward references have been supplied to resolve dependencies between
  //the following interfaces.
  SimpleText = interface;
  //These interface definitions were generated from the IDL from which this
  //unit originated.


  //Signature for the "CorbaServer_i.SimpleText" interface derived from the IDL
  //interface "SimpleText".

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                        SimpleText.idl", line 2
  ** IDL Name       : interface
```

continues

LISTING 27.10 Continued

```
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
  *)
  SimpleText = interface
    ['{C8864064-C211-B145-29DB-CD5119D884CD}']

    //Interface methods representing IDL operations.

    (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                          SimpleText.idl", line 3
    ** IDL Name       : operation
    ** Repository Id  : IDL:CorbaServer/SimpleText/setText:1.0
    ** IDL definition :
    *)
    function   setText (const txt : AnsiString): AnsiString;
  end;

implementation


  //The implementation code (if any) is located in the associated _C file.

initialization


end.
```

The second generated file, `CorbaServer_c.pas`, contains the implementation code for the stub class as well as a helper object (`TSimpleTextHelper`) that facilitates the passing of non-simple data type such as structs, unions, and user-defined data types. The generated implementation code is shown in Listing 27.11.

LISTING 27.11 Stub and Helper Classes Generated from `Idl2Pas`

```
unit CorbaServer_c;

// c:\icon99\MultiLanguage\MyProjects\CorbaServer\SimpleText.idl.

//Delphi Pascal unit CorbaServer_i for the CorbaServer IDL module.
// The purpose of this file is to implement the client-side classes (stubs)
// required by the associated Interface unit (CorbaServer_i).
// This unit must be matched with it's associated skeleton unit on the server
// side.
```

```
//This unit contains the stub code for IDL module CorbaServer.

(* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                      SimpleText.idl", line 1
** IDL Name       : module
** Repository Id  : IDL:CorbaServer:1.0
** IDL definition :
*)

interface

uses
  CORBA,
  CorbaServer_i;

type
  //These forward references have been supplied to resolve dependencies between
  //the following interfaces.
  TSimpleTextHelper = class;
  TSimpleTextStub = class;
  //These stub and helper interfaces were generated from the IDL from which
  //this unit originated.

  //Pascal helper class "CorbaServer_c.TSimpleTextHelper" for the Pascal
  //interface "CorbaServer_i.SimpleText".

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                        SimpleText.idl", line 2
  ** IDL Name       : interface
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
  *)

  TSimpleTextHelper = class
    class procedure Insert(const A: CORBA.Any;
      const Value: CorbaServer_i.SimpleText);
    class function  Extract(const A: CORBA.Any): CorbaServer_i.SimpleText;
    class function  TypeCode: CORBA.TypeCode;
    class function  RepositoryId: string;
    class function  Read(const Input: CORBA.InputStream):
      CorbaServer_i.SimpleText;
    class procedure Write(const Output: CORBA.OutputStream;
      const Value: CorbaServer_i.SimpleText);
    class function  Narrow(const Obj: CORBA.CORBAObject; IsA: Boolean = False):
      CorbaServer_i.SimpleText;
    class function  Bind(const InstanceName: string = '';
```

continues

LISTING 27.11 Continued

```
      HostName : string = ''): CorbaServer_i.SimpleText; overload;
    class function  Bind(Options: BindOptions;
      const InstanceName: string = ''; HostName: string = ''):
      CorbaServer_i.SimpleText; overload;
  end;

  //Pascal stub class "CorbaServer_c.TSimpleTextStub supporting the Pascal
  //interface "CorbaServer_i.SimpleText".

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                       SimpleText.idl", line 2
  ** IDL Name       : interface
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
  *)

  TSimpleTextStub = class(CORBA.TCORBAObject, CorbaServer_i.SimpleText)
  public

    (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                         SimpleText.idl", line 3
    ** IDL Name       : operation
    ** Repository Id  : IDL:CorbaServer/SimpleText/setText:1.0
    ** IDL definition :
    *)
    function setText ( const txt : AnsiString): AnsiString; virtual;

  end;

implementation
  //These stub and helper implementations were generated from the IDL from
  //which this unit originated.

  //Implementation of the Pascal helper class "CorbaServer_c.TSimpleTextHelper"
  //supporting the Pascal interface "CorbaServer_i.SimpleText.

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                       SimpleText.idl", line 2
  ** IDL Name       : interface
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
  *)

  class procedure TSimpleTextHelper.Insert(const A: CORBA.Any;
    const Value: CorbaServer_i.SimpleText);
```

```
begin
  //TAnyHelper.InsertObject(Value);
end;

class function TSimpleTextHelper.Extract(const A: CORBA.Any):
  CorbaServer_i.SimpleText;
begin
  //TAnyHelper.ExtractObject as CorbaServer_i.SimpleText;
end;

class function TSimpleTextHelper.TypeCode: CORBA.TypeCode;
begin
  Result := ORB.CreateInterfaceTC(RepositoryId, 'CorbaServer_i.SimpleText');
end;

class function TSimpleTextHelper.RepositoryId: string;
begin
  Result := 'IDL:CorbaServer/SimpleText:1.0';
end;

class function TSimpleTextHelper.Read(const Input: CORBA.InputStream):
  CorbaServer_i.SimpleText;
var
  Obj: CORBA.CORBAObject;
begin
  Input.ReadObject(Obj);
  Result := Narrow(Obj, True)
end;

class procedure TSimpleTextHelper.Write(const Output: CORBA.OutputStream;
  const Value: CorbaServer_i.SimpleText);
begin
  Output.WriteObject(Value as CORBA.CORBAObject);
end;

class function TSimpleTextHelper.Narrow(const Obj: CORBA.CORBAObject;
  IsA: Boolean): CorbaServer_i.SimpleText;
begin
  Result := nil;
  if (Obj = nil) or (Obj.QueryInterface(CorbaServer_i.SimpleText, Result) = 0)
    then Exit;
  if IsA and Obj._IsA(RepositoryId) then
    Result := TSimpleTextStub.Create(Obj);
end;
```

continues

LISTING 27.11 Continued

```
  class function TSimpleTextHelper.Bind(const InstanceName: string = '';
    HostName: string = ''): CorbaServer_i.SimpleText;
  begin
    Result := Narrow(ORB.bind(RepositoryId, InstanceName, HostName), True);
  end;

  class function TSimpleTextHelper.Bind(
    Options: BindOPtions; const InstanceName: string = '';
    HostName: string = ''): CorbaServer_i.SimpleText;
  begin
    Result := Narrow(ORB.bind(RepositoryId, Options, InstanceName, HostName),
      True);
  end;
  //Implementation of the Pascal stub class "CorbaServer_c.TSimpleTextStub"
  //supporting the Pascal "CorbaServer_i.SimpleText" interface.


  //Implementation of Interface methods representing IDL operations.

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                        SimpleText.idl", line 3
  ** IDL Name       : operation
  ** Repository Id  : IDL:CorbaServer/SimpleText/setText:1.0
  ** IDL definition :
  *)
  function   TSimpleTextStub.setText ( const txt : AnsiString): AnsiString;
  var
    Output: CORBA.OutputStream;
    Input : CORBA.InputStream;
  begin
    inherited _CreateRequest('setText',True, Output);
    Output.WriteString(txt);
    inherited _Invoke(Output, Input);
    Input.ReadString(Result);
  end;

initialization

  //These stub and helper initialization calls were generated from the IDL from
  //which this unit originated.

  //Initialization of the Pascal helper class "CorbaServer_c.TSimpleTextStub".

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                        SimpleText.idl", line 2
```

```
  ** IDL Name       : interface
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
  *)
  CORBA.InterfaceIDManager.RegisterInterface(CorbaServer_i.SimpleText,
    CorbaServer_c.TSimpleTextHelper.RepositoryId);

  //Initialization of the CorbaServer_c.TSimpleTextStub interface stub for the
  //CorbaServer_i.SimpleTextInterface.

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                        SimpleText.idl", line 2
  ** IDL Name       : interface
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
  *)
  CORBA.StubManager.RegisterStub(CorbaServer_i.SimpleText,
    CorbaServer_c.TSimpleTextStub);

end.
```


You may notice that the marshaling code contained within the `setText` method of the generated code differs slightly from the code we wrote to hand-marshal this same interface. This is because the `Idl2Pas` tool uses a different DLL to provide ORB/Pascal access (`OrbPas33.dll`) and provides two new Pascal units that supplement the Delphi CORBA framework (`Corba.pas`, `OrbPas30.pas`). These new additions will peacefully coexist and not replace the units and libraries currently shipping with Delphi 5.

The release of the Inprise `Idl2Pas` compiler will help you to simplify some of the more difficult CORBA tasks such as calling servers written in other languages, marshaling non-simple data types, and handling custom user exceptions.

Deploying the VisiBroker ORB

The VisiBroker ORB requires a runtime deployment license. Although Delphi 5 Enterprise includes the VisiBroker services in the development environment, you should check with Inprise before actually deploying your solutions.

ORB services will need to be deployed on server machines as well as client computers. As mentioned previously, many of the other VisiBroker services (such as `osagent`, `irep`, and `oad`) can be executing anywhere in your local network; therefore, deployment of these services may not be necessary on all machines that are using ORB software. As mentioned, the primary C++ ORB used with Delphi is the dynamic link library `orb_br.dll`. A common problem reported

with Windows VisiBroker installations is that the DOS path is not correctly defined. This must be done in order for the system to locate the ORB DLLs. Also, remember that Delphi uses a special "thunking" layer (`orbpas50.dll`) in order to map IDL interfaces to Delphi interfaces and provide other access to the C++ ORB. `Orbpas50.dll` must also be deployed for all Delphi 5 CORBA installations.

Summary

In this chapter we've examined the basics of CORBA development with Delphi 5. We've created both CORBA clients and servers as well as experimented with both early and late binding. We've also looked at what's required in order to early-bind to a CORBA server written in another language. Finally, we have taken a sneak peek at the Inprise `Idl2Pas` compiler and demonstrated how the release of this tool will help simplify CORBA development with Delphi.