# DDObjects

Version 1.0.2



www.delphi-online.at

www.delphi-online.at

stefan.meisner@gmx.net

# Table of Contents


Introduction ..... 4
Installation ..... 5
System Requirements ..... 5
Shareware Version without Sources ..... 5
Registered Version incl. Sources ..... 5
Tutorials and Implementation ..... 6
First Steps ..... 8
Implementing Callbacks ..... 13
Callbacks and Asynchronous Callbacks ..... 15
Asynchronous Calls ..... 16
Type Rich Exception Handling ..... 18
Using Sessions ..... 19
Records, Sets and Enumerations ..... 20
Implementing IDDOPersistent ..... 21
Events, Properties and API ..... 24
TDDOListener ..... 24
TDDORequester ..... 26
TDDODataGrid ..... 28
TDDOTimer ..... 29
TDDOThreadStringList ..... 29
Miscellaneous ..... 29
Command Line Tools ..... 30
Common Persistent Classes ..... 32
TDDOPersistentStrings ..... 32
TDDOPersistentList ..... 32
Known Issues ..... 33
Limitations of the Shareware (unregistered) Version ..... 33
DDObjects fails to compile in C++ Builder 2006 ..... 33
Frequently Asked Questions ..... 34
About DDObjects ..... 34
Installation ..... 34
Programming with DDObjects ..... 35
General Questions ..... 36
History of Releases ..... 37
Final Notes ..... 40

## *Introduction*

DDObjects is a remoting framework to be used with Borland Delphi and C++ Builder which originally has been started out of personal interest for technologies like DCOM, RMI, Corba etc. A main goal while developing DDObjects has not been only to keep the code one has to implement in order to utilize DDObjects as simple as possible but also very close to Delphis 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 to be used in an internal 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 spending more than 2 years in my drawer I started with some first extensions which have been published in February 2006. After being involved into a larger .NET 2.0 project, I continued the development: Version 1.0 saw the light of the day on the 6th of march 2007.

### *DDObjects supports*

- Remote method calls
- Server callbacks
- Asynchronous calls
- Asynchronous callbacks
- Stateful and -less objects
- Client sessions
- Sets and Enumerations
- RecordTypes
- Compression

### *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.

# Installation

## *System Requirements*

DDObjects supports Delphi 5 to 7, Delphi 2005 and 2006 as well as C++ Builder 6 and 2006. DDObjects is running on Windows NT 4, Windows 2000, Windows XP and above. Windows 98 is only partially supported (see FAQ for more details).

## *Shareware 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 and C++ Builder. 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 delphi-online.at, containing seven 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 which contains the package files.*

## *Registered Version incl. Sources*

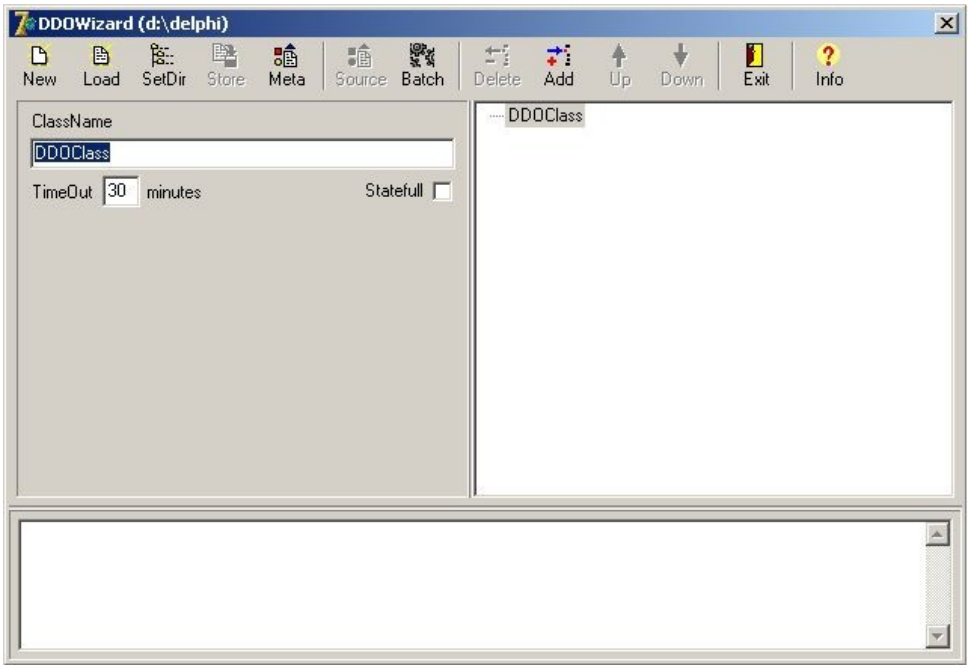
The steps to install DDObjects are the same as described above. If you would like to build DDObjects yourself you'll find the complete source code within the sub folder Source. The sub folder packages contains the package files. Users who like to use DDObjects support for TDataSet should open the file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

*Users of C++ Builder 2006 should read “DDObjects fails to compile in C++ Builder 2006” within the chapter “Known Issues” before building and installing the packages.*

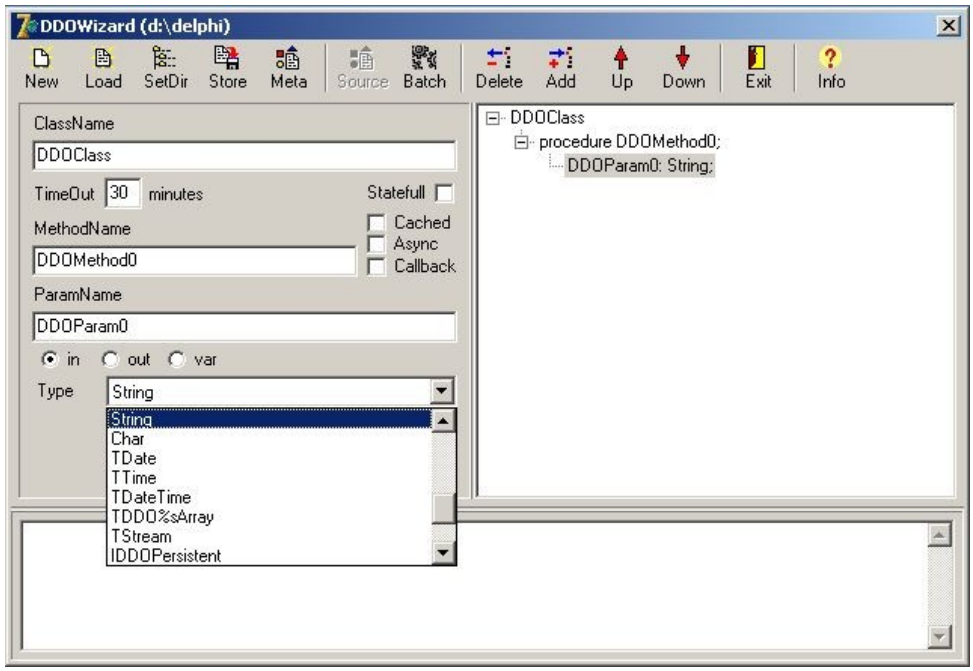
# Tutorials and Implementation

### *DDO Wizard*

Before starting with the tutorials a short introduction to the wizard, which will be your primary tool when developing with DDObjects should be given. To start the wizard select File|New|Other, switch to the tab labelled delphi-online.at 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 on in this screenshot:

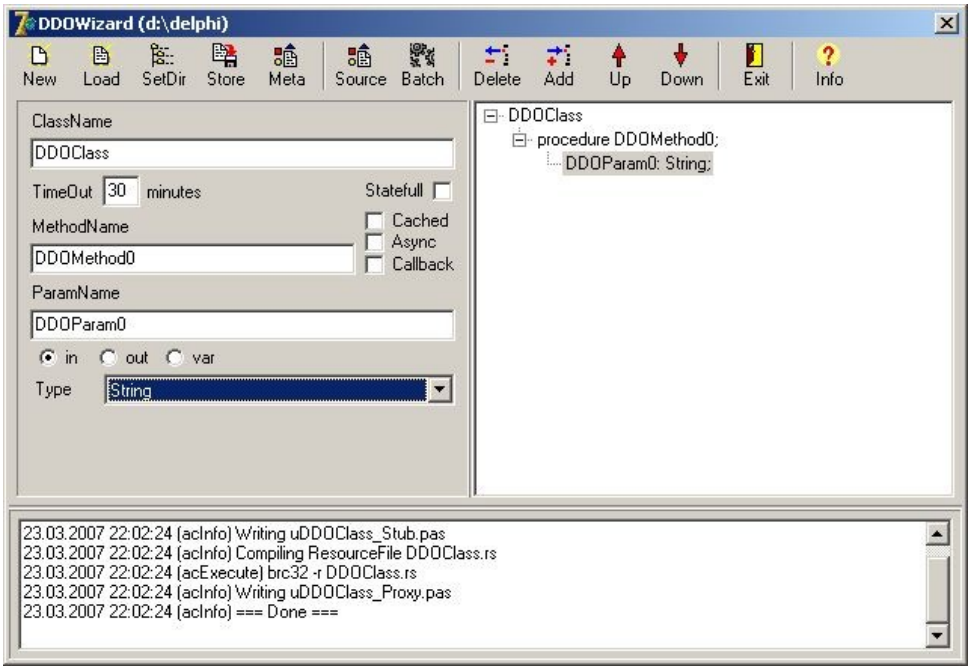


You can enter the name of the method or parameter, select whether the parameter is an in, out or var parameter and select its type. Some types as e.g. arrays and records need further refinement. Additional fields to enter that information will be shown as needed. Nodes can be deleted, moved up and down 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 e.g. 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 labelled Source. Although source code generation usually should be done within a moment the wizard shows each step it does execute in the lower memo as shown in the this screenshot:




This does finish this short introduction to the wizard. Most of the settings which do 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.meisner@gmx.net

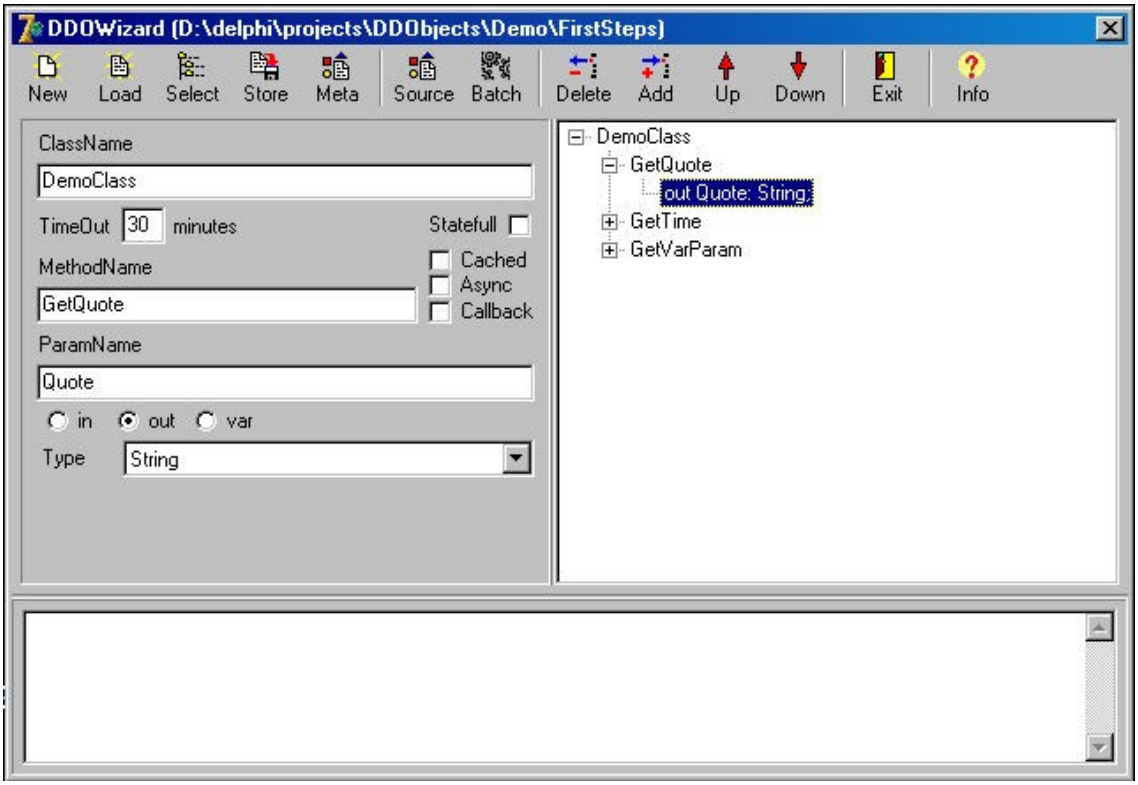
## *First Steps*

### *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 'delphi-online.at' and double click on the contained icon (DDOWizard) to run the integrated wizard. Within that wizard, provide an appropriate name to be used for the class name (we'll assume the name 'DemoClass' as an example) and add these methods along with their parameters and types as shown in the following screen shot.




Leave the checkboxes labelled '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 should be stored.

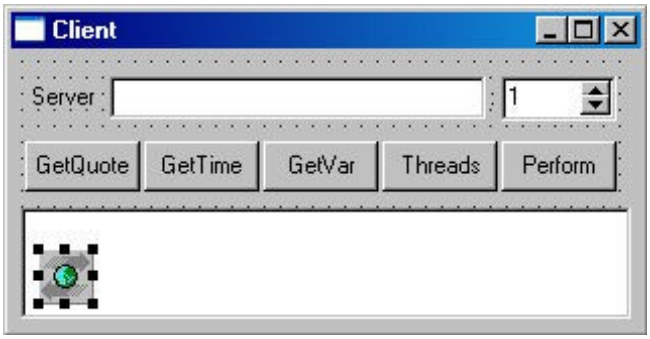
### *Generating Sources*

Click on the button labelled '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 whose names will depend 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 'delphi-online.at' within 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 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 labelled 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 within a single line of code:

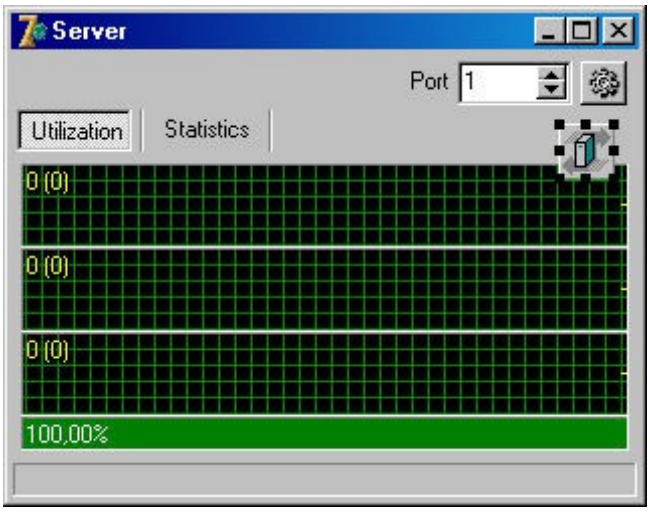
```
procedure TClientMainForm.ButtonGetQuoteClick(Sender: TObject);
var
  AQuote: String;
begin
  FDemoClass.GetQuote(AQuote);
  MemoResult.Text := AQuote;
end;
```

### *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 'delphi-online.at' 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 do 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.

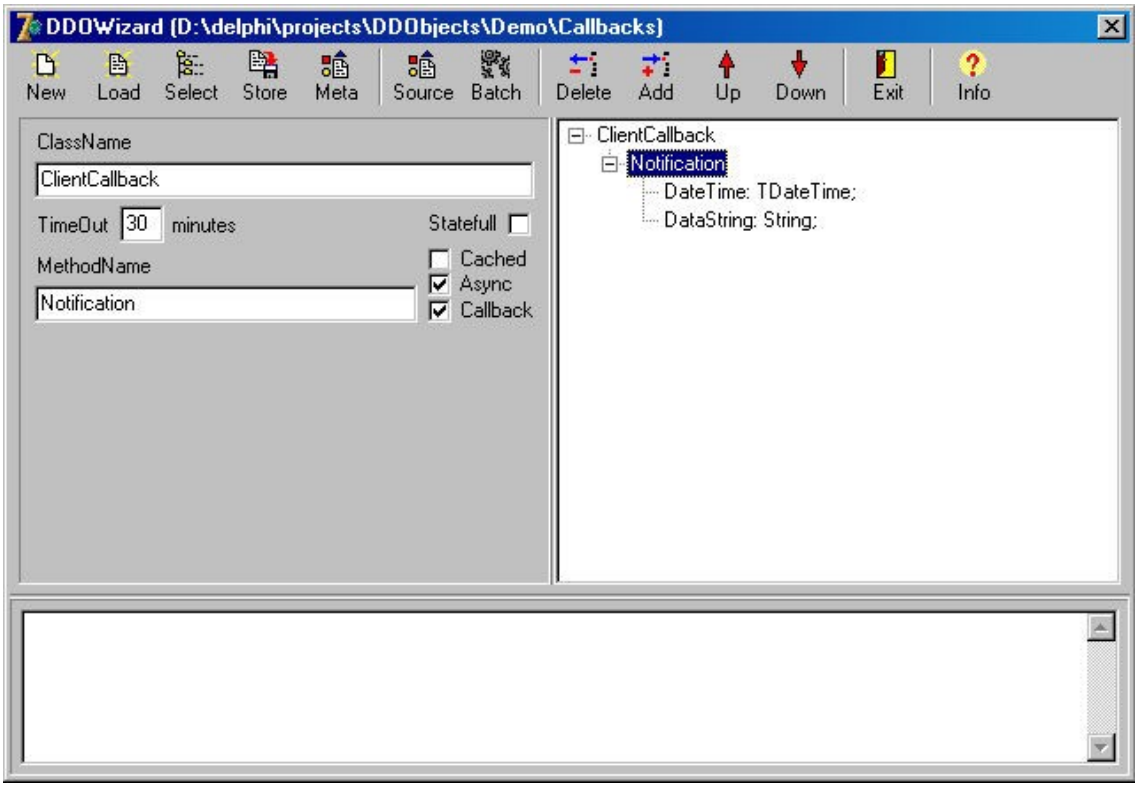
## *Implementing Callbacks*

### *Defining the Interface*

Again, we do start by defining the interface. Select File|New|Other, select the tabsheet labelled 'delphi-online.at' and doubleclick the contained icon (DDOWizard) to run the integrated wizard. The name of the class we define should 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 the class is tagged to be 'Stateful' and we have selected the procedure Notification being an 'Async Callback' (the difference between callbacks and async callbacks will be described at the end of this chapter)method. Stateful does mean, that this object is 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 do 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 do start building the client first. Create a new application and construct the main form's GUI. Select the tabsheet labelled 'delphi-online.at' within the component-palette and drop a DDORequester on your form. Add the unit uClientSession_Proxy to the project and use it within the form's code. Note that generated units, which need to be added to the client, always do 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;
```

which is an usual event handler. The procedural type can be located within the unit uClientSessionCallbackListener_Stub.pas and does reflect the parameters we did define within the wizard. Any time a callback is received by the server, the 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 which does reflect the 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 an value of 0 the DDOListener component will select an appropriate one of 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;
```

You should be aware that the callback will not be executed within the MainThread, so you should care for synchronization. Although it's not recommended to access VCL components from within an another than the MainThread the above is safe as setting a Memo's text internally does

use 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 'delphi-online.at' from the component palette and drop a TDDOListener as well as a TTimer onto the form.

As within the first tutorial do 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 within the first tutorial.

We still do need to implement the notifications. The listener offers a method called EnumStubs which does take 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*

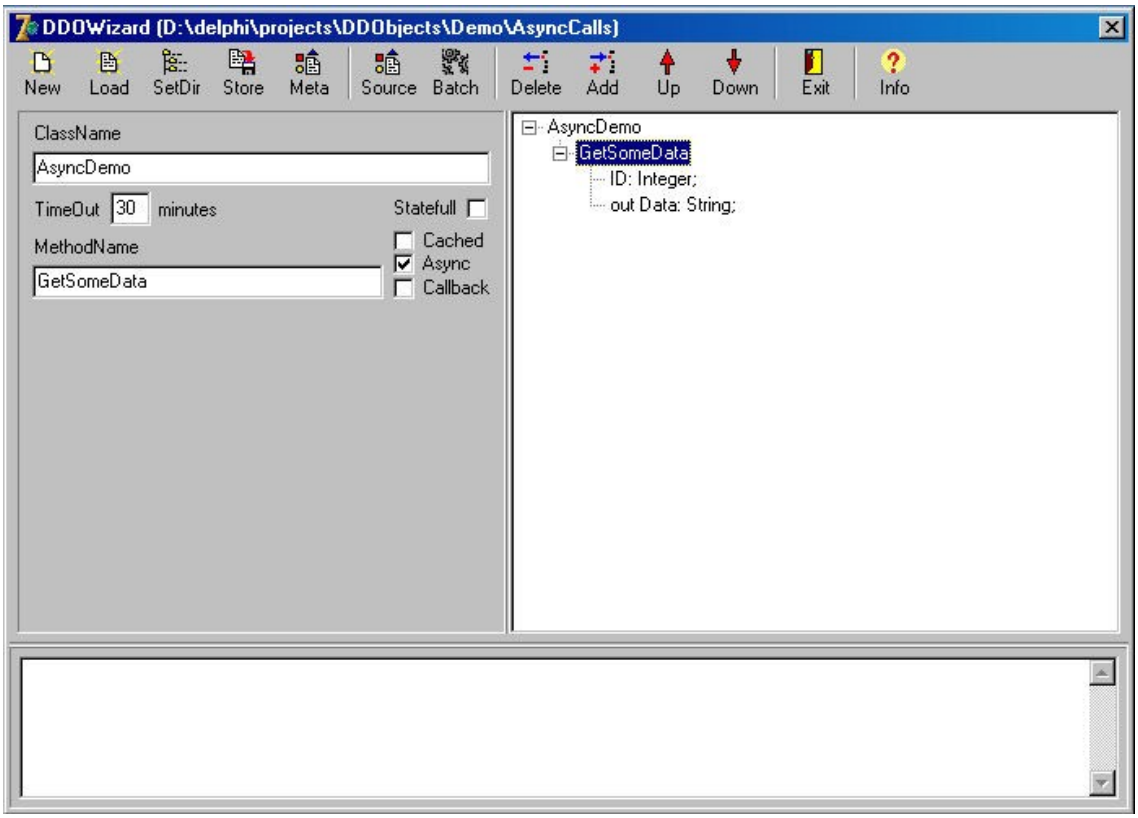
The difference between Callbacks and Asynchronous Callbacks is that the server will be blocked when executing Callbacks waiting for the client's result. As described within the next tutorial, Asynchronous Callbacks will be executed by another thread which will trigger a notification if the client's result has been received. The notification is optional and can be omitted. If you don't have special reasons 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 as the result of the call will not be available immediately but at a later time. As 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.

The class of this simple example contains only one method which is tagged as being Async. Actually, this is anything you need to do within the wizard before executing code generation. The screenshot does show 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 within the client. First we'll do 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 incorporates the parameters which represent the result returned by the server as well as an additional parameter named GUID. The proxy itself does contain the method GetSomeData as we've defined it within 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 an method pointer which accords to 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.

You should 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 Rich Exception Handling*

In contrast to other remoting systems which do not support a type rich 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
- EAbstractError
- EOSError
- EInvalidCast
- EListError
- EStringListError
- EInvalidOperation

To register custom exceptions invoke DDOExceptionRegistry.RegisterException contained within 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. E.g. 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 embrace this code within a common 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;
```

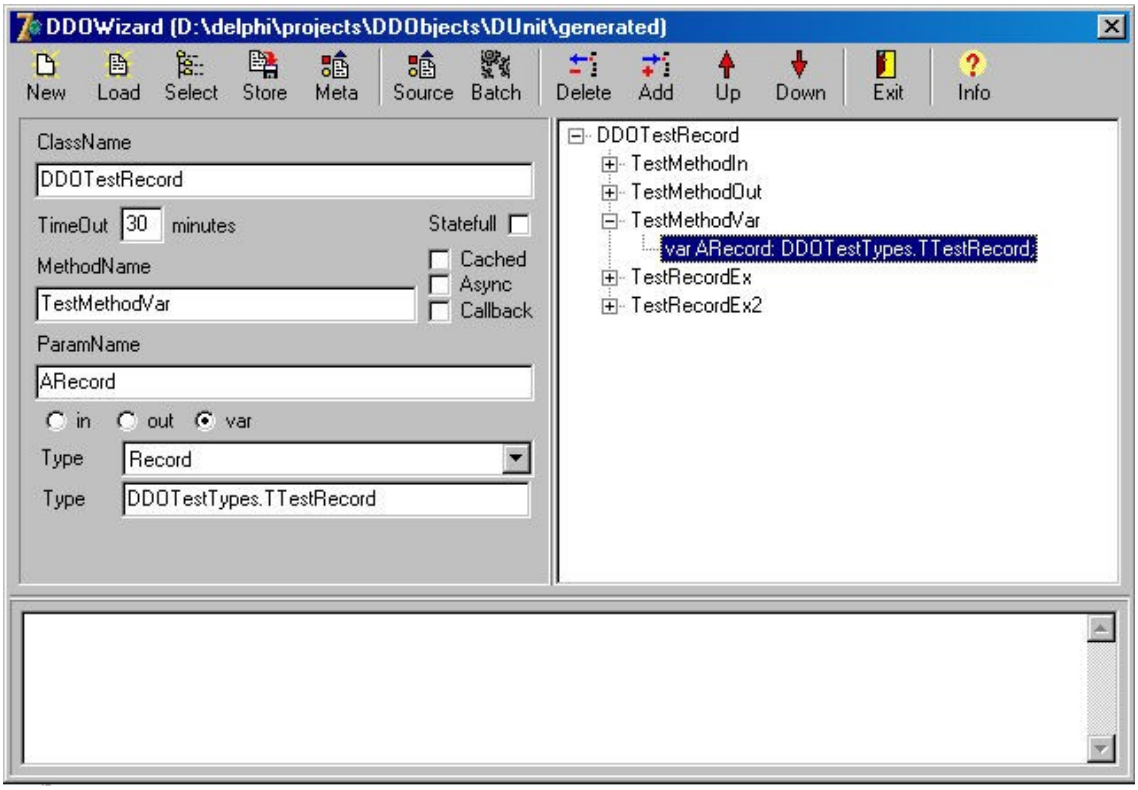
A short code snippet - taken from the demo project - should illustrate the above said. Please note that the session values are set within the class TDDOSessionLogin and being accessed and used within the class TDDOSessionTest.

```
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. DDObjects version 1.1 will enable session persistence so invalidated sessions can be resumed at a later time.

## *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 within the next screen shot:




The unit will be used by the units generated by the DDOWizard within the client- as well as within the server code. Therefore it might be wise to embrace the type definitions within a common 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 facilitated 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);
```

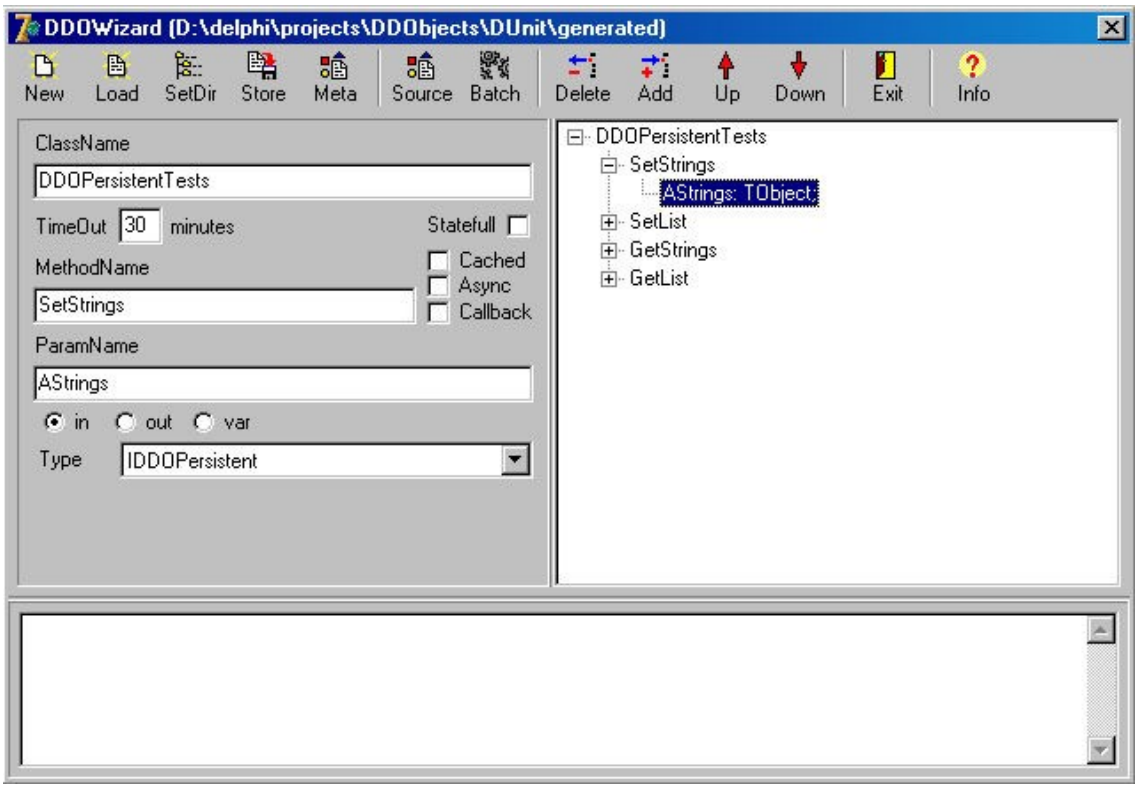
The second parameter is a user supplied function which will be called back by DDObjects if a new instance of the registered class needs to be created. This is not only being necessary as TObject doesn't define a virtual constructor but also as the construction could be more complex e.g. passing additional parameters to the constructor. As this registration is necessary within the server as well as within the client it might be wise to to embrace this code within a common 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 within the folder DUnit\generated into the wizard. As shown within the screenshot, the parameter's type is IDDOPersistent.




As usual we do 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 within 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 did assign 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 using a cast afterwards.

*This issue might be addressed in version 1.1 but would need an upgrade of your defined interfaces as well as complete code regeneration so currently I'm not sure if this would be worth the effort.*

# Events, Properties and API

This section describes the public properties, event handlers and methods of the most commonly used components as well as some utility functions which are part of DDObjects. I would also like to point out to the DUnit tests which accompany this release. The DUnit tests cover various aspects of DDObjects and can be seen as part of the documentation ;-)

## *TDDOListener*

A non visual component to be used within the server application. A TDDOListener listens for new and handles client connections, cares for the creation and life time management of stubs, threads and sessions and handles client requests to be executed by concrete stub instances.

### *Methods*

**EnumStubs**

Enumerates all instances of a given stub class and calls a user supplied callback for each active instance. This method is being used in conjunction with callbacks (see tutorial).

**RegisterClass**

Registers the given class at the listener.
After registering, remote calls can be executed.

**UnregisterClass**

Unregisters the given class at the listener.

### *Properties*

**Active: Boolean**

Determines the state of the listener.
Setting this value to true will activate the listener.

**ActiveConnections: Integer**

The number of currently active connections (see property Connections)

**ActiveStubs: Integer**

The number of currently active stubs.

**ActiveThreads: Integer**

The number of currently active threads.

**CheckVersion: Boolean**

Compares the version of the proxy which executed a call against the version of the appropriate stub class before processing the request.

**ClassCount: Integer**

The number of registered classes (see .property StubClass).

**ClientAccess: TClientAccess**

Determines whether client requests will be serialized (caSerialized) or multi threaded (caMultiThreaded). A value of caSerialized will queue all client requests, processing one after the other. Setting this value reduces the overall performance and should be avoided.

**Connections[Index: Integer]: TCustomWinSocket**

Retrieves the TCustomWinSocket associated with the connection at the given index (see property ActiveConnections).
*This property might become obsolete/protected.*

**IdleThreads: Integer**

Returns the number of client threads which are currently idle.

**KeepAlive: Boolean**

Determines whether the listener will notify the client during processing of a request. Using this method the client's TimeOut can be extended. This might be used if execution time is not predictable or is subject to large changes. See also property TDDORequester.TimeOut.

**MaxThreads: Integer**

Returns the maximum number of client threads which have been active.

**Port: Integer**

Gets/sets the port the Listener is being binded to.

**Requests: Integer**

Returns the number of requests the listener has processed.

**SessionTimeOut: Byte**

Gets/sets the number of minutes after an inactive session will be discarded.

**StubClass[Index: Integer]: TDDOStubClass**

Retrieves the TDDOStubClass at the given index (see property ClassCount).
*This property might become obsolete/protected.*

**ThreadCacheSize: Integer**

Gets/sets the number of client threads which will be cached for later reuse.

**ThreadTimeOut: Byte**

Gets/sets the number of minutes after an inactive client thread will be terminated.

**UseSSL: Boolean**

Determines whether SSL will be used. This property is currently not used.

### *Events*

**OnActivate** procedure(Sender: TObject) of object;

Fired immediately after the listener has become active.

**OnClientException** procedure(Sender: TObject; E: Exception; const Method: String) of object;

Fired if an exception has been raised during processing of the given method.

**OnDataEvent** procedure(Sender: TObject; Sending: Boolean; var Data: string) of object;

Fired if a request has been received or the result will be sent. The parameter Sending determines whether the data has been retrieved (Sending = False) or is about to be sent (Sending = True). This event handler might be used for encryption or compression.

**OnDeactivate** procedure(Sender: TObject) of object;

Fired immediately before the listener gets deactivated.

**OnRequestProcessed** procedure(Sender: TObject; Result: TDDODocument) of object;

Fired after a request has been processed.

**OnRequestReceived** procedure(Sender: TObject; Request: TDDODocument) of object;

Fired if a request has been received.

**OnStubCreated** procedure(Sender: TObject; Stub: TDDOStubBase) of object;

Fired if a new stub has been created.
This event handler might be used to initialize the stub.

## *TDDORequester*

A non visual component to be used within the client application. A TDDORequester initiates and handles the communication with the remote server, serializes and transforms requests to be sent to the server and unserializes and transforms the results as they are being received.

### *Methods*

#### Lookup

If connected to a TDDONameServer, this function will lookup the location (remote host and port) of a given class and returns a new proxy which will connect and send it's requests to the registered host.

### *Properties*

#### Active: Boolean

Determines the state of the requester.

#### Cursor: TCursor

The cursor to use during execution of synchronous calls.

#### Host: String

The name or IP of the remote host to connect to.

#### Port: Integer

The port of the remote host to connect to.

#### Requests: Integer

The number of requests which have been sent.

#### Synchronized: Boolean

Setting this property to true will force callbacks and responses of asynchronous requests to be executed within the context of the application's main thread.

#### TimeOut: Integer

Determines the number of milliseconds the client will wait to receive the result of a request. If the result is not being received within that amount of time, the request will be aborted.

#### UseSSL: Boolean

Determines whether SSL will be used. This property is currently not used.

### *Events*

**OnActivate** procedure(Sender: TObject) of object;

Fired immediately after the requester has become active.

**OnAsyncFailed** procedure(Sender: TObject; const GUID: String; E: Exception) of object;

Fired if an asynchronous request failed. Please refer to the tutorial for further details.

**OnDataEvent** procedure(Sender: TObject; Sending: Boolean; var Data: string) of object;

Fired if a request has been received or the result will be sent. The parameter Sending determines whether the data has been retrieved (Sending = False) or is about to be sent (Sending = True). This event handler might be used for encryption or compression.

**OnDeactivate** procedure(Sender: TObject) of object;

Fired immediately before the requester gets deactivated.

**OnAfterDeactivate** procedure(Sender: TObject) of object;

Fired after the requester has been deactivated.

**OnException** procedure(Sender: TObject; e: Exception) of object;

Fired if an exception occurred while sending a request and waiting for the result.

**OnExceptionReceived** procedure(Sender: TObject; e: Exception) of object;

Fired if the server did send an exception as result of a request.

**OnReceiveData** procedure(Sender: TObject; Size, Expected: Integer) of object;

Fired during receiving the result of a request. Depending on the size of the returned result, the event might be triggered multiple times.

**OnResultReceived** procedure(Sender: TObject; Result: TDDODocument) of object;

Fired after a request has been processed and the result has been received.

**OnSendRequest** procedure(Sender: TObject; Request: TDDODocument) of object;

Fired immediately before a request will be sent.

## *TDDODataGrid*

A visual component which mimics the style of the grid as seen within the Task Manager. The DataGrid can display various items which can be configured and labelled accordingly. The drawing of the curve as determined by the items values will adapt itself to fit the DataGrid's height. Various visual aspects of the DataGrid as e.g. the background colour can be configured.

### *Properties*

**Active: Boolean**

Activates the DataGrid. If active, the DataGrid will update itself periodically.

**DataLines: TDDODataLines**

The collection of items which will be displayed within the DataGrid..

**GridPainter: TDDOGridPainter**

Configures various visual aspects of the DataGrid.

**Interval: Integer**

Determines the amount of time, in milliseconds, that passes before the active DataGrid component updates itself and initiates another OnGetData event.

### *Events*

**OnGetData** procedure(Sender: TObject; Index: Integer; var Value: Integer) of object;

Occurs when a specified amount of time, determined by the Interval property, has passed. This event will be called once for each item within the DataLines collection identified by the parameter Index. The user should supply the value of the current item using the parameter Value.

## *TDDOTimer*

A non visual component which acts as a replacement for Delphi's TTimer component. Delphi's TTimer suffers from a shortcoming that could lead to an EOutOfResources if used in multi threaded environments (e.g. enabling/disabling the timer within the context of different threads). The events and properties are the same as of TTimer and therefore not listed here.

## *TDDOThreadStringList*

A descendent of TStringList (contained within the unit DDOSystem.pas) which is thread safe. The methods and properties are the same as of TStringList so they won't be repeated here.

## *Miscellaneous*

Some functions and classes which are of common interest can be found throughout DDObjects. Some of the probably most common ones are listed here:

| | |
|---|---|
| DDOBase | **procedure Compress(var Data: String);**<br>**procedure Uncompress(var Data: String);**<br>Based on zlib 1.2.3, these procedures ease the compression and decompression of a string passed as parameter. |
| DDOZlib | Based on the original file Zlib.pas, which comes with Delphi, this modified unit has been adapted for zlib 1.2.3 |

DDOSystem

**TDDOCriticalSection**

Just another wrapper for TRTLCriticalSection with some additional methods and properties:

**procedure CheckEntered;**

**function TryEnter: Boolean;**

**property OwningThread: THandle**

**property RecursionCount: LongInt**

**function _Hack(AnObject: TObject; Offset: Integer;**
**ExpectedClass: TClass = nil): Pointer;**

This is the Pandora's Box for each Delphi programmer as it allows access to private fields ;-) In order to use the function one needs intimate knowledge about the object which should be accessed.

DDOStringReplace

**function StringReplace_HV_IA32_1**(...

Much faster replacement for Delphi's StringReplace.

DDOFirewall

**TDDOProtocol = (prUDP, prTCP);**

**procedure Open(Port: Integer; Protocol: TDDOProtocol);**

**procedure. Close(...)**

**function. SafeOpen(...): Boolean;**

**function. SafeClose(...): Boolean;**

**function. Allowed(...): Boolean;**

Some utility functions to access the Windows Firewall to open and close specific ports. The procedures Open / Close will raise an exception on failure, while SafeOpen and SafeClose will return whether the call has been succeeded.

## *Command Line Tools*

The package contains a set of command line tools which extend some of the functions found within the wizard. As the wizard won't be available within 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**

# Common Persistent Classes

For convenience two common classes are provided which implement IDDOPersistent and already have been introduced within the tutorial “Implementing IDDOPersistent”:

## *TDDOPersistentStrings*

This class inherits from TStringList and implements IDDOPersistent. No additional methods have been added with the exception of the interface implementation. As the documentation is already provided by Delphi it won't be repeated here. Lookup TStringList to see a list of methods and properties.

## *TDDOPersistentList*

This class inherits from TList and implements IDDOPersistent. No additional methods have been added with the exception of the interface implementation. As the documentation is already provided by Delphi it won't be repeated here. Lookup TList to see a list of methods and properties.

### *Contribute to DDObjects*

Feel free to implement your own classes... what about a TDDOPersistentImage ;-)? If you like to see your class(es) to be added to DDObjects you can send me an eMail with your source code. Of course, your name will be included within the list of contributors.

# Known Issues

For a most frequent list please locate your browser to

http://www.delphi-online.at/projects/ddobjects/bugs.php

## *Limitations of the Shareware (unregistered) Version*

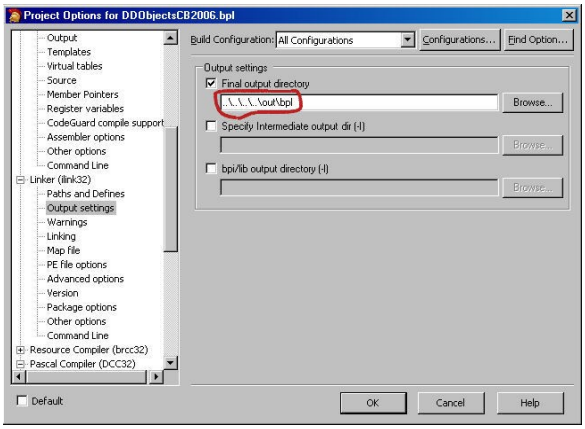
Beneath the fact that the source code is not included there are no more limitations except the number of calls to be processed by all listeners within a certain server process. As soon as 10,000 requests have been handled, all DDOListeners will be deactivated. The server needs to be restarted in order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

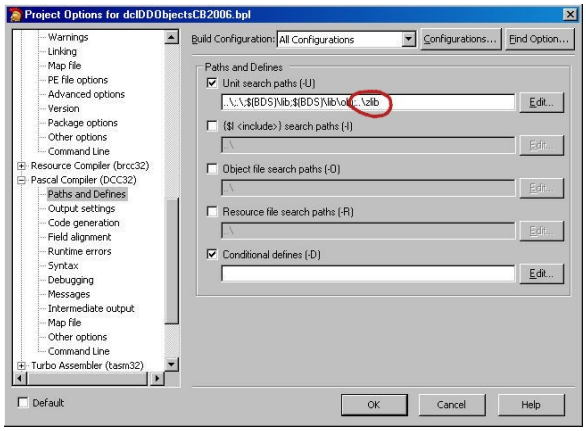
## *DDObjects fails to compile in C++ Builder 2006*

After opening and building DDObjectsCB2006.bpk you want to build the design time package dclDDObjectsCB2006.bpk but receive a message similar to the following one:

```
[Pascal Fatal Error] DDOBase.pas(174): F1026 File not found: 'DDOZLib.dcu'
```

After opening the above packages the IDE might have issued a notice that these packages have been updated to version 10. Unfortunately some of the project options need to be edited manually. At first, you might want to adjust the output path for both packages as shown in the first screen shot. Finally you need to add the folder '..\zlib' to the list “Unit search paths”:




After applying these changes, you can rebuild both packages and install the design time package.

# 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 an 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 within 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 package which does 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 section contains 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 do 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 answer can be found: either do put it into the same folder (which is Delphi's default-setting) or make sure your search path does include the folder where the package resides.

**While building the packages or demos with Delphi 5, the compiler tells me that 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.

**When running the wizard in Delphi 5 it does complain about a property called Designsize.**

Shame on me, I have forgotten to check the DFM-files before releasing the build you are using. Delphi 7 does include them within those files. While that property didn't exist in Delphi 5, the DFM can not be read. You can open the appropriate DFM-file, locate the lines and safely remove them to fix the problem.

## *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 e.g. has been received and encrypt/decrypt it. Be aware that, in order not to block other threads and events, these events are not being synchronized with the main thread.

Please note that OpenSSL will be supported as of version 1.5, therefore encryption and authentication will be an integral part of DDObjects.

**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 exposes 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 production systems.

## *General Questions*

**I have encountered a bug.**

Please send me a description of the problem and, if possible, some source code which does expose 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.

# History of Releases

| | |
|---|---|
| **2007-03-26 (1.0.2)** | Shame on me: I forgot to finish the implementation<br>of a fix in 1.0.1 introducing an Access Violation if<br>using asynchronous calls on a synchronized Requester. |
| **2007-03-24 (1.0.1)** | Fixed a conversion bug if string parameter is empty.<br>WideStrings supported as parameters type.<br>Improved reliability of asynchronous calls<br>(in certain situations - when combining remote<br>calls and asynchronous calls - a deadlock could<br>have been easily introduced in GUI applications)<br>Added an chapter about the usage of the wizard |
| **2007-03-04 (1.0)** | Demo projects consolidated<br>Some typos in documentation fixed<br>Class TProcessTimes renamed<br>Fixed bug where brc32.exe couldn't be found<br>Fixed not compiling unit DDODataSet.pas<br>Added shortcuts to Wizard's GUI |
| **2007-02-12 (0.9.98)** | Burn-In-Test (48 hours) successfully completed:<br>no failures: no Access Violations, no memory leaks.<br>Documentation completed<br>Bug fixed in Wizard's GUI<br>Cleaned up uses clauses<br>Fixed cache which has not been working correctly |
| **2007-01-27 (0.9.97)** | Worked around an Delphi memory leak<br>Fixed an memory leak transferring Exceptions<br>Fixed a very sporadically occurring AV in async. calls<br>New property: TDDORequester.Synchronized |
| **2007-01-24 (0.9.96)** | Fixed an AV within the internal GarbageCollector<br>Fixed an AV if IDDOPersistent param. has been nil<br>Fixed using a wrong const within TDDOListener<br>New event: TDDORequester.OnAfterDeactivate<br>Updated and extended documentation |
| **2007-01-22 (0.9.95)** | Added property TDDORequester.Requests<br>SafeDispose and SafeFreeAndNil removed<br>(never encourage sloppy programming *g*)<br>Added some classes impl. IDDOPersistent<br>TDDOPersistentStrings<br>TDDOPersistentList<br>Additional DUnit tests<br>Documentation extended<br>Some small bugfixes and enhancements |
| **2007-01-18 (0.9.92)** | Bugfix in Wizard storing files |

| | |
|---|---|
| **2007-01-18 (0.9.91)** | Fixed bug in source generator (using records in callbacks)<br>Bug in Wizard (Batch) fixed<br>First Documentation Draft added<br>Registered C++ Builder release:<br>Packages accidental have been build as trial version |
| **2007-01-10 (0.9.90)** | Wizard enhanced<br>Sets and Enumerations supported<br>Type rich exception handling<br>Callbacks enhanced (multiple subscriber on stateless objects)<br>TDDOTimer; corrects a shortcoming of Delphi's TTimer<br>Bugfixes, DUnit tests added |
| **2007-01-05 (0.9.85)** | ReadAttributes fixed<br>Fixed enum values in DDOFirewall<br>Callbacks enhanced using threads<br>Included zlib 1.2.3 |
| **2007-01-03 (0.9.80)** | Native support for RecordTypes<br>Performance considerably increased<br>Integration of functions contained within the command line tools<br>Wizard GUI enhanced, bugs fixed<br>Fixed DDOFirewall (working now)<br>Annoying bug in DDODocument fixed<br>Fixed bug using callbacks |
| **2006-12-18 (0.9.70)** | Property TDDOListener.KeepAlive<br>Packages for each supported version of Delphi and C++ Builder<br>to allow side-by-side install. |
| **2006-12-14 (0.9.66)** | IDDODocumentHandler updated<br>Fixed an AV (property UseSSL)<br>Fixed bug in source generator<br>TDDOConnectDlg added<br>GUI tests added |
| **2006-12-14 (0.9.65)** | Interface IDDODocumentHandler added<br>C++ Builder 5 discarded |
| **2006-12-10 (0.9.60)** | C++ Builder 5 and 6 supported<br>Missing lib files added |
| **2006-12-07 (0.9.51)** | Smaller fixes<br>Package files for C++ Builder added |
| **2006-12-06 (0.9.50)** | Delphi 2006 (Win32) supported<br>Support for C++ Builder 2006<br>Data types Real48 and Comp discarded<br>Installation issue (wizard) fixed<br>Command line tools added<br>New feature added: Sessions<br>DUnit test cases added<br>GUI of DDOWizard enhanced<br>Issue reg. CRLF in string fixed<br>Bug in Demo fixed<br>Several other enhancements |

| | |
|---|---|
| **2006-04-06 (0.9.14)** | Implementation TDDOBroker finished<br>AVs within async. callbacks fixed<br>Sender for async. callback events added<br>Several minor enhancements |
| **2006-02-27 (0.9.13)** | Delphi 2005 supported<br>New component: TDDONameServer |
| **2006-02-27 (0.9.12)** | Performance considerably increased<br>Memory leak in TDDOStubBase fixed<br>Server statistics reworked and integrated within demos<br>Some small bugfixes and enhancements |
| **2006-02-25 (0.9.11)** | Asynchronous callbacks supported<br>Issue reg. growing handles and in adequate memory consumption fixed<br>Bug in source code generator for async. calls without parameters fixed<br>Events of async. calls are now synchronized with the main thread<br>Renamed to DDObjects; classes, types unit names etc. changed accordingly<br>Some minor enhancements |
| **2006-02-22 (0.9.10)** | TRemBroker extended<br>New interface IRemPersistent<br>Asynchronous calls supported<br>New component TRemProgressBar<br>Multi threading enhanced |
| **2006-02-20 (0.9.09)** | New component TRemDataGrid<br>GUI of demos enhanced |
| **2006-02-19 (0.9.08)** | New component and demo: TRemBroker<br>Severe bug in RemSrcGen.pas fixed<br>Subscribe/Unsubscribe for callbacks |
| **2006-02-17 (0.9.07)** | Removed modified unit ScktComp.pas |
| **2006-02-17 (0.9.06)** | Batch compile (Wizard extended)<br>Delphi 5 supported |
| **2006-02-16 (0.9.05)** | Callbacks have been added |
| **2006-02-13 (0.9.00)** | First public release |

# 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! ;-)