DDObjects

Version 1.2.1

Developer's Handbook
and Reference

www.ddobjects.de

www.ddobjects.de

stefan@ddobjects.de

Table of Contents

Introduction

DDObjects is a remoting framework to be used with Borland Delphi 5-7 as well as Delphi 2005 to Delphi Rio (32bit and 64bit) and Borland Kylix 3. With only minor changes, DDObjects should also be usable with C++ Builder 6 and above.

It's development was begun out of personal interest for technologies like DCOM, RMI, Corba etc. A main goal in the development of DDObjects has not only been to keep the code one has to implement in order to utilize DDObjects as simple as possible but also very close to Delphi's usual style of event-driven programming. DDObjects doesn't mimic other implementations as DCOM or Corba, which are generalized to a least common denominator but makes use of Delphis rich type system including Objects, Exceptions, Records, Sets and Enumerations.

The initial version was developed in November 2003 and used to implement a multi tier accounting system connected to several Oracle databases and serves about 50 clients executing as much as 500,000 calls each day and is running smoothly since then. After that some other multi tier-projects have been implemented using DDObjects.

DDObjects supports

- Remote method calls
- Server callbacks
- Asynchronous calls
- Asynchronous server callbacks
- Stateful and stateless objects
- User sessions, Session classes
- Sets, Enumerations, Record
- Compression
- SSL encryption
- Virtual Stubs and much more...

Contributors

Many thanks to Indra Gunawan for his suggestions as well as to Markus Landwehr who has provided the code which has been adapted within the unit DDOFirewall.pas. Many thanks to Primož Gabrijelčič (http://http://17slon.com/gp) who provided some code to keep compatibility with Delphi 5. Special thanks go to Hallvard Vassbotn who has provided a very fast replacement for StringReplace. DDObjects includes zlib 1.2.3. See http://www.zlib.net for more details.

I also would like to say “Thank you!” to Lukas Gebauer (http://synapse.ararat.cz) for his friendly and competent support. Last but not least I would like to thank Federico Simonetti (http://www.liveye.net) for his suggestions and motivation as well as Alexander Bauer (http://www.familie-bauer.info) for his help porting DDObjects to Kylix.

Changes in Version 1.1 and 1.2

Version 1.1 introduces an exchangeable connection layer which enables usage of 3rd party socket libraries. Support for Borland Socket Components, Synapse, Indy 9 and Indy 10 is available. Due to the pluggable architecture of Synapse and Indy, DDObjects can take advantage of their support for SSL encryption and security using OpenSSL, Eldos SecureBlackbox and others.

By default the packages compile with support for Borland Socket Components only. If you like to use Synapse and/or the Indy socket library please open the include file DDObjects.inc and remove the dot (.) In front of each conditional define as shown here:

```
{.$DEFINE DDOConIndy9}
{.$DEFINE DDOConIndy10}
{.$DEFINE DDOConSynapse}
{$DEFINE DDOConScktComp}
```

After that you can rebuild all packages and install the design time package into the IDE. For further details please read the chapter “Installation”.

Depending on the previous made selection several new components will be available which act as adapters for the socket libraries above:

Indy 9	TDDOClientIndy9Connection, TDDOServerIndy9Connection
Indy 10	TDDOClientIndy10Connection, TDDOServerIndy10Connection
Synapse	TDDOClientSynapseConnection, TDDOServerSynapseConnection
Borland	TDDOClientScktCompConnection, TDDOServerScktCompConnection

The client connection components, which inherit from TDDOClientConnection will need to be assigned to a TDDORequester component while the server connection components, which inherit from TDDOServerConnection will need to be assigned to a TDDOListener resp. TDDONameServer component in order to utilize the specific socket library.

To ease the migration from prior versions, it is not necessary to assign any connection components as described above. If no connection component has been explicitly selected, all components will internally create appropriate ones on their own.

Upgrading from versions prior to 1.1

Due to the exchangeable connection layer which leaves support for SSL to the used socket library, the property UseSSL of the components TDDOListener and TDDORequester has been removed. To upgrade your existing applications you need to open each DFM file which contains one of these components in your favourite text editor and remove the declaration as shown in the next code snippet:

```
object DDORequester1: TDDORequester
  Host = 'xyz'
  Port = 8888
  Cursor = crDefault
  Synchronized = False
  UseSSL = False  // <-- remove this line
  Left = 240
  Top = 16
end
```

Support for FreePascal

This version contains files and packages for Lazarus / FreePascal. Only basic remoting scenarios are working including callbacks and asynchronous calls. The standard demo project as well as the unit tests have been ported. If you would like to contribute porting DDObjects to FPC please send me an e-mail. Actually I am looking for someone who is more skilled using Lazarus as well as it's debugger than me :-)

Enhancements in Version 1.1 and 1.2

A feature which makes DDObjects unique among other remoting frameworks is the concept of “Virtual Stubs” which enables polymorphism to be used on the server side. A sample which demonstrates this unique concept will be available for download soon at www.ddobjects.de

The sourcecode wizard now does have an additional checkbox labelled “Safe” for each method. Checking this option will change the implementation of the generated proxy class so that the concerned method will be wrapped into a try/except block returning a boolean value reflecting success of the call.

Free Requests enable remote calls without the use of any generated proxy class. A sample which demonstrates this concept will be available for download soon at www.ddobjects.de

The DDOListener and DDORequester components do have a new property Encoding which determines the encoding of binary data being send by DDObjects. Beneath Hex- and Base64 Encoding you can select custom encoding: using the events OnCustomEncoding and OnCustomDecoding you can add your own implementation.

Installation

System Requirements

DDObjects supports Delphi 5 to 7 and Delphi 2005 up to Delphi Rio (32bit and 64bit) as well as Borland Kylix 3. DDObjects is running on Windows NT 4, Windows 2000, Windows XP and above. Windows 98 is only partially supported (see FAQ for more details).

Trial Version without Sources

Unzip the package to a folder of your choice. The folder contains a subfolder Lib which contains the precompiled DCU respectively OBJ files for each supported version of Delphi. Add the appropriate folder to your library path and install the contained package dclDDObjects within your IDE selecting Component|Install Packages. A new tabsheet labelled DDObjects, containing several new components, should appear.

If you have any problems in proceeding these steps, check your library- as well as your windows search path which should include the directory containing the package files.

Registered Version incl. Sources

The steps to install DDObjects are the same as described above. If you like to build DDObjects yourself you find the complete source code in the sub folder Source. The sub folder Packages contains the package files. Users who like to use DDObjects support for TDataSet need to open the file DDObjects.inc and activate the conditional define DDODataSet which is disabled by default to support users of Delphi's personal editions.

DDObjects for Kylix

Support for Kylix 3 is available within the registered Version only. The installation procedure is the same as already described above. However: for Kylix the DDObjects runtime package has been split into two packages: DDObjectsK3 and DDObjectsVclK3 to avoid having dependencies to the Qt library for the core library and components of DDObjects. Only the last package which contains some additional, nevertheless non essential GUI components does depend on Qt.

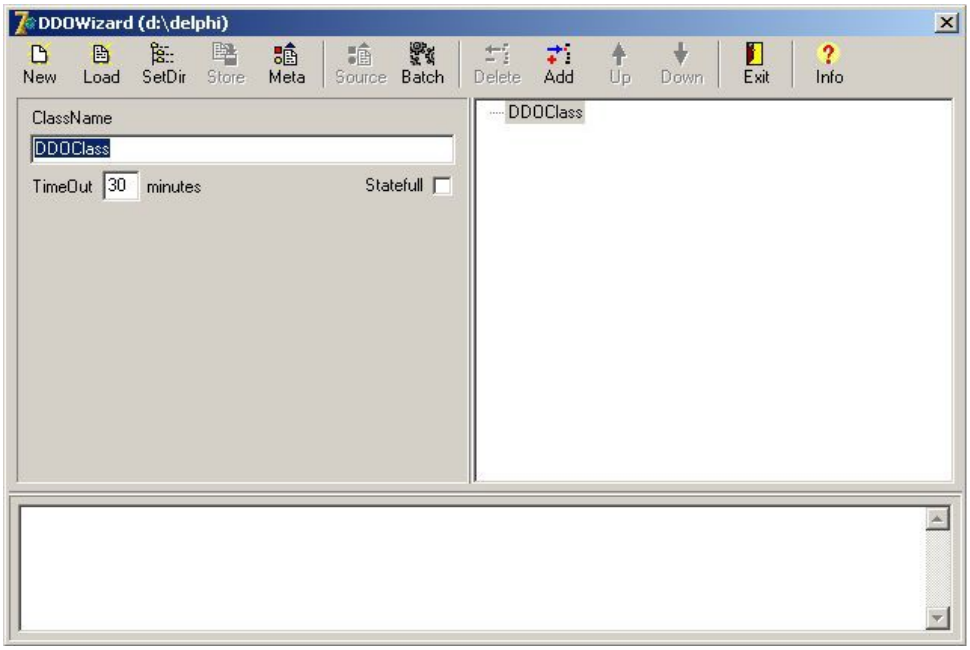
DDObjects for Free Pascal

Please note, that DDObjects for FreePascal depends on Synapse Socket Library which can be downloaded for free at http://synapse.ararat.cz/. Open and compile the file ddobjectsfpc.lpk which is available within the folder source/packages . After that you need to open, compile and install the package dclddobjectsfpc.lpk which is in the same folder. After Lazarus has been rebuilded and restarted the components should be available on a tabsheet labelled 'DDObjects'.

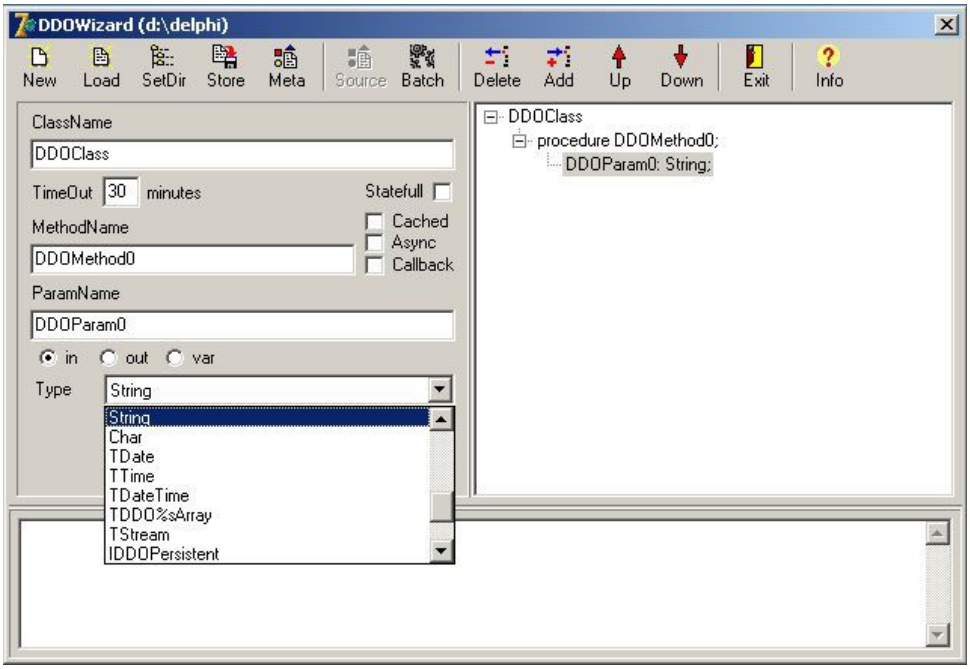
Tutorials and Implementation

DDO Wizard

Before starting with the tutorials a short introduction to the wizard, which will be your primary tool when developing with DDObjects. To start the wizard select File|New|Other, switch to the tab labelled DDObjects and select DDOWizard. The wizard will be shown as in this screenshot:



To define a new class, its methods and parameters click on the button labelled Add in the upper panel. If the class node is selected within the tree a new method will be added. If one of the methods is selected, a new parameter node will be added which is shown in this screenshot:

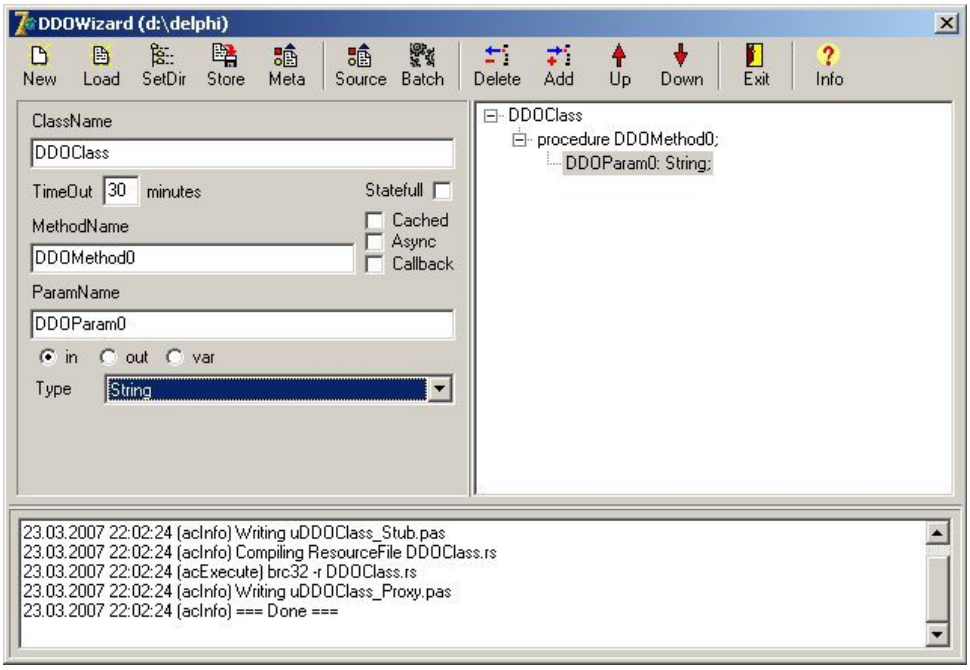


You can enter the name of the method or parameter, select whether the parameter is an in, out or var parameter and select its type. Some types like arrays and records need further refinement. Additional fields to enter these information will be shown accordingly. Nodes can be deleted, moved up and down using the buttons labelled Delete, Up and Down.

Before starting source code generation, the class definition needs to be stored to disk. Click on the button Store in order to do so. The file will be named as the class like DDOClass.xml in this case. To change an already defined class, you can load the definition, remove, change or add parameters and methods as appropriate and store the changes to file.

Only after the definition has been stored the button Source will be enabled!

The final step before leaving the wizard is the source code generation which can be started clicking on the button labeled Source. Although source code generation usually should be done in an instant the wizard shows each step it executes in the bottom memo as shown in the this screenshot:



This finishes this short introduction to the wizard. Most of the settings which appear above (TimeOut, Async etc.) are being described within the next tutorials and therefore are not mentioned here. If you do have any question or suggestions about this or any other tutorial don't hesitate to send an eMail to stefan@ddobjects.de

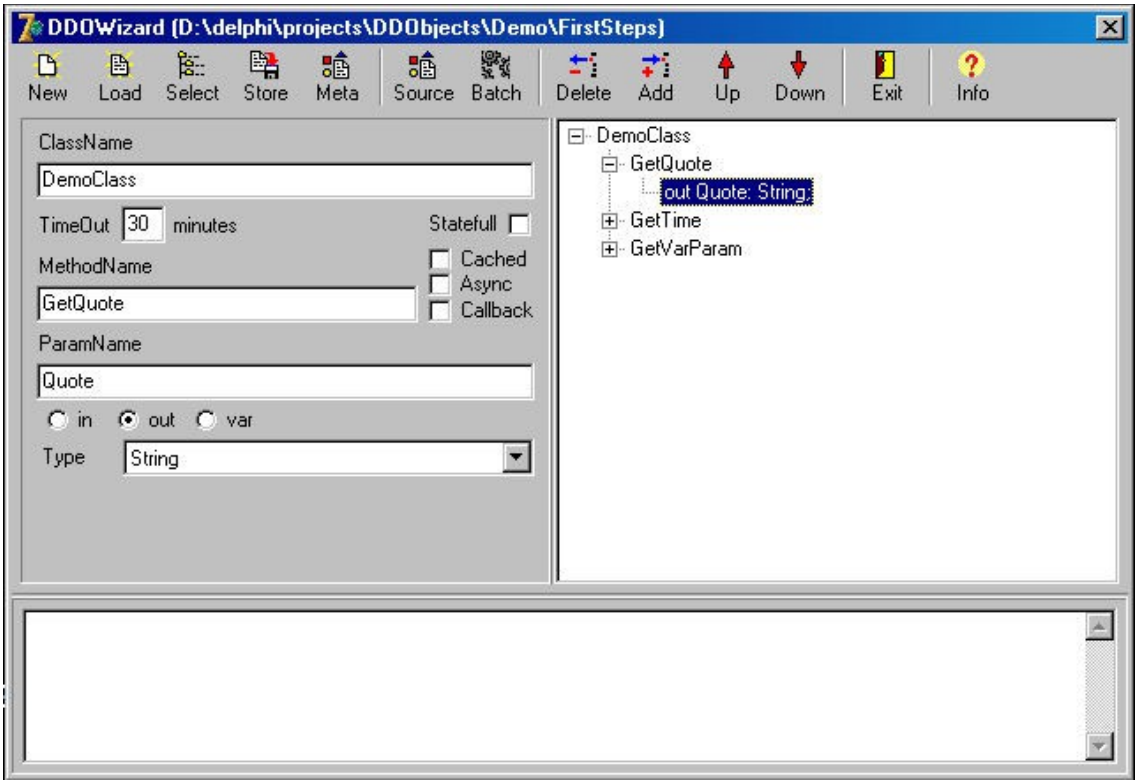
First Steps

Defining the Interface

We'll start defining the interface for the class we would like to implement. For the sake of simplicity we'll define three simple methods only:

```
procedure GetQuote(out Quote: String);
procedure GetTime(out DateTime: TDateTime);
procedure GetVarParam(var AString: String);
```

Select File|New|Other, select the tabsheet labelled 'DDObjects' and doubleclick on the icon (DDOWizard) to run the integrated wizard. Within that wizard, provide an appropriate name to be used for the class name (here the name 'DemoClass' serves as an example) and add these methods along with their parameters and types as shown in the following screen shot.



Leave the checkboxes labeled 'Stateful', 'Cached', 'Callback and 'Async' unchecked and don't change the value for 'TimeOut'. These settings are not relevant at the moment and will be discussed in an advanced tutorial. Select a directory where this information (expressed in XML) and the generated sources shall be stored.

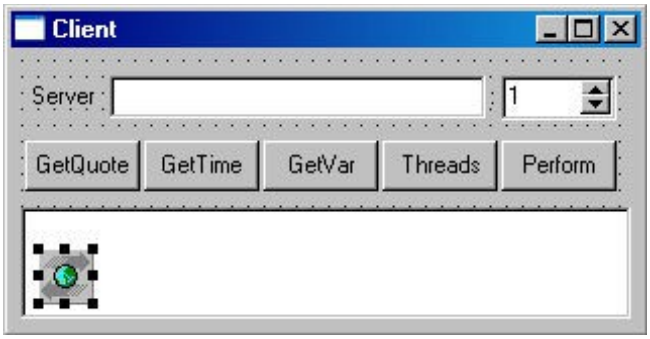
Generating Sources

Click on the button labeled 'Sources' and wait a few seconds. You'll be notified as soon as the wizard has completed its task. The wizard will generate two units with names depending on the

class name: uDemoClass_Proxy.pas will be included within the client, while the unit uDemoClass_Stub.pas will be included within the server. These units contain the classes that implement the previously defined interface.

Creating the Client

We'll build the client first as this is fairly straightforward. Create a new application and construct the main form's GUI. Add the unit which has been generated earlier (uDemoClass_Proxy.pas) to the application. Select the tabsheet labelled 'DDObjects' on the component-palette and drop a DDORequester on your form.



Some settings and events of the component (e.g. server and port) can be configured using the Object Inspector. We'll leave all settings at their default values and do not add any events for now. Create an instance of TDemoClass within the OnCreate handler of the form, and free it within OnDestroy:

```
procedure TClientMainForm.FormCreate(Sender: TObject);
begin
  FDemoClass := TDemoClass.Create(DDORequester1);
end;

procedure TClientMainForm.FormDestroy(Sender: TObject);
begin
  FreeAndNil(FDemoClass);
end;
```

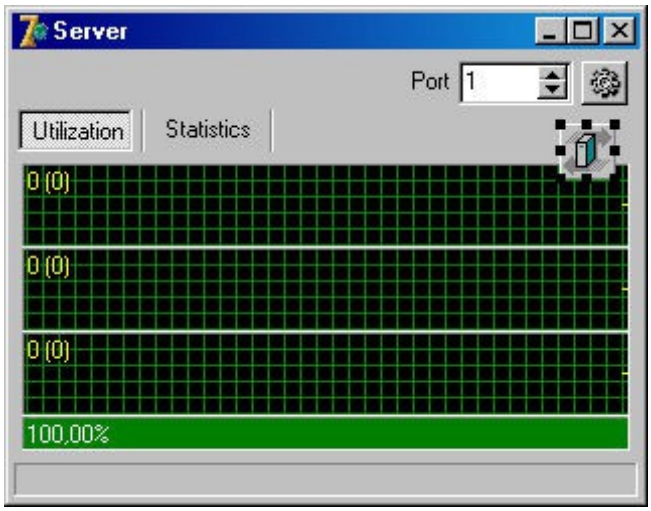
All that's left to do is to use this instance within the application. The button labeled GetQuote in the above screenshot has an OnClick event handler assigned that calls a method of the object. This call will be sent over the network and executed by the server (which we'll create in the next section); the server then sends the result back to us. All this is done with a single line of code:

```
procedure TClientMainForm.ButtonGetQuoteClick(Sender: TObject);
var
  AQuote: String;
begin
  FDemoClass.GetQuote(AQuote);
  MemoResult.Text := AQuote;
end;
```

Creating the Server

Let's start creating the server. This is not much more complicated than creating the client. However, advanced issues and scenarios dealing with multi-threading can be a challenging subject of their own. Although the server does use several threads to perform its task, this tutorial does not deal with these issues.

Create a new application and construct the main form's GUI. Once you are done add the unit which has been created by the wizard (uDemoClass_Stub.pas) to the application and use it within the form's code. Select the tabsheet labelled 'DDObjects' from the component palette and drop a DDOListener onto the form:



Open the unit uDemoClass_Stub.pas and locate the following snippet of code. Note that the class inherits from the abstract class TDemoClass_Stub which is defined and (partly) implemented within the same unit.

```
// ************************************************************
//  Use this comment block as a template to implement the stub
// ************************************************************
 {
   TDemoClass = class(TDemoClass_Stub)
   public
     procedure GetQuote(out Quote: String); override;
     procedure GetTime(out DateTime: TDateTime); override;
     procedure GetVarParam(var AString: String); override;
     procedure Initialize; override;
     destructor Destroy; override;
   end;
 }
```

Copy the provided code to the form unit, press Ctrl+Shift+c to perform code completion, and implement the methods as appropriate. Note that instead of overriding the constructor, the class offers a virtual method called Initialize which should be used instead. The destructor can be overridden as usual:

```
procedure TDemoClass.Initialize;
begin
  inherited;
  FQuotes := TStringList.Create;
  FQuotes.LoadFromFile('Quotes.txt');
end;

destructor TDemoClass.Destroy;
begin
  FreeAndNil(FQuotes);
  inherited;
end;
```

Once a client calls the method GetQuote on the proxy (as shown in the client code above), the call will be sent over the network, received by the DDOListener component, forwarded to the implementation in TDemoClass and finally sent back to the client which receives the result. You do not need to implement any specific code for this, as it happens automatically.

```
procedure TDemoClass.GetQuote(out Quote: String);
begin
  // The "+1" causes an exception to be raised from time to time.
  // This is intentional to show how DDObjects handles exceptions.
  Quote := FQuotes[Random(FQuotes.Count + 1)];
end;
```

Some simple steps are missing before we can use the server. At first, the DDOListener we've dropped onto the form needs to know about the newly created class. We do this by registering it:

```
procedure TServerMainForm.FormCreate(Sender: TObject);
begin
  DDOListener1.RegisterClass(TDemoClass);
end;
```

Finally the server needs to be activated:

```
procedure TServerMainForm.SpeedButtonStartClick(Sender: TObject);
begin
  DDOListener1.Active := True;
end;
```

The component offers several events which can be hooked into. The following code snippet shows one of these handlers which is triggered whenever an exception is raised (which happens from time to time in our deliberately faulty implementation of TDemoClass.GetQuote):

```
procedure TServerMainForm.DDOListener1ClientException(Sender: TObject;
  E: Exception; Method: String);
begin
  Log(Format('Exception: %s in %s', [E.ClassName, Method]));
end;
```

Starting the Demo

Build both projects and start both the client and the server application. If both applications run on the same machine and port 6666 is available, you can start immediately as those are the default settings. Otherwise you need to select an appropriate port within the server application and

configure host and port within the client application accordingly. Start the server by clicking the appropriate button.

The server contains a simple form which offers some basic logging facilities, showing the requests and results as they happen. Note that this slows down the application, so you can turn logging off. The client application contains several buttons, which will call the corresponding methods of the proxy and show the received results within a memo.

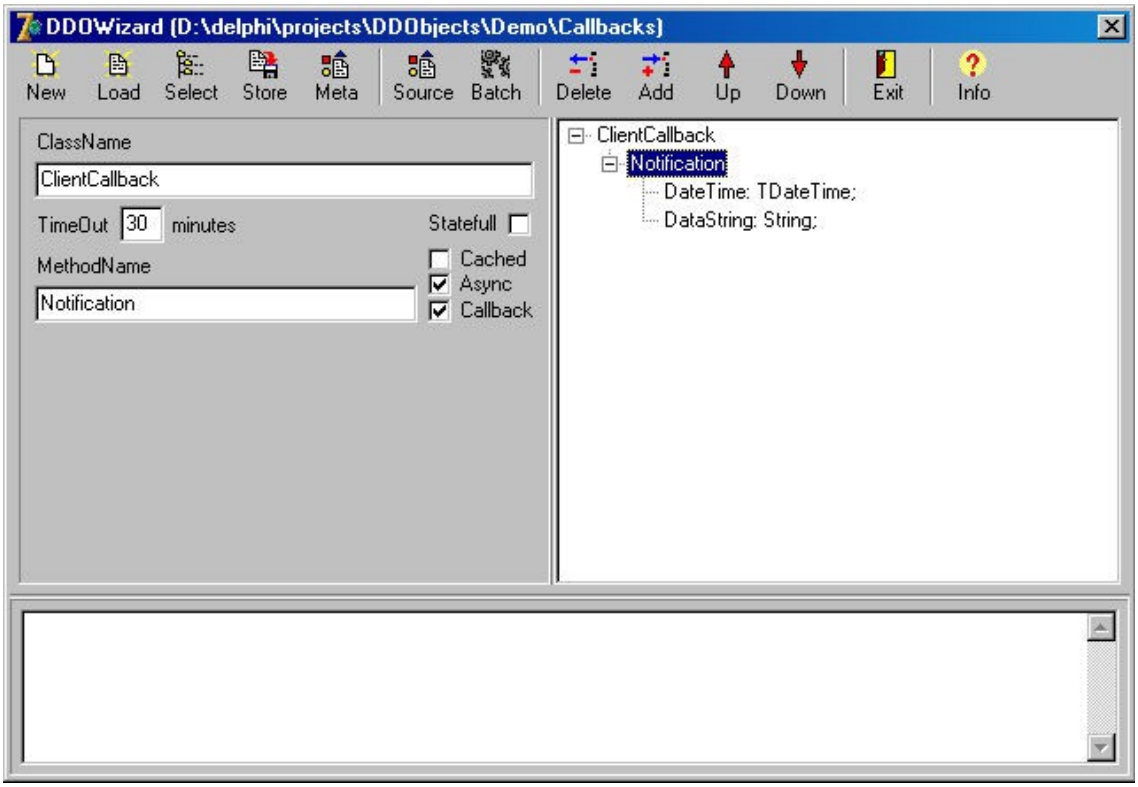
Implementing Callbacks

Defining the Interface

Again, we start by defining the interface. Select File|New|Other, select the tabsheet labeled 'DDObjects' and doubleclick the icon (DDOWizard) to run the integrated wizard. The class we define will be named be 'ClientSession' and offers three methods:

```
procedure LogOn(UserName: String; Password: String);
procedure LogOff;
procedure Notification(DateTime: TDateTime; DataString: String);
```

The most notable difference, compared to the first tutorial, is that we have selected the procedure Notification to be an 'Async Callback'. The difference between callbacks and async callbacks will be described at the end of this chapter. Stateful means, that objects of this class will always be associated to a specific client. The first time a client invokes a method of that object, a new instance will be created. Subsequent calls will be handled by this instance; it's exclusively used by that client.



Again we execute the source code generation. The wizard will generate five new units of which only two are of interest to us (the remaining three units will be used implicitly). These units define the classes implementing the interface, a wrapper- and a listener class as well as the procedural type which defines the callback.

Creating the Client

Again we start building the client first. Create a new application and construct the main form's GUI. Select the tabsheet labeled 'DDObjects' on the component-palette and drop a DDORequester on your form. Add the unit uClientSession_Proxy to the project and enclose it in the form's uses-clause. Note that generated units, which need to be added to the client, always end with '_proxy'. Open the unit uClientSession_Proxy.pas and locate the declaration of TClientSessionProxyStub. Notice the declaration of the property

```
property OnNotification: TClientSessionNotification
  read FClientSessionNotification write FClientSessionNotification;
```

The procedural type can be located within the unit uClientSessionCallbackListener_Stub.pas and introduces parameters we defined with the wizard. Any time a client receives an callback which has been generated by the server, this event handler will be triggered.

Actually there are just a few steps left to finish the client. Firstly we need to create an instance of the TClientSessionProxyStub and destroy it if we are done. We'll do this within the OnCreate and OnDestroy handler of the form. Next, we need to add an event handler with the same parameters of TClientSessionNotification and attach it to the instance. Finally, we need to subscribe at the server in order to receive notifications. This is being done by calling the method Subscribe. The parameter of Subscribe tells the server on which port we are listening for callbacks. In case we do provide a value of 0 the DDOListener component will select an appropriate one on it's own.

```
procedure TClientMainForm.FormCreate(Sender: TObject);
begin
  FClientSession := TClientSessionProxyStub.Create(DDORequester1);
  FClientSession.OnNotification := NotificationReceived;
end;

procedure TClientMainForm.FormDestroy(Sender: TObject);
begin
  FreeAndNil(FClientSession);
end;

procedure TClientMainForm.NotificationReceived(Sender: TObject;
  DateTime: TDateTime; Data: String);
begin
  MemoNotifications.Text := DateTimeToStr(DateTime) + #13#10 + Data;
end;

procedure TClientMainForm.ButtonSubscribeClick(Sender: TObject);
begin
  FClientSession.Subscribe(0)
end;
```

Note that the callback will not be executed within the MainThread, so you must care for synchronization. Although it's not recommended to access VCL components from within an other than the MainThread the above is safe as setting a Memo's text internally uses SendMessage which does force a thread context switch. Refer to the Windows API to learn more about this.

Creating the Server

Create a new application and construct the main form's GUI. Once you are done add the unit which has been created by the wizard (uClientSession_Stub.pas) to the project and use it within the form's code. Select the tabsheet labelled 'DDObjects' from the component palette and drop a TDDOListener as well as a TTimer onto the form.

Like in the first tutorial open the unit uClientSession_Stub.pas, copy the template to the main form's unit and implement the procedures LogOn and LogOff. The class we inherit from - TClientSession_ActiveStub - already contains the implementation of the procedure Notification which the server will call in order to notify it's clients.

The code to activate the server and to register the class will be omitted as it's almost identical to the one already described in the first tutorial.

We still need to implement the notifications. The listener offers a method called EnumStubs which takes a class reference, a callback as well as an user defined Integer as it's parameters. For each stub, which is of the given classtype, the callback (SendQuote) will be executed once. The stub can be casted to a TClientSession to invoke the method Notification:

```
procedure TServerMainForm.SendQuote(Stub: TDDOStubBase;
  UserData: Integer; var Continue: Boolean);
begin
  Assert(Stub is TClientSession);
  TClientSession(Stub).Notification(Now,
    FQuotes[Random(FQuotes.Count)], nil);
end;

procedure TServerMainForm.Timer1Timer(Sender: TObject);
begin
  DDOListener1.EnumStubs(TClientSession, SendQuote, 0);
end;
```

Callbacks and Asynchronous Callbacks

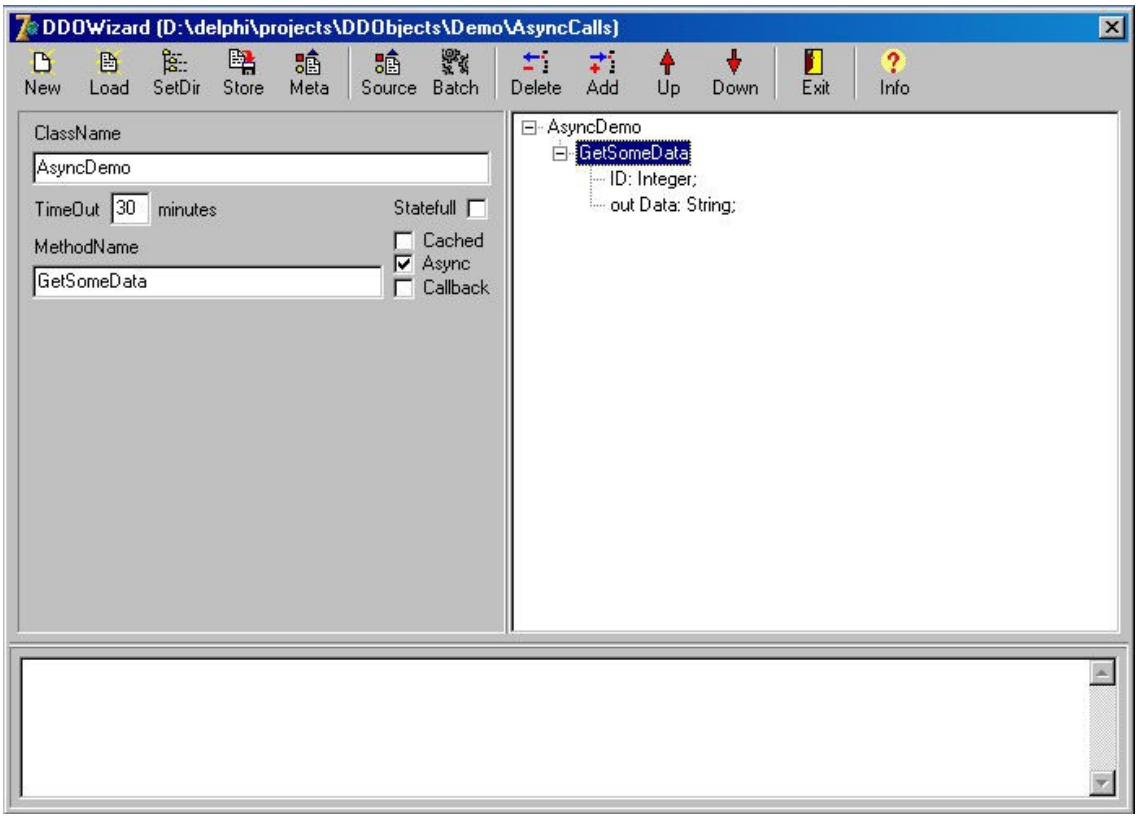
The difference between Callbacks and Asynchronous Callbacks is that the server will be blocked when executing Callbacks waiting for the client's result. As described within the next tutorial, Asynchronous Callbacks will be executed by another thread which will trigger a notification when the client's result has been received. The notification is optional and can be omitted. If you don't have special reasons not to do so, you should favour asynchronous callbacks over callbacks.

If you did not read the previous tutorials I do advise to do so first, as the remaining ones will be much more compact and shorter, leaving out some concepts and explanations which have been discussed before.

Asynchronous Calls

Asynchronous calls differ from callbacks by not having the result of the call at hand immediately but at a later time. Because the client should not be blocked waiting for the result, DDObjects will trigger an appropriate event as soon as the result is available. Therefore it's not necessary to poll or determine the state of the operation on the client side.

The class of this simple example contains only one method which is tagged as being Async. Actually, this is all you need to do in the wizard before executing code generation. The screenshot shows the necessary settings:



If we look at the server code we'll notice that it does not differ from previous examples:

```
TAsyncDemo = class(TAsyncDemo_Stub)
public
  procedure GetSomeData(ID: Integer; out Data: String); override;
end;

procedure TAsyncDemo.GetSomeData(ID: Integer; out Data: String);
begin
  Data := 'Some Data for ID ' + IntToStr(ID);
  Sleep(Random(5000) + 5000);  // simulate some "lengthy" process
end;
```

The situation will be slightly different with the client. First we'll have a look into the unit uAsyncDemo_Proxy.pas which has been generated by the wizard and examine the relevant differences:

```
type
  TGetSomeDataEvent = procedure (Sender: TDDOProxyBase;
    GUID: String; Data: String) of object;

  TAsyncDemo = class(TDDOProxyBase)
  ...
  public
    function GetSomeData(ID: Integer;
      Callback: TGetSomeDataEvent): String;
  ...
  end;
```

At first we'll notice the type TGetSomeDataEvent which has the parameters which represent the result returned by the server as well as an additional parameter named GUID. The proxy itself contains the method GetSomeData as we've defined it in the wizard excluding the parameters which are part of the server's result as well as an additional parameter of the above defined TGetSomeDataEvent type. We also notice that, in contrast to other implementations we've seen so far, the method returns a String.

Let's have a look at the client to see the usage of such an asynchronous call.

```
procedure TDemoClientMainForm.ButtonAsyncClick(Sender: TObject);
var
  lGUID: String;
begin
  lGUID := FProxy.GetSomeData(Random(99), GetSomeDataCallback);
  AddText(MemoRequests, lGUID);
end;

procedure TDemoClientMainForm.GetSomeDataCallback(Sender: TDDOProxyBase;
  GUID: String; Data: String);
begin
  DelText(MemoRequests, GUID);
  AddText(MemoResults, TimeToStr(Now));
  AddText(MemoResults, ' ' + Data);
end;
```

The method ButtonAsyncClick invokes the asynchronous call, passes a method pointer which matches the definition of TGetSomeDataEvent, namely GetSomeDataCallback, and adds the returned string to a memo. By this time you already might know about the usage of this value: it's a GUID which will help us identifying the callback. Please note, that this asynchronous call might be invoked several times therefore it might be necessary to collate a call within the callback. For this purpose the callback contains the additional parameter GUID which will be identical to the GUID which has been returned when invoking the call.

Be aware that the callback will not be executed within the context of the application's main thread, so you should care for synchronization unless the property TDDORequester.Synchronized has been set to true.

Type Safe Exception Handling

In contrast to other remoting systems which do not support a type safe exception handling (which means all exceptions raised within the server are being reported as e.g. ERemote, therefore loosing relevant information) DDObjects handles exceptions in it's own way:

Although unknown exceptions will still be raised as EDDORemote, one can register exception classes so they are known to DDObjects. By default, some of the most common exception classes are already registered:

- Exception
- EAssertionFailed
- EIntfCastError
- EExternal
- EAccessViolation
- EVariantError
- EOSError
- EInvalidCast
- EListError
- EStringListError
- EInvalidOperation

To register custom exceptions call DDOExceptionRegistry.RegisterException from the unit DDOExceptions with the exception class to register. The second parameter is a callback function which will be invoked to “enrich” the message of an exception before it will be transmitted to the client. For example, the default registered exception class EOSError defines the following callback:

```
procedure _EOSError(e: Exception; var AMessage: String);
begin
  Assert(e is EOSError);
  AMessage := AMessage + ' ErrorCode: ' + IntToStr(EOSError(e).ErrorCode);
end;
```

As this registration needs to be done for the server as well as for the client, it might be wise to put this code within a separate unit to be included in both applications.

Using Sessions

Sessions are an integral part of DDObjects. You do not need to take any actions or meet any special conditions to use them within your application. While stateful objects can store client specific values between different calls, this is not possible using stateless objects. Also there has been lack of a common way exchanging such values between different objects. Sessions facilitate the storage and exchange of user specific values between different calls and different server side objects.

To access the current user's session, the base class for server side objects (DDOStubBase) offers a protected property Session: TClientSession. The public interface of TClientSession is quite concise and enables checking for the existence of a given key as well as associating and retrieving values by key:

```
    procedure Lock;
    procedure Unlock;
    function Exists(Key: String): Boolean;
    property Value[Key: String]: Variant;
    procedure Clear;
```

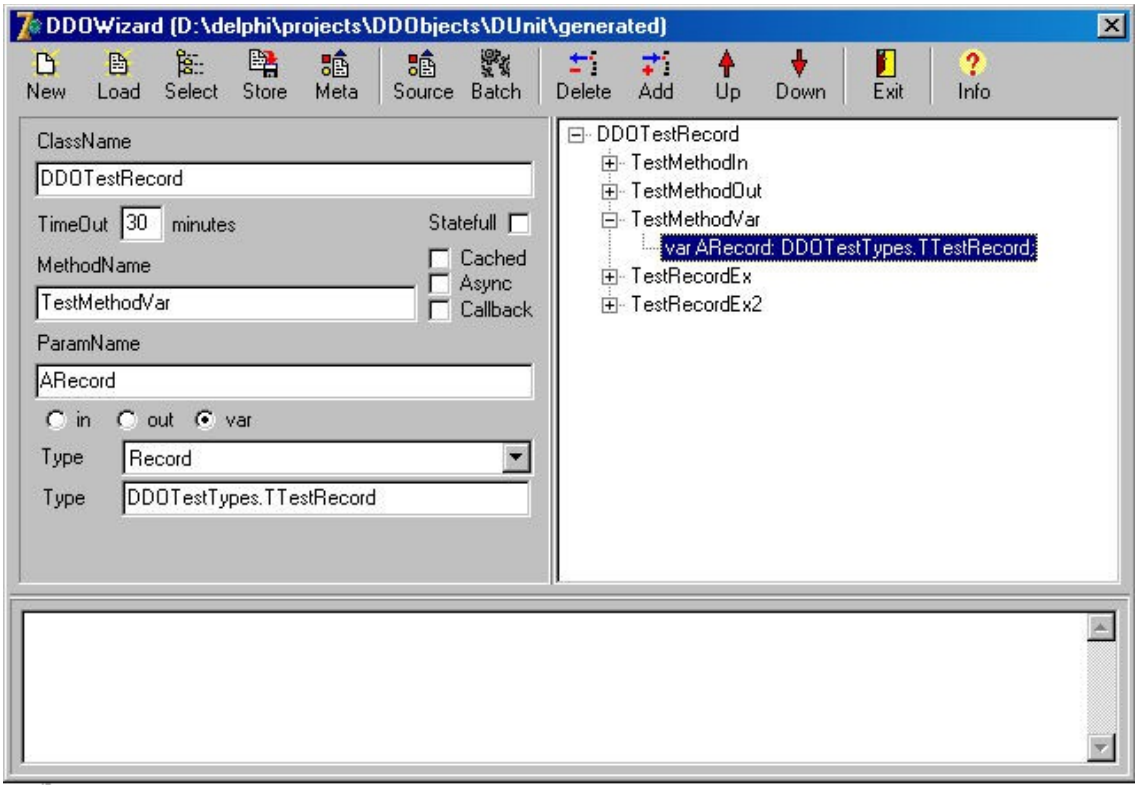
A short code snippet - taken from the demo project - should illustrate the above said. Please note that the session values are set within the class TDDOSessionLogin and being accessed and used within the class TDDOSessionTest.

```
procedure TDDOSessionLogIn.LogIn(UserName, Password: String);
begin
  if Password <> '1234567' then
    raise Exception.Create('Wrong password');
 // set session value(s)
  Session.Value['loggedin'] := True;
  Session.Value['username'] := UserName;
end;

procedure TDDOSessionLogIn.LogOff;
begin
 // set session value(s)
  Session.Value['loggedin'] := False;
end;

procedure TDDOSessionTest.GetData(out AText: String);
begin
 // check session value(s)
  if not Session.Exists('loggedin') or not Session.Value['loggedin'] then
    raise Exception.Create('You are not logged in');
  AText := 'Hello ' + Session.Value['username'] + '!';
end;
```

A session which has not been used within a configurable amount of time (see property TDDOListener.SessionTimeOut) will be invalidated and destroyed.

Records, Sets and Enumerations

DDObjects offers using native records, sets and enumerations as parameters. Within the DDOWizard you need to select “Record”, “Set” or “Enumeration” as parameter type and enter the **fully qualified type name** (name of the unit which contains the type definition and the name of the type separated by a dot) as shown in the next screen shot:



The unit will be used by the units generated by the DDOWizard on the client- as well as the server side. Therefore it might be wise to put the type definitions in a separate unit. Using records, sets or enumerations is not different from using any ordinary type. A short code snippet - taken from the DUnit tests - should illustrate this:

```
procedure TDDORemoteRecordTests.TestRecordVar;
var
  lRecord, lCompare: TTestRecord;
begin
  lRecord := GetDefaultRecord;
  Fproxy.TestMethodVar(lRecord);  // var param!
  lCompare := EditRecord(GetDefaultRecord);
  CompareRecord(lRecord, lCompare);
end;
```

Implementing IDDOPersistent

DDObjects supports using objects as parameters of remote calls thus enabling transfer of more complex structures than provided by using records. This is being done by means of the interface IDDOPersistent which is defined as follows:

```
IDDOPersistent = Interface
  procedure DDOLoadFromStream(Stream: TStream);
  procedure DDOSaveToStream(Stream: TStream);
end;
```

Classes which should be transferred by DDObjects need to implement this interface. Besides of implementing the interface, these classes also need to be registered at DDObjects as shown in this example taken from the unit DDOPersistentClasses.pas:

```
    DDOPersistentRegistry.RegisterClass(TDDOPersistentStrings, CreateStrings);
```

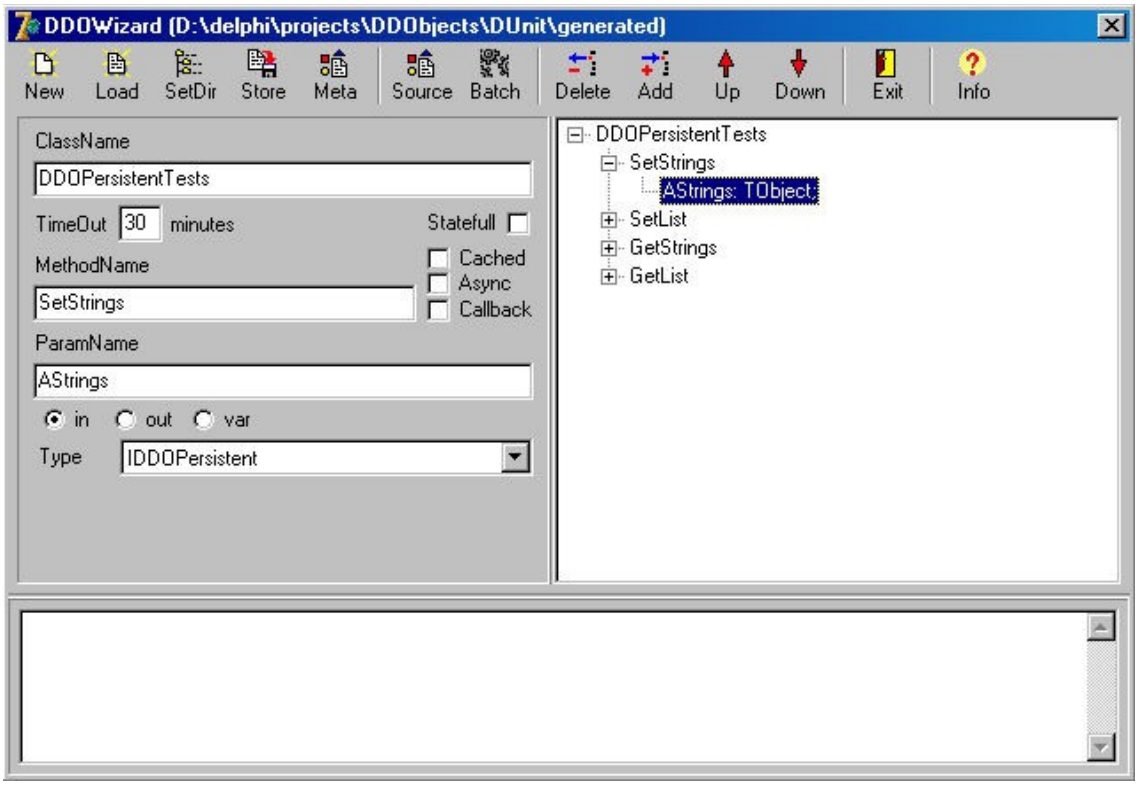
The second parameter is a user supplied function which will be called back by DDObjects if a new instance of the registered class needs to be created. This is not only being necessary as TObject doesn't define a virtual constructor but also because the construction could be more complex (E.g. passing additional parameters to the constructor). As this registration is necessary within the server as well as within the client it might be wise to to put this code in a separate unit to be included in both applications.

For convenience two common classes are provided implementing IDDOPersistent: TDDOPersistentStrings and TDDOPersistentList. To complete this introduction the implementation of IDDOPersistent within the class TDDOPersistentStrings is being shown here (IFDEFs specific to Delphi 5 omitted):

```
procedure TDDOPersistentStrings.DDOLoadFromStream(AStream: TStream);
var
  lSorted, lCaseSensitive: Boolean;
  lDuplicates: TDuplicates;
begin
  AStream.Read(lSorted, SizeOf(Boolean));
  AStream.Read(lDuplicates, SizeOf(TDuplicates));
  AStream.Read(lCaseSensitive, SizeOf(Boolean));
  Sorted := lSorted;
  Duplicates := lDuplicates;
  CaseSensitive := lCaseSensitive;
  LoadFromStream(AStream);
end;

procedure TDDOPersistentStrings.DDOSaveToStream(AStream: TStream);
begin
  AStream.Write(Sorted, SizeOf(Boolean));
  AStream.Write(Duplicates, SizeOf(TDuplicates));
  AStream.Write(CaseSensitive, SizeOf(Boolean));
  SaveToStream(AStream);
end;
```

Now let's take a look at a very simple example which (again) has been taken from the DUnit test cases:

As the interface has already been defined, we'll start the wizard and load the file DDOPersistentTests.xml which can be found in the folder DUnit\generated into the wizard. As shown in the screenshot, the parameter's type is IDDOPersistent.



As usual we execute the source code generation and start creating the client and the server. These steps won't be described within this tutorial again. Please refer to one of the former tutorials if you are unsure about how to use the TDDOListener or TDDORequester.

Let's have a look at the server code. The class TDDOPersistentTests is a descendent of TDDOPersistentTests_Stub which has been generated by the wizard before. The code creates an instance of TDDOPersistentStrings, sets some of it's properties and adds some strings to it.

```
procedure TDDOPersistentTests.GetStrings(out DDOParam0: TObject);
var
  lStrings: TDDOPersistentStrings;
  i: Integer;
begin
  lStrings := FreeAfterProcessing (TDDOPersistentStrings.Create);
  lStrings.Duplicates := dupError;
  lStrings.Sorted := True;
  for i := 0 to 4 do
    lStrings.Add(IntToStr(i));
  DDOParam0 := lStrings;
end;
```

Please note the call of FreeAfterProcessing in the above code. As the created instance has been created locally we do have to destroy it in order to avoid memory leaks. Unfortunately we can't do this here as we assigned it to the out parameter of the method. Therefore DDObjects needs to handle this for us after the call has been processed.

Finally let's have a look at the client code:

```
procedure TDDORemotePersistentTests.TestGetStrings;
var
  lStrings: TDDOPersistentStrings;
begin
  FProxy.GetStrings(TObject(lStrings));
  try
    Assert(lStrings.Duplicates = dupError);
    Assert(lStrings.Sorted = True);
    for i := 0 to 4 do
    Assert(StrToInt(lStrings[i]) = i);
  finally
    lStrings.Free;
  end;
end;
```

Again, there's nothing special in this code. DDObjects handles the complete task of transferring the instance from server to client as well as creating a new instance on the client side. The above code just makes some assertions as it has been taken from the DUnit tests. The call of FProxy.GetStrings might seem to be a bit unsafe as DDObjects currently does not offer type safety using IDDOPersistent as it does when using Records, Sets or Enumerations. However, you can still declare a local variable of type TObject and add a simple assertion as Assert(lTemp is TDDOPersistentStrings) and use a cast afterwards.

Using SSL in DDObjects

Two samples which demonstrate the basic usage of SSL in DDObjects are enclosed within this distribution and can be found in appropriate sub folders in the folder Demo.

EldosIndySSL	A client and server using Indy 9 or 10 with Eldos SecureBlackbox
Synapse SSL	A client and server using Synapse with OpenSSL or Eldos SecureBlackbox. The used SSL implementation can be switched by commenting resp. uncommenting the conditional define {$DEFINE SecureBlackbox} within the include file SynapseSSLDemo.inc.

Remember to select the DDObjects connection components for Indy resp. Synapse as described within the chapter “Changes in Version 1.1” as well as the SecureBlackbox helper package(s) as described in SecureBlackbox readme. Alternatively you may add the appropriate path to SecureBlackbox to your default or projects search path.

To use the demos for SecureBlackbox you need to open the file EldosLicense.inc contained within the folder CertLic and insert the key you have received from Eldos.

The samples also show a very basic implementation of a helper class (TSSLHelper) with concrete descendants for Eldos and Synapse which could be used to keep the application code clean from any code specific to the used socket library or SSL implementation.

Command Line Tools

The package contains a set of command line tools which extend some of the functions found in the wizard. As the wizard won't be available in the personal editions of Delphi 2005 and 2006 you also can invoke it using these command line tools. The tools can be executed via DDOTools.exe on the command line. The following parameters are available:

-gui	Invokes the DDObjects wizard
-l[ist] ip:port	Displays available services on specified host
-**d[isplay] ip:port name**	Displays definition of specified service
-c[ode] ip:port name	Generates code to access specified service

The following example will connect to a DDOListener running on port 7000 on the host with the IP 192.168.1.99, import the meta data which defines the available methods of the service called TestClass. Afterwards code-generation will be triggered in order to generate source files to be included within your project to access and invoke the service TestClass using a DDORequester component.

DDOTools -c 192.168.1.99:7000 TestClass

Known Issues

Currently callbacks are not working in conjunction with the windows message based transport layer. For a most up to date list please locate your browser to

http://www.ddobjects.de

Limitations of the Trial (unregistered) Version

Apart from the fact that the source code is not included there are no more limitations except the number of calls to be processed by all listeners within a certain server process. As soon as 10,000 requests have been handled, all DDOListeners will be deactivated. The server needs to be restarted in order to handle new requests.

The trial version further does not allow usage of other than the Borland Socket Components, therefore lacks support for SSL, and does not contain the preview for Free Pascal and no support for Borland Kylix.

Frequently Asked Questions

About DDObjects

What's the difference between remote calls, callbacks, asynchronous calls and asynchronous callbacks?
Remote calls are blocking calls initiated by the client. The client waits until it receives the result by the server. Asynchronous calls are also initiated by the client. In contrast to remote calls the client won't wait but an appropriate event will be triggered within the client. Callbacks are blocking calls initiated by the server which offers a subscribe/unsubscribe mechanism to clients. Asynchronous callbacks are almost identical to callbacks but won't wait for the client. Instead, as it's the case with asynchronous calls, an appropriate event will be triggered as soon as the server receives the client's result.

What are stateful and stateless objects?
Stateful objects are associated to a client. The first time a client invokes a method of an object, a new instance will be created. Following calls will be handled by this object; it's exclusively used by that specific client. As a result of that, the object can track these calls and keep relevant information in it's private fields; it does have a "state". In contrast to this, stateless objects are created and destroyed on demand; neither the number of objects can be predicted nor is there any way to know which instance will execute the clients request.

Installation

I can't install the components into Delphi 2005 Personal Edition.
Delphi 2005 Personal Edition does not have the xmlrtl.dcp file which is required by the DesignIde package. This means that users of that edition won't be able to compile any design time packages which require it. The solution is to install a faked xmlrtl.dcp like xmlrtlFAKE.dcp which contains the two faked units XmlDom and XmlIntf. These two units do not have an implementation section; their interface sections contain only the interface section parts that are needed by the DesignIde package. With this .dcp file it is possible to use design time packages in Delphi 2005 Personal Edition. You can download the file at:

http://unvclx.sf.net/other/D2k5PExmlrtlFake.zip

However, within Delphi 2005 Personal Edition the DDOWizard will not be available. Users of this version can use the command line tools instead which are available since release 0.9.50.

While building the packages with Delphi 5 I am getting an error L1496.
This is an internal error caused by the Delphi linker which is frequently encountered and not specific to DDObjects. While doing a complete build will not always succeed, the best is to delete all DCU-Files before doing so.

While building the packages Delphi complains about a missing entry point.
Probably you already have an older version of the package installed. You can ignore the error but don't forget to re-install the components afterwards.

When I try to install the components Delphi notifies me that the package DDObjects.bpl can not be found.
The design time package tries to load the package DDObjects.bpl but fails to do so. Check your search path and make sure, the package in question can be found: either put it into the same folder (which is Delphi's default-setting) or make sure your search path includes the folder where the package is located.

While building the packages or demos with Delphi 5, the compiler tells me the unit Variants.pas is missing.
I have forgotten to exclude the unit by using the appropriate IFDEF clause. With the advent of Delphi 6 several routines which deal with variants have been moved from unit System.pas, which is automatically used by any unit, to Variants.pas. You can safely remove the unit from the uses-clauses and recompile.

Programming with DDObjects

All data is being sent over the wire in plain text. Is there any way to encrypt the data?
Yes, there is. Although DDObjects doesn't contain any algorithms for encryption/decryption on it's own, the DDORequester as well as the DDOListener both offer a suitable event: OnDataEvent. You can attach an event handler to it, which receives the XML which is about to be transferred or has been received and encrypt/decrypt it. Be aware that, in order to not block other threads and events, these events are not being synchronized with the main thread.
Since version 1.1 you can take advantage of the exchangeable connection layer which supports Synapse and Indy 9 and 10. These libraries offer pluggable support for SSL.
Is there a way to compress the data?
Yes, there is. Check the answer to the question above ("data being send in plain text"). The same event can be used to compress and decompress the data. For this purpose the unit DDOBase.pas provides two functions - Compress and Uncompress - using zlib 1.2.3 which is included within DDObjects.

Can I develop using DDObjects on a Windows 98SE PC?

There's one critical API call which won't succeed in Windows 98 - TryEnterCriticalSection (it's only an empty stub on that OS). To make a long story short: you can develop using DDObjects on Windows 98SE but running server applications on that OS is not advisable! You can work around this issue by setting the property ClientAccess of TDDOListener to caSerialized. Using this value all client calls will be queued and serialized. This will reduce the performance significantly so this setting should be avoided in time-critical systems.

General Questions

I have encountered a bug.

Please send me a description of the problem and, if possible, some source code which provides it. I'll let you know whether I have been able to reproduce it and try to fix it within the next build. In case it's a basic functionality failure, the bug will be published on the website and handled with high priority so you can track the state of the bug.

I do have a suggestion.

Great. Please let me know about it and share your thoughts. If it does fit within the concept I'll try to include it within one of the next builds.

Final Notes

Performance, Delphi and it's MemoryManager

As Delphi's default MemoryManager is known to scale bad in multithreaded environments and suffers from memory fragmentation I recommend using FastMM as MemoryManager replacement at least for server applications build with DDObjects. Note that Borland has replaced Delphi's MemoryManager by FastMM since Delphi 2005.

FastMM can be downloaded at SourceForge.net:

http://sourceforge.net/projects/fastmm/

Currently nothing else remains to say but

Enjoy! ;-)

Software License

As a proprietary product of Stefan Meisner, DDObjects is protected by copyright laws. At all times Stefan Meisner retains full title to the software. Subject to your acceptance of and accordance with the terms and conditions stated in this agreement, you shall be granted a single-user software license. Any previous agreement with Stefan Meisner is superseded by this agreement.

THIS SOFTWARE LICENSE GIVES YOU THE RIGHT TO

- Install and use the software for the sole purposes of designing, developing, testing, and deploying application programs which you create. You may install a copy of the software on two computers and freely move the software from one computer to another, provided that you are the only individual using the software.
- Write and compile your own application programs using the software.
- Make one copy of the software for backup or archival purposes or copy the software to a single permanent storage medium provided you keep the original solely for backup or archival purposes.
- Distribute the DDObjects runtime packages for the sole purpose of executing application programs created with Delphi.

ENGAGING IN ANY OF THE ACTIVITIES LISTED BELOW WILL TERMINATE THE SOFTWARE LICENSE. IN ADDITION TO SOFTWARE LICENSE TERMINATION, STEFAN MEISNER MAY PURSUE CRIMINAL, CIVIL, OR ANY OTHER AVAILABLE REMEDIES

- Distribution of any files contained in this software package, other than the runtime package explicitly listed above, including but not limited to .pas, .dfm, .dcu files, .dcp files, and design-time packages.
- Removal of proprietary notices, labels or marks from the software or documentation.
- Creation of an application that does not differ materially from the software.
- Creation of an application (whether it will be freeware, shareware or a commercial product) which competes directly or indirectly with DDObjects.
- Distribution of an application program created using the software to another developer. A developer is defined as any person who is executing an application program created using the software, on a computer which contains an installation of Delphi. In order to execute such an application, the developer must own a license to the software and must have installed the software on the computer.
- You may not use the software to create components to be used by other developers.
-

USE OF SOURCE CODE

Recipient will not utilize the source for the creation of software (whether it is freeware, shareware or a commercial product) which competes directly or indirectly with DDObjects. In addition, recipient will not disclose the source itself, nor the implementations discovered therein, to any party.

DISTRIBUTION OF SOURCE CODE

Recipient will not distribute the source. Specifically this includes all .dcu, .dfm, and .pas files which Stefan Meisner has provided.

TECHNICAL SUPPORT FOR SOURCE CODE

Stefan Meisner will not provide support for changes recipient makes to the source. Recipient assumes full responsibility for supporting any code or application which results from such modification. Recipient will not hold Stefan Meisner liable, directly or indirectly, for any changes made to the source, including changes which recipient has made based on advice or suggestions provided by Stefan Meisner.

SOURCE IS PROVIDED AS IS

Stefan Meisner makes no warranties, express or implied, with respect to the source and hereby expressly disclaims any and all implied warranties of merchantability and fitness for a particular purpose. In no event shall Stefan Meisner be liable for any direct, indirect, special, or consequential damages in connection with or arising out of the performance or use of any portion of the source.

LIMITED WARRANTY

Except as specifically stated in this agreement, the software and software documentation is provided and licensed "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

In no event will Stefan Meisner be liable to you for any damages, including without limitation lost profits or revenues, loss of data, business interruption loss, recovery or substitution costs, or claims by third parties, or other indirect incidental or consequential damages, arising out of the use or inability to use the software, even if Stefan Meisner has been advised of the possibility of such

damages. In no case shall Stefan Meisner' liability exceed the amount of the license fee paid by you for the software.