

The Developers Magazine

*published for members every two months by The Developers Group
incorporating the DotNET Developers Group and UK Borland User Group*

May/June 2004

Contents

Meetings	2
Tips from the Support Team	2
Bits and pieces	3
Some useful information and sources, and more	
Change Imperative	5
Craig Murphy looks at alternatives to Outlook Express	
Learning To Adapt	7
Andy Denton uses the Adapter Pattern to convert interfaces	
.NET Remoting and DataSets.....	11
Bob Swart builds Distributed Database Apps with Delphi 8	
Essential Delphi 8 for .NET	17
E-book by Marco Cantù	
Some other e-books.....	18
Delphi for .NET Developer's Guide	19
A sample chapter from Xavier Pacheco's forthcoming book	
Serving Up RSS using Delphi 8 (part 2).....	31
Craig Murphy looks at dynamic websites	

The Developers Group (DDG & UK-BUG), run by Joanna Pooley and friends,
is a division of Richplum Ltd, 7 Devizes Road, Upavon, Wiltshire SN9 6ED, U.K.
telephone 01980 630032 fax 01980 630602 e-mail joanna@richplum.co.uk
websites www.dotnetuk.com and www.richplum.co.uk

Meetings

Meetings in 2004:

- Combine material formerly covered separately by the Borland User Group (mainly Delphi, both for .NET and Win32) and the DotNET Developers Group (C# etc).
- Are alternately at
 - POSK in Hammersmith on Tuesdays (with tea/coffee and biscuits, plus two-course supper). POSK is at 238-246 King Street, Hammersmith, London W6 0RF (Ravenscourt Park on the District line - 2 mins walk, Hammersmith on the Piccadilly, District and Hammersmith & City lines - 12 mins walk). Meter parking is available in side streets and in the King Street Mall at around £8.
 - Microsoft's offices in Reading on Thursdays (substantial refreshments and a buffet meal). We meet in Building 3, Microsoft Campus, Thames Valley Park, Reading RG6 1WG. Parking is available and there are frequent courtesy buses from/to central Reading and the station.
- Take place as regularly as possible but avoid weeks shortened by bank holidays.
- Take place (more or less) monthly except in August and December.
- Are free to members **provided that places are booked at least two days in advance** (even if only provisionally).
- Comprise technical sessions presented (mainly) by members - the main purpose of the user group being to facilitate the interchange of knowledge between members.

Meeting dates (agendas to be advised):

- June 17th at Microsoft
- July 13th at POSK
- August - no meeting
- September 23rd at Microsoft
- October 19th at POSK
- November 18th at Microsoft
- December - no meeting

Masterclasses are also being planned for the autumn.

Tips from the Support Team

Trigger Order in SQL Server

The following command is used to control the order in which triggers fire in SQL Server:

```
sp_settriggerorder @triggername='ACTION_ITEM_TRG_INSERT_UPDATER',  
                  @order='First', @stmttype = 'UPDATE'
```

where order can be "First", "Last" or "None".

SQL Server Max/Min vs. Top 1 with Order By

Executing the following queries produces an identical query plan:

```
select top 1 * from applicable_action  
ORDER BY applicable_action_id DESC  
  
select * from applicable_action  
where applicable_action_id =  
(SELECT MAX(applicable_action_id) FROM applicable_action)
```

Personally, I think it is easier/quicker to write/interpret the first one!

SQL Server Dropping column bound to default on a user defined type

```
EXEC sp_unbindefault N'[ACTION_ITEM].[IS_PRINTED]'  
  
ALTER TABLE ACTION_ITEM DROP COLUMN IS_PRINTED
```

Bits and pieces

Updated Delphi 8 Licence

An Updated Delphi 8 License is now available from Borland Developer Network at <http://bdn.borland.com/article/0,1410,32293,00.html>. *{Sorry, Folks, you'll have to cut and paste that one - I can't get the link to work. Ed}*

The major change involves point 3 ("Compiled Programs and Redistributables") which not only gives you the right to distribute the redistributable files from Borland (as part of your component or application), but now also gives you the ability to sublicense to your end users the rights to distribute the redistributables. Moreover, 3.6 specifically targets component developers with a special agreement that allows them to grant rights to the components customers (the only thing I'm not sure about is what to do with freeware or Open Source components built with Delphi 8 for .NET). The bottom line is that we can safely distribute the Borland redistributable files as part of our components or applications.

Bob Swart

Delphi 8 for .NET Architect Trial

The Delphi 8 Architect 30-day Trial edition is now available for download from http://www.borland.com/products/downloads/download_delphi_net.html.

Note that the Delphi 8 Architect Trial is a two-part download. The recommended procedure is to download Part 1 first (83,643,156 bytes), this contains extras including Component One, Wise Owl Demeanor, and MDAC as well as a set of configuration files for use with C#Builder, then download and run Part 2 (88,285,589 bytes), the main Delphi 8 install. The extras and main Delphi 8 packages should be extracted to the same location on the local machine.

Bob Swart

Borland Bloggers

A number of Borland developers have made their web log (blog) available, including the following:

- Allen Bauer's blog... by Allen Bauer <http://homepages.borland.com/abauer/>
- Delphi Compiler Core by Danny Thorpe <http://homepages.borland.com/dthorpe/blog/delphi/>
- Anders Ohlsson by Anders Ohlsson http://homepages.borland.com/aohlsson/blog_beta/
- Thoughts from Dan Miser by Dan Miser <http://www.distribucon.com/blog/>

See also Nick's Delphi Blog (from Nick Hodges) at <http://www.lemanix.com/nick/>

Bob Swart

Should we trust our data to CD-Rs?

Clive Henson draws our attention to an article in the Independent Review of 21st April which says that CD-Rs may only last a couple of years and shouldn't be used for archiving as the dyes have a tendency to fade. http://news.independent.co.uk/world/science_technology/story.jsp?story=513486

Thank you for your donation

Thanks to all members who have put money in our charity tin at events, we have once again been able to send a cheque to Marie Curie Cancer Care. This time we have collected £70 (we gave £105 in January 2002).

UG meeting report

Susie Black and Joanna Pooley recently attended a Community Leaders' meeting at Microsoft, with several MS people and ten other user group leaders. The object of the exercise was to give us the opportunity to meet face to face, to share ideas and to work with Microsoft to improve the services available to user group members. You're quite right, that mission statement does sound familiar: this association aims to give user groups what user groups aim to give developers, ie: better and more accessible technical information, better co-ordinated and publicised events diaries, with opportunities for members to attend events hosted by other groups, better resources, eg products, literature, speakers, giveaways, etc, better sharing of ideas and knowledge between user group principals. The mood of the meeting was very positive and signals the start of initiatives led by Microsoft in which the DG and other groups will be participating. More soon.



World-class *training* for
professional developers

Delphi
XHTML/HTML
Javascript
XML
.NET
C#
UML

Brooks Associates
01452 770060
admin@brooksassociates.com

Brooks Associates Ireland
01-832 0373
brooksassociates@eircom.net

www.brooksassociates.com

THE Delphi MAGAZINE

Subscribe to The Delphi Magazine
and get expert help from some of
the most experienced Delphi
and Kylix developers around

Call +44 (0)870 740 7610;
Email info@TheDelphiMagazine.com

Published monthly, each issue includes
regular columns and in-depth feature
articles, plus reviews of relevant add-on
software such as components

Visit our website to find out more
and to view lots of sample articles

www.TheDelphiMagazine.com



www.pentamid.co.uk

Pentamid

**Developers of Software
& Electronic solutions**

Pentamid offers a software, robotics and electronics development service. Pentamid's clients range from very small British companies to large multi-national corporations. Our principal software engineer, Susie Black, has been a Delphi developer since version 1, and has developed award winning software. Susie was previously technical editor of the UK BUG magazine.

Our robotics expert is Dr Dave Keating. Dave was the chief designer of Hasbro's interactive R2-D2 robot toy which was voted "Most innovative and fun toy of 2003" in the USA. Microcontrollers and embedded firmware allow R2 to be voice controlled and play many games. R2 is able to follow people using infrared and sonar sensors, but retails at only \$99.

Change Imperative

by Craig Murphy

I have been using Outlook Express (OE) for a very long time. So much so, I have a directory that's chock full of e-mail and newsgroups - it's 500mb! Whilst OE has served me well over the last few years, I felt that I needed something that would help me organise my e-mail a little better.

OE's filters are reasonable, they work, it's difficult to save them and restore them when the computer is rebuilt (formatted and OS reinstalled). Equally, OE's handling of newsgroups leaves a lot to be desired. Whilst OE can remember .dbx files that contain e-mails, it seems to forget that newsgroup messages are held in .dbx files too. After each rebuild it forces a newsgroup resynchronisation ... which is a real pain to say the least.

I looked at a few replacements for OE and canvassed opinions from my peers. This short article is the result of my findings.

Reasons for Change

Outlook Express is more susceptible to attacks from malicious code such as viruses, worms, Trojans, embedded scripted HTML and who knows what else. This didn't worry me too much as I always use the most up to date anti-virus protection, firewall, and [anti] spy-ware software.

I organised my e-mail using an alphabetic approach. I had 26 folders, A through to Z. Inside the each folder I created another folder Surname, Forename, e.g. inside C I have Cooper, Jim. This works for me, but your mileage may vary!

However OE has little or no means of ranking e-mails. Apart from creating another folder Important, I have little way of identifying an important e-mail from one that isn't at all important. Ideally, some sort of colour coding and filtering would be useful...

This isn't going to be a product review; it's going to be a simple product plug. I'll explain a little about the products that I looked at, and then I'll explain my reasons for choosing particular tools. We all use applications day-to-day, but what applications make you more productive? I see other people using applications that I've never heard of or seen - hopefully this article will prompt you to raise your hand and tell us all about the killer application that you've been assuming we've all be using too.

What did I look at?

It would appear that I am rather picky and choosy when it comes to selecting a new piece of software, especially one that involves a one-way exchange of money!

I looked at PocoMail (<http://www.pocomail.com/>) - this was an excellent contender. It offered advanced filtering and colour coding and was very aesthetically pleasing. However, it did have a couple of flaws that put me off. I was really looking for OE on steroids, i.e. a product that gave me e-mail, newsgroups, filtering, colour coding, etc. PocoMail gave me all of this and more, but it did have a few niggles. My primary annoyance was the newsgroup message view - for some reason it would execute an expand all when moving between newsgroups. This meant that the message's treeview was always fully expanded which made reading each unique thread a chore.

Next, I looked at The Bat! It's a fully-fledged e-mail only client, offering extensive filtering and colour coding. It doesn't do newsgroups, however it's certainly well endowed with organisational features: my favourite is Remind Later. I'm forgetful, so I write things down or set reminders in whatever electronic device is flavour of the day. The Bat lets me flag an e-mail that needs a response - it'll even remind me a day later, a week later, a week later followed by daily or a month later followed by weekly. It's great for making sure I get on top of my article deadlines (I'm hoping that our esteemed Editor doesn't read this bit! *[Selective editing, eh, Craig? Ed].*) However, the scheduling capabilities don't stop there. The Bat also allows the creation of standalone events, i.e. events that are not tied to a particular e-mail. So, if you don't need something as heavyweight as Microsoft Outlook, The Bat offers some useful scheduling/e-mail management in a lightweight tool.

After seeing The Bat, I conceded and felt that I had to adopt two tools: one for e-mail, the other for newsgroups. So I looked at XanaNews for newsgroups. XanaNews is freeware; it's written in Delphi 7 and is supplied with source. Nirvana? It does exactly what I would expect of a newsgroup tool: it has the ubiquitous

treeview on the left hand side of the screen, a pane for threaded conversations and message pane. However, XanaNews has the largest array of configuration settings you could wish for: all layouts, colours, font sizes etc. can be configured to suit your taste. Very impressive.

XanaNews has a really neat and amusing feature in the concept of a Bozo Bin. Newsgroups are frequented by spammers and “guests” who really shouldn’t be there. XanaNews lets us mark such individuals as Bozos, thus relieving us of the need to even pass over their postings again!

The End Result

In the end, I conceded and agreed with myself that two products might be better than one, i.e. one product for e-mail and another for newsgroups. I now use The Bat for e-mail and XanaNews for newsgroups.

So, rounding off, here’s my final toolset:

- E-Mail:** The Bat <http://www.ritlabs.com/en/products/thebat/>
- Newsgroup reading:** XanaNews <http://www.wilsonc.demon.co.uk/delphi.htm>
- Spam protection:** Mailwasher <http://www.firetrust.com/home/index.php>
- URL/Link management:** Linkman <http://www.outertech.com/>

That last one, Linkman is very handy. It lets me manage and work with multiple collections of URLs and links (IR favourites) keeping them synchronised, even between different machines. This means that my three XP installations all share the same collection of links/favourites once Linkman has synchronised.



Craig is an author, developer, speaker, project manager and is a Certified ScrumMaster. He specialises in all things XML, particularly SOAP and XSLT. Craig is evangelical about .NET, C#, Test-Driven Development, Extreme Programming, agile methods and Scrum. He can be reached via e-mail at: bug@craigmurphy.com, or via his web site: <http://www.craigmurphy.com> (where you can also find the source code and PowerPoint files for all of Craig’s articles, reviews and presentations).

Would you like to have more time for the important things in life?


Would you love to get rid of the BDE?

Do you want to improve your performance and stability?

Are you looking for an easy port to Client/Server?



We have THE solution for you:



Advantage Database Server (ADS) – Probably the leanest embedded RDBMS available! Advantage Database Server gives you more time and money for the important things in life...

Advantage Database Server is a true alternative to conventional SQL Client/Server databases – at only a fraction of the cost. ADS already has more than 1.5 million users! See for yourself: www.advantagedatabase.com

ADVANTAGE
DATABASE SERVER

Advantage Database Server offers

- native client kits for Delphi/C++ Builder / Kylix (TDataSet Descendant) including source code
- eliminates need for both BDE & dBaseExpress
- free Advantage Local Server
- multi platform support (MS Windows, Linux, Novell NetWare)
- easy implementation / integration / installation / administration
- scalability (local, peer-to-peer, client/server, internet with one set of code)

Extended Systems®

Extended Systems Bristol Ltd
7 - 8 Portland Square
Bristol BS2 8SM
Tel: +44 (0) 117 901 5000
Fax: +44 (0) 117 901 5001
advantage@extendedsystems.co.uk
www.extendedsystems.com

Learning To Adapt

by Andy Denton

What is a pattern? Well, anyone who has worked on more than one project in their career (which I hope is most of you), will have encountered the same, or very similar problems time and again. In the old days of procedural programming we had standard algorithms to perform certain tasks like a sort, for example. A Design Pattern is essentially the object oriented equivalent of an algorithm. It's a tried and tested method of solving a common programming problem. In this article, I want to cover a pattern that I've used quite a bit over the past few years, the Adapter pattern (also known as the Wrapper pattern). The GoF book describes the Adapter/Wrapper pattern as follows "A wrapper converts the interface of a class into another interface clients expect. Wrappers let classes work together that couldn't otherwise because of incompatible interfaces".

The Problem

Many of you will, over the years, have acquired different libraries for performing a particular task, exporting data, for example. Imagine you have one which handles the exporting of a dataset to a CSV file and another library from another vendor which exports to HTML and XML. This is all well and good, but the two vendors have each developed a distinct interface to their classes. For example one might look something like this:-

```
TExportDataSetCSV = Class
...
  Procedure Execute(pSource : TDataSet; pFileName : String);
...
End;
```

Whereas the other has taken this approach:-

```
THTMLExporter = Class
...
  Property ExportDataSet : TDataSet;
  Property ExportFileName : String;
  Procedure Execute;
...
End;
```

On the face of it this might not seem too much of a problem, as you could define an ordinal type for all the export formats we support

```
Type
  TExportFormat = (efCSV, efHTML, efXML, efExcel);
```

And then code something like this:-

```
Procedure ExportData(pFormat : TExportFormat; pDataSet : TDataSet;
  Const pFileName : String);
Begin
  Case pFormat Of
    efCSV : Begin
      With TExportDataSetCSV.Create(Nil) Do
        Begin
          Try
            Execute(pDataSet, pFilename);
          Finally
            Free;
          End;
        End;
      End;
    efXML : Begin
      With TXMLExporter.Create(Nil) Do
        Begin
          Try
            ExportDataSet := pDataSet;
            ExportFileName := pFileName;
            Execute;
          Finally
            Free;
          End;
        End;
      End;
    End;
  End;
End;
```

```

    efHTML : Begin
        With THTMLExporter.Create(Nil) Do
        Begin
            Try
                ExportDataSet := pDataSet;
                ExportFileName := pFileName;
                Execute;
            Finally
                Free;
            End;
        End;
    End;
End; { Case }
End;

```

I don't know about you, but to me this code "smells bad". It would be much easier if we could just code something like:-

```

Procedure Export(pFormat : TExportFormat; pDataSet : TDataSet;
                Const pFileName : String);
Var
    lExporter : TAbsExporter;
Begin
    lExporter := ExportFactory.GetExport(pFormat);
    Try
        lExporter.DataSet := pDataSet;
        lExporter.FileName := pFileName;
        lExporter.Execute;
    Finally
        lExporter.Free;
    End;
End;

```

NB: - I have cheated somewhat in the above code by making reference to a class factory. I'm not going to cloud the issue by discussing it further here. For the uninitiated, a class factory is another design pattern which will provide a ready created object of the type requested by the client application.

The Solution

Well, we can do something like that. Basically, we create a new set of classes that wrap or adapt the functionality we already have into a uniform set of classes as though all the functionality came from the same vendor. We can choose the interface from the ones we currently have at our disposal or define our own to suit. Here's how we go about it. Firstly, we need to decide on what our common interface is going to look like. This is largely up to you. In our (admittedly contrived) case we have two choices; we can pass the dataset to export and filename, as parameters to the Execute method, or we can set the dataset to export and filename as properties and then call the Execute method. I'm going to opt for the latter for the purposes of this article. So far, then we have something like this:-

```

TAbsExporter = Class
Private
    FDataSet : TDataSet;
    FFileName : String;
Public
    Procedure Execute; Virtual; Abstract;
    Property DataSet : TDataSet Read FDataSet Write FDataSet;
    Property ExportFileName : String Read FFileName Write FFileName;
End;

```

Notice, I haven't done anything meaningful in this class - it's just an abstract class upon which we'll later base our concrete implementations. Let's continue and actually create something that does something. Taking our CSV class first:-

```

TCSVExporter = Class(TAbsExporter);
Private
    FWrappedExport : TExportDataSetCSV;
Public
    Constructor Create;
    Destructor Destroy;
    Procedure Execute; Virtual; Override;
End;

Implementation

```

```

Constructor TCSVExporter.Create;
Begin
  Inherited;
  FWrappedExport := TExportDataSetCSV.Create(Nil);
End;

Destructor TCSVExporter.Destroy;
Begin
  FWrappedExport.Free;
  Inherited;
End;

Procedure TCSVExporter.Execute;
Begin
  If Assigned(DataSet) Then
    FWrappedExport.Execute(DataSet, FFileName)
  Else
    Raise Exception.Create('DataSet not specified!');
End;

```

As you can see, there's nothing particularly special about this class. But what it does do is hide the interface of the wrapped or adapted class, which is the whole point. We can now adapt one of the other export classes thus:-

```

THTMLExport = Class(TAbsExporter)
Private
  FWrappedExport : THTMLExporter;
Public
  Constructor Create;
  Destructor Destroy;
  Procedure Execute; Virtual; Override;
  Property ExportFileName : String Read FFileName Write FFileName;
End;

Implementation

Constructor THTMLExport.Create;
Begin
  Inherited;
  FWrappedExport := THTMLExporter.Create(Nil);
End;

Destructor THTMLExport.Destroy;
Begin
  FWrappedExport.Free;
  Inherited;
End;

Procedure THTMLExport.Execute;
Begin
  If Assigned(DataSet) Then
    Begin
      FWrappedExport.DataSet := DataSet;
      FWrappedExport.ExportFileName := FileName;
      FWrappedExport.Execute;
    End
  Else
    Raise Exception.Create('Dataset not specified!');
  End;

```

What we now have are two new classes performing the same functions as the original two, but they now have an adapted, common interface, making maintenance easier and improving readability.

Other Benefits

The main benefit here might not be immediately apparent. Because we have wrapped or adapted the interface of these classes into one of our own choosing, we can swap the adapted classes with impunity. If we came across some classes which handled all the export formats we needed, we could update our classes accordingly *without changing any code in the applications that use them*. The same would apply if we found or wrote another class to export to another custom export format. If we'd simply dropped components onto forms we'd have considerably more work on our hands.

One of the major additional benefits of the Adapter pattern comes when it is used in conjunction with the Factory Pattern. It will allow you seamlessly to switch implementations of functionality whenever required without any additional coding. An excellent example of this can be seen in the TechInsite Object Persistence Framework (tiOPF) which uses this combination to allow applications developed with it to switch persistence layers (database back ends) at runtime. I've deliberately avoided a discussion on the Factory Pattern in this article as I think it's important to focus on the core concept being discussed. If Ed is agreeable, I can cover this pattern in a future article. *[Yes please! Ed.]*

Finally

It's difficult to know what the consensus is in the general Delphi community is regarding Design Patterns. For all I know, every pattern article written elicits a groan and the secret thought "Oh no, he's going to talk about the Complicator pattern or the Percolator pattern or some other concept with an equally arcane name" and the good reader skips off to read about something more interesting like Dr Bob's latest sojourn into web development. You might think "Oh goody! Another pattern article – this will be good!" or the reaction will be somewhere in between. I'd be very interested to know what people's thoughts are on design patterns in general, so please feel free to drop me a line and let me know what you think.



Andy is Senior Software Engineer at Q-Systems (International) Ltd where he is responsible for developing EPoS applications for the nightclub and restaurant industries. He has been programming Pascal since 1984 (Turbo Pascal from 1986) and with Delphi since it first shipped. In his brief moments of spare time he enjoys playing with his daughters (Jasmine 1 and Molly 4), judo, and playing guitar (though not simultaneously). He can be contacted at adenton@q-systems.com



Developer-friendly
web hosting and
dedicated
servers

with great support

Are you fed up with explaining what you need to your web host? Wish they understood about .NET, ISAPI, CGI and XML? Wish you could get some real technical support?

At TDMWeb we know about software development: we also publish The Delphi Magazine, and deal with developers all day long!

So talk to us and you're talking with friends. We even have a special Windows Developer package designed just for the development and testing of web applications.

From simple shared hosting to your own dedicated server (complete with our Total Management service if you need it), we can do it all.

- Windows and Linux hosting and dedicated servers.
- ISAPI, ASP, ASP.NET, CGI, PHP, Perl, SSI, MySQL, Access, SQL Server, SOAP, XML, XSLT and more.
- Spam and virus protection.
- Excellent prices, great support.

Full package details at www.TDMWeb.com
Call +44 (0)870 740 7610 or Email info@tdmweb.com

www.tdmweb.com **TDMWeb**

.NET Remoting and DataSets Distributed Database Applications with Delphi 8 for .NET

by Bob Swart

In this article, we continue our coverage of .NET Remoting techniques, using it to build a simple multi-tier application, passing a DataSet from the Server to Clients and updates back to the Server.

.NET Remoting

Last time, I explained that a .NET Remoting architecture consisted of a server side and a client side, as well as an object that is used between the server and the client. The object can be a remote object - meaning that it always remains at the server side, and methods are executed at the server side only. Alternately, you can also have so-called mobile objects, that are serialised and sent to the client machine, where they are deserialised and executed.

The remote object remains at the server side, and is referenced to and used by one or more clients. The clients create a reference to the remote object and invoke (remote) methods from this server. The .NET Remoting framework supports different communication protocols (HTTP and TCP), message formats (binary and SOAP) and security (IIS security and SSL), which can all be extended as well.

Remote objects are derived from the class MarshalByRefObject, specifying that the objects will be returned by reference. When a remote object is activated at the server, the client receives a reference to the remote object called a proxy (an instance of TransparentProxy class). The client can then use the proxy to call the remote methods, turning the request into a serialised message by a formatter using a specified format (binary or SOAP). The serialised message is transported to the remote server object using a transport channel that uses HTTP or TCP. At the server side, the remote object receives an incoming message using a transport channel, it has to deserialise the message into a request for a method invocation. The response of the method call is formatted into a response message, transported using the transport channel, received by the client again, and deserialised into the answer that can be used.

Shared Assembly

Using .NET Remoting, the client and server must be able to understand each other. This is achieved by ensuring that both client and server share the same definition (or interface) of the remote object. One of the techniques that is used is a shared assembly, which has to be deployed on the server as well as the client machines. The shared assembly only has to contain the shared interface definition, which is also called the remote object manager interface.

The main focus of this article will be to build a .NET Remoting Server that exposes the EMPLOYEE table from the InterBase Employee.gdb database (as a .NET DataSet), as well as a .NET Remoting Client that can receive this .NET DataSet and work with it, including the ability to send updates back to the server. The interface for the communication between the .NET Remoting Client and Server is defined in the shared assembly.

Using Delphi 8 for .NET, this means you have to start a new package project with *File / New Package*, saving the package in SharedRemoteInterface. Right-click on the Requires node, and choose Add Reference. Use the Add Reference dialog to add the System.Data assembly to the requirements of the package. Next, do *File / New - Unit* add a new unit to the package, and save that unit in file SharedInterface.pas. The interface definition for the remote object can be placed in this unit, and is as follows:

```
unit SharedInterface;
interface
uses
  System.Data;
type
  IRemoteObjectManager = interface
    function GetEMPLOYEE(start, count: Integer): DataSet;
    function SetEMPLOYEE(ClientDS: DataSet): Boolean;
  end;
implementation
end.
```

The shared assembly only contains the interface definition of the remote object. The interface `IRemoteObjectManager` consists of two methods: `GetEMPLOYEE` and `SetEMPLOYEE`. The `GetEMPLOYEE` method will be used to return a .NET `DataSet` which is filled with the records from the `EMPLOYEE` table, starting with record number `start`, and returning count records. If you want to receive all records, you can use 0 as value for `start`, and a very large number as value for `count`.

Remote Server

Once we have the remote object interface definition, we can implement it inside a remote server project. Note that there is a little problem with the Delphi 8 for .NET IDE if you start a new project that references an assembly from a previous project (that you just created in the IDE). To avoid these problems, you have to close the IDE between an assembly project and an assembly-using project.

Restart Delphi 8 for .NET, and create a new project for the remote server. To keep it simple, create a console project for the remote server, and call it `RemoteDataBaseServer`. Right-click on the project node and do `Add Reference` to add the reference to the `SharedRemoteInterface.dll`. Since this assembly will not be installed in the Global Assembly Cache, you should use the `Browse` button to locate it. Note that after you've compiled the console project, this will automatically copy the `SharedRemoteInterface.dll` to your project directory.

You can now implement the remote object, by first adding the `SharedInterface` unit to your `uses` clause, and then writing the following code:

```
type
  RemoteObjectManager = class(MarshalByRefObject, IRemoteObjectManager)
  public
    function GetEMPLOYEE(start, count: Integer): DataSet;
    function SetEMPLOYEE(ClientDS: DataSet): Boolean;
  end;
```

The implementation of the `GetEMPLOYEE` and `SetEMPLOYEE` methods also require the `Borland.Data.Provider` assembly - use the `Add Reference` dialog to add the reference to this assembly as well.

Get/SetEMPLOYEE Implementation

Then, add the `System.Data`, `Borland.Data.Provider`, `Borland.Data.Common`, and `SharedInterface` units to the `uses` clause of the `RemoteDataBaseServer` project, and implement the `GetEMPLOYEE` and `SetEMPLOYEE` methods as follows:

```
function RemoteObjectManager.GetEMPLOYEE(start, count: Integer): DataSet;
var
  Connection: BdpConnection;
  DataAdapter: BdpDataAdapter;

begin
  Result := DataSet.Create;

  Connection := BdpConnection.Create('database=localhost:C:\Program Files\' +
  'Common Files\Borland Shared\Data\employee.gdb;' +
  'assembly=Borland.Data.Interbase,Version=1.5.1.0,Culture=neutral,' +
  'PublicKeyToken=91d62ebb5b0d1b1b;vendorclient=gds32.dll;' +
  'provider=Interbase;username=sysdba;password=masterkey');
  Connection.ConnectionOptions := 'rolename=myrole;' +
  'transaction isolation=ReadCommitted;sqldialect=3;' +
  'waitonlocks=False;loginprompt=False;servercharset=;commitretain=False';
  DataAdapter := BdpDataAdapter.Create('SELECT * FROM EMPLOYEE', Connection);
  DataAdapter.Fill(Result, start, count, 'EMPLOYEE')
end;
```

`Borland.Data.Interbase.dll` versioning: Note the version 1.5.1.0 is the current version (after Update 2) and 1.5.0.0 is the version of before Update 2 of Delphi 8. And also note the `start` and `count` values that are passed to the `DataAdapter`'s `Fill` method. This ensures that we do not have to retrieve the entire contents of the `EMPLOYEE` table at once.

```
function RemoteObjectManager.SetEMPLOYEE(ClientDS: DataSet): Boolean;
var
  Connection: BdpConnection;
  DataAdapter: BdpDataAdapter;
begin
  Result := False;
  Connection := BdpConnection.Create('database=localhost:C:\Program Files\' +
  'Common Files\Borland Shared\Data\employee.gdb;' +
```

```

    'assembly=Borland.Data.Interbase,Version=1.5.1.0,Culture=neutral,' +
    'PublicKeyToken=91d62ebb5b0d1b1b;vendorclient=gds32.dll;' +
    'provider=Interbase;username=sysdba;password=masterkey');
Connection.ConnectionOptions := 'rolename=myrole;' +
    'transaction isolation=ReadCommitted;sqldialect=3;'+
    'waitonlocks=False;loginprompt=False;servercharset=;commitretain=False';
DataAdapter := BdpDataAdapter.Create('SELECT * FROM EMPLOYEE', Connection);
DataAdapter.AutoUpdate(ClientDS, 'EMPLOYEE', BdpUpdateMode.All);
Result := True // success!
end;

```

Note that most of the code is used to initialise the BdpConnection component in both methods. The last few lines, working with the DataAdapter, are more interesting. In the case of SetEMPLOYEE we can call the AutoUpdate method of the DataAdapter which will send the updates back to the database. I'll cover error handling (for example when a record is already deleted or has been changed by another user) at some other time.

System.Runtime.Remoting

Right-click on the project node and add a reference to the System.Runtime.Remoting assembly. This assembly contains a number of useful namespaces, like System.Runtime.Remoting, System.Runtime.Remoting.Channels and System.Runtime.Remoting.Channels.HTTP, that we need to add to the uses clause as well. The complete uses clause should now be as follows:

```

uses
    System.Data,
    Borland.Data.Provider,
    Borland.Data.Common,
    SharedInterface,
    System.Runtime.Remoting,
    System.Runtime.Remoting.Channels,
    System.Runtime.Remoting.Channels.HTTP;

```

We can now implement the server as follows:

```

const
    PortNumber = 4242;
    ServerResource = 'RemoteObjectManager.soap';
var
    Channel: HttpChannel;
begin
    writeln('RemoteServer started. ');
    Channel := HTTPChannel.Create(PortNumber);
    ChannelServices.RegisterChannel(Channel);
    writeln('Listening for SOAP messages on HTTP port ', PortNumber);
    RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(RemoteObjectManager),
        ServerResource,
        WellKnownObjectMode.Singleton); // only one remote object
    writeln('Hit <enter> to stop. ');
    readln
end.

```

Note that you can only run one instance of the remote server application - the WellKnownObjectMode.Singleton will make sure of that.

Remote Client

Time to start the .NET Remoting client application. Since we should be able to use the result of the GetEMPLOYEE method, and send changes back using the SetEMPLOYEE method, we should create a new Windows Forms Application for the client, and save it as RemoteDataBaseClient. Place a DataGrid on the form. Next, right-click on the project and add the System.Runtime.Remoting and SharedRemoteInterface assemblies as references. Also add SharedInterface, System.Runtime.Remoting, System.Runtime.Remoting.Channels, and the System.Runtime.Remoting.Channels.HTTP units to the uses clause.

We need to create an HttpChannel again in order to communicate with the RemoteServer. This time however, we do not have to specify a portnumber for the channel, since this is part of the RemoteServer information. This, as well as the first call to GetEMPLOYEE can be done in the OnLoad event of the WinForm (note that Channel and ObjManager are declared as private fields of the WinForm):

```

type
  TForm1 = class(System.Windows.Forms.Form)
    ...
  private
    { Private Declarations }
    Channel: HttpChannel;
    ObjManager: IRemoteObjectManager;
  end;
...
const
  RemoteServer = 'http://localhost:4242/';
  ServerResource = 'RemoteObjectManager.soap';

procedure TForm1.TForm1_Load(sender: System.Object; e: System.EventArgs);
begin
  Channel := HTTPChannel.Create; // no PortNumber needed
  ChannelServices.RegisterChannel(Channel);
  try
    ObjManager := Activator.GetObject(typeof(IRemoteObjectManager),
      RemoteServer + ServerResource);
  except
    MessageBox.Show('Could not get reference to IRemoteObjectManager.')
  end;
  // now we can use the ObjManager
  DataGrid1.DataSource := ObjManager.GetEMPLOYEE(0,10); // first 10 records
  DataGrid1.DataMember := 'EMPLOYEE'
end;

```

Note that the last two lines of the OnLoad event handler already call the GetEMPLOYEE method and add it to the DataSource property of the DataGrid, as well as assigning the name EMPLOYEE to the DataMember property. This is enough for the .NET Remoting client to create the Remote Object and call the remote method with the DataSet as result, showing the first 10 records of the EMPLOYEE table.

Apart from calling GetEMPLOYEE in the OnLoad event handler, it would be more flexible to have a button with the text "Connect" and only connect to the .NET Remoting server and call the GetEMPLOYEE method if we click on the Connect button. That would ensure that the .NET Remoting client application doesn't hang if you accidentally start it without starting the server first.

The implementation for the OnClick event handler of btnConnect is as follows:

```

procedure TForm1.btnConnect_Click(sender: System.Object; e: System.EventArgs);
begin
  DataGrid1.DataSource := ObjManager.GetEMPLOYEE(0,20); // only 20 records
  DataGrid1.DataMember := 'EMPLOYEE';
end;

```

You can now maintain the number of records, including the last record number. It's possible to add more buttons like "Next 20", "Previous 20", "First", etc. almost like a navigator that will retrieve sets of 20 records. As long as you assign the new DataSet to the DataSource property of the DataGrid, the client will only show 20 records at a time. I leave that as exercise for the reader.

Applying Updates

The next step is not difficult: drop two additional buttons, call them btnUndo and btnUpdate, and implement their Click event handlers as follows:

```

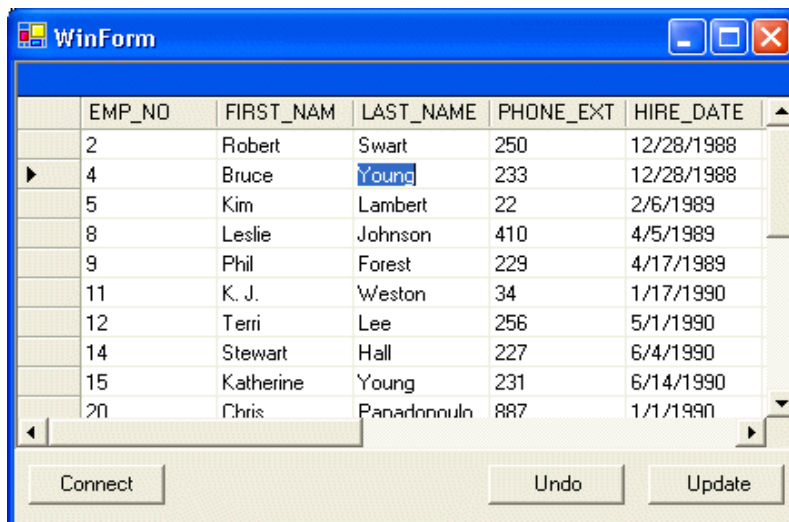
procedure TForm1.btnUndo_Click(sender: System.Object; e: System.EventArgs);
begin
  (DataGrid1.DataSource as DataSet).RejectChanges
end;

procedure TForm1.btnUpdate_Click(sender: System.Object; e: System.EventArgs);
var
  Changes: DataSet;
begin
  try
    Changes := (DataGrid1.DataSource as DataSet).GetChanges;
    if ObjManager.SetEMPLOYEE(Changes) then
      begin
        (DataGrid1.DataSource as DataSet).Merge(Changes);
        (DataGrid1.DataSource as DataSet).AcceptChanges
      end
    except
      on Ex: Exception do
        MessageBox.Show(Ex.StackTrace, Ex.Message)
      end
    end;
end;

```

Note that the Update only sends the changes back to the server using GetChanges. This avoids having to send the entire DataSet (including the changes and all original record field values) - it can save a lot of bandwidth. See <http://www-106.ibm.com/developerworks/db2/library/techarticle/dm-0403swart/> for an article where I implemented a similar distributed dataset architecture, based on ASP.NET Web Services instead of .NET Remoting, but sent not only the changes back from the client to the server, but sent the entire DataSet instead.

Anyway, the resulting .NET Client application is a thin-client, independent of the database type used, and can be used to view, edit and update the EMPLOYEE table using a DataGrid, as shown in the screenshot below:

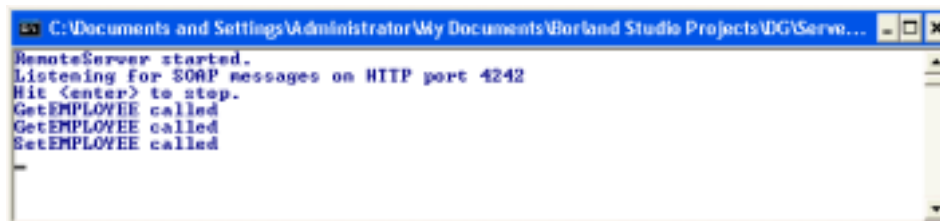


Note that the Undo and Update buttons are enabled, which is only the case if and only if the HasChanges property of the DataSet returns true. The DataSet is actually the DataSource property of the DataGrid, so the check for changes has to be done as follows:

```
procedure TWinForm.DataGrid1_CurrentCellChanged(sender: System.Object; e: System.EventArgs);
begin
  btnUndo.Enabled := (DataGrid1.DataSource as DataSet).HasChanges;
  btnUpdate.Enabled := btnUndo.Enabled;
end;
```

This OnCurrentCellChanged event handler will cast the DataGrid's DataSource property to a DataSet (which it is, since it's the result of the GetEMPLOYEE method), and call the HasChanges method. If this returns True, then both the Undo and Update button should be enabled, otherwise they remain disabled.

Obviously, the .NET Remoting server must be running before the client can connect to it. It will report the calls made to GetEMPLOYEE and SetEmployee, as shown below:



Summary

In this article, we've examined how we can build simple distributed applications with Delphi 8 for .NET, using .NET Remoting as a technique to allow clients to obtain a reference to a remote object. The example implemented a GetEMPLOYEE and SetEMPLOYEE method, passing a .NET DataSet from server to client applications.

Bob Swart (aka Dr Bob - www.drbob42.net) is a software developer, author, trainer, consultant and webmaster for his own one-man company, Bob Swart Training & Consultancy in Helmond, The Netherlands. He writes for numerous computing magazines as well as his own training material, and is also webmaster to the group.



WISDOM

The tribal wisdom of the Dakota Indians, passed down from generation to generation, says that when you discover that you are riding a dead horse, the best strategy is to dismount. In the Public Service, however, a whole range of far more advanced strategies is often employed, such as:

1. Change riders.
2. Buy a stronger whip.
3. Do nothing: "This is the way we have always ridden dead horses".
4. Visit other countries to see how they ride dead horses.
5. Perform a productivity study to see if lighter riders improve the dead horse's performance.
6. Hire a contractor to ride the dead horse. (Can be as useful as a saddle when it comes to protecting your rear.)
7. Harness several dead horses together in an attempt to increase the speed.
8. Provide additional funding and/or training to increase the dead horse's performance.
9. Appoint a committee to study the horse and assess how dead it actually is.
10. Re-classify the dead horse as "living-impaired".
11. Develop a Strategic Plan for the management of dead horses.
12. Rewrite the expected performance requirements for all horses.
13. Modify existing standards to include dead horses.
14. Declare that, as the dead horse does not have to be fed, it is less costly, carries lower overheads, and therefore contributes substantially more to the bottom line than many other horses.
15. Promote the dead horse to a supervisory position. (but the competition for positions is fierce).

thanks to Eamon Donoghue for this

VS.NETcodePrint is an add-in to Microsoft® Visual Studio .NET that enables you to produce professional style printouts of the source code of Visual Basic .NET and Visual C# .NET applications. VS.NETcodePrint professional styled color coded printouts are ideal for:



- Learning programming using Visual Basic .NET and and Visual C# .NET
- Documenting Visual Basic .NET and Visual C# Projects
- Debugging and supporting complex systems developed using Visual Basic .NET and Visual C# .NET
- Code inspections
- Submitting presentations to tutors and/or clients

You can preview the output on screen before printing or exporting to RTF, HTML and PDF formats. The output is fully customizable enabling professional looking color coded output to be generated which can be exported to files with Rich Text Format(RTF), HTML and Adobe Portable Document Format (PDF) formats.



Essential Delphi 8 for .NET

e-book by Marco Cantù

There are links here to the freely available chapter of Marco's work-in-progress e-book. Marco has not only given us permission to reproduce this in our magazine, but has also generously offered extra material exclusively to DG members, at no charge. We'll bring this to you when he has finished writing it.

Marco writes:

Due to a series of reasons, after 7 editions of Mastering Delphi I'm going to skip an edition of the book. But you still get a chance to read my description of the newest Borland offering, through an e-book I'm currently writing, tentatively titled "Essential Delphi 8 for .NET". The book will be released later, when at least most of its contents is ready. For now, you can preview a rather complete draft of Chapter 3, covering the changes in the Delphi language, and there are many. The 40 page chapter is available in PDF format (inside a ZIP file so you cannot browse it online!).

Feel free to read, store for your use, print this file as you wish. The only thing you cannot do is sell it, give it away at seminars you teach, and make any direct or indirect profit from it (unless you get a specific permission from the author, of course). Don't distribute on the web, but refer others to this book page on this web site.

While this chapter is freely available, the complete e-book will not be free, you'll need to pay for it (unless I can line up enough sponsors to make it freely available for all).

Table of Contents

The book is aimed at existing Delphi developers moving to .NET. If you are not fluent in Delphi consider buying and reading Mastering Delphi 7 or another equivalent book first.

The table of contents of the book is still under construction, along with the actual content. This is an early list of things I plan to cover, even if at different depths:

- Chapter 1: A Personal View of .NET
- Chapter 2: Delphi 8 for .NET IDE
- (Free) Chapter 3: The Delphi Language
- Chapter 4: The Delphi RTL
- Chapter 5: FCL Core Libraries
- Chapter 6: Using the VCL for .NET
- Chapter 7: Using WinForms
- Chapter 8: ADO.NET and Other Data Access Technologies
- Chapter 9: Indy and other Internet Programming Techniques
- Chapter 10: Web Sites and Web Services with ASP.NET

Downloads

Language Chapter (ch3): Current Version: 0.03, April 2nd 2004 http://www.marcocantu.com/d8ebook/d8n_ch3_v003.zip

Language Chapter (ch3): Source Code, version 003 http://www.marcocantu.com/d8ebook/d8ncode_03_v003.zip



Some other e-books

(words by the publishers)

Delphi & COM+

This book was originally intended for a publisher company but finally did not fit well into their schedule. Delphi & COM+ is available **free** with samples for Delphi 5, 6 and 7. Over 500 printed pages in MS Word doc format, (450 Kb). <http://www.softinterop.com/files/book/BookCOMPlus.zip>. Free.

Indy in Depth

Indy in Depth from Nevrona: is an e-book written by the Indy experts themselves. Indy in Depth covers from the very basic introduction to sockets, all the way up to advanced usage. Whether you are an absolute beginner and have never programmed sockets before, or you are an advanced socket developer, Indy in Depth has material suited for you. <http://www.atozed.com/indy/book/> Price from 29 Euros.

C# .NET Web Developer's Guide E-book

The focus of The C#.NET Web Developer's Guide is on providing you with code examples that will help you leverage the functionalities of the .NET Framework Class Libraries. http://www.syngress.com/catalog/sg_main.cfm?pid=1675 Price: \$24.95

.NET Mobile Web Developer's Guide E-book

The .NET Mobile Web Developer's Guide provides a solid foundation for developing mobile applications using Microsoft Technologies. http://www.syngress.com/catalog/sg_main.cfm?pid=1755 Price: \$24.95.

ASP.NET Web Developer's Guide E-book

Since 1996, ASP programmers have faced one upgrade after another, often with no visible advantages until version 3.x. Now you have the first significant improvement in ASP programming within your grasp- ASP.NET. http://www.syngress.com/catalog/sg_main.cfm?pid=1669. Price: \$24.95

C# for Java Programmers

This book will compare and contrast many of the advantages and drawbacks of Java and C#, allowing programmers to make informed, intelligent decisions based on the unique uses of each language. http://www.syngress.com/catalog/sg_main.cfm?pid=2235 Price: \$24.95

Developing .NET Web Services with XML E-book

With the development of Microsoft's .NET initiative, industry experts are predicting a new dawn in the age of web development and services. http://www.syngress.com/catalog/sg_main.cfm?pid=2065 Price:\$24.95

Hack Proofing Your Web Applications E-book

The Complete Guide to Developing Secure Web Applications. As a developer, the best possible way to focus on security is to begin to think like a hacker. http://www.syngress.com/catalog/sg_main.cfm?pid=1375 Price: \$24.95

Palm OS Web Developer's Guide E-book

With an 80% hand-held device market-share, the Palm Organizer is the platform of choice for Mobile Internet application developers. http://www.syngress.com/catalog/sg_main.cfm?pid=1395 Price: \$24.95

Webmaster's Guide to the Wireless Internet E-book

The mobile landscape is in a state of continual change. New devices are introduced to the market almost weekly, and wireless access options multiply. http://www.syngress.com/catalog/sg_main.cfm?pid=1595 Price: \$24.95

XML .NET Developer's Guide E-book

Often touted as the new 'lingua franca' of the internet, XML will allow developers to free information from its presentational prison and give their site and services a new flexibility. http://www.syngress.com/catalog/sg_main.cfm?pid=1555 Price: \$24.95

Delphi for .NET Developer's Guide

A sample chapter, courtesy of publishers Pearson Books from the new book by Xavier Pacheco, due for release in May

Chapter 15 Reflection API

In this chapter

- Reflecting an Assembly
- Reflecting a Module
- Reflecting Types
- Runtime Invocation of a Type's Members (Late Binding)
- Emitting MSIL Through Reflection

Reflection is the .NET way to obtain metadata information about classes and types. This is similar in nature to Delphi's the runtime type information (RTTI), although the implementation is substantially different. Metadata is provided via the classes defined in the System.Reflection namespace.

Reflection offers more capabilities than just retrieving metadata information. Through Reflection, you can instantiate various classes, invoke their methods, get and set member values, define and create new classes, dynamically add types to an assembly, and more. Another powerful use of reflection is defining and checking for the presence of custom attributes. These can give additional information about properties, methods, classes, etc. that can be used for debugging, documentation, inspection, design time information etc. This chapter covers the capabilities of the Reflection API.

What Happened to RTTI?

RTTI is similar to Reflection and was the basis behind dynamically discovering information about Delphi VCL classes at run-time. RTTI has not disappeared: It can be found in the Borland.Vcl.TypInfo.pas file, is still used for class discovery of VCL classes, and is a means of portability between the Win32 and .NET versions of VCL.

Reflecting an Assembly

Chapter 8 discusses assemblies and how they contain metadata about the types contained in them. This section discusses reflecting the Assembly itself. The following sections show how to drill down further into the assembly's modules, extracting detailed information about their structure and contents.

The Assembly class itself contains several properties and methods that you can use to reflect information about it. Listing 15.1 illustrates some of these methods.

Listing 15.1 Example of Reflecting an Assembly

```
1: procedure TWinForm.ReflectAssembly(aAssembly: Assembly);
2: var
3:     arModule: array of Module;
4:     i: integer;
5:     tn: TreeNode;
6:     tni: TreeNode;
7:     arAttrib: array of System.Object;
8:     arAsmName: array of AssemblyName;
9:     arType: array of System.Type;
10: begin
11:     // Show key information
12:     tn := TreeView1.Nodes.Add(aAssembly.FullName);
13:     tn.Nodes.Add('Location: '+aAssembly.Location);
14:     if aAssembly.GlobalAssemblyCache then
15:         tn.Nodes.Add('Global Cache: Yes')
16:     else
17:         tn.Nodes.Add('Global Cache: No');
18:
19:     // Determine calling assembly
20:     tn.Nodes.Add('Calling Assembly: '+aAssembly.GetCallingAssembly.FullName);
```

```

21:
22:     // Examine custom attributes
23:     tni := tn.Nodes.Add('Custom Attributes');
24:     arAttrib := aAssembly.GetCustomAttributes(true);
25:     for i := Low(arAttrib) to High(arAttrib) do
26:         tni.Nodes.Add(arAttrib[i].ToString);
27:
28:     // Examine Referenced assemblies
29:     tni := tn.Nodes.Add('Referenced Assemblies');
30:     arAsmName := aAssembly.GetReferencedAssemblies;
31:     for i := Low(arAsmName) to High(arAsmName) do
32:         tni.Nodes.Add(arAsmName[i].FullName);
33:
34:     // Examine types defined in an assembly
35:     tni := tn.Nodes.Add('Assembly Types');
36:     arType := aAssembly.GetTypes;
37:     for i := Low(arType) to High(arType) do
38:         tni.Nodes.Add(arType[i].ToString);
39:
40:     // Examine and drill down into an assembly
41:     arModule := aAssembly.GetModules;
42:     tn := tn.Nodes.Add('Modules');
43:     for i := Low(arModule) to High(arModule) do
44:         ReflectModule(arModule[i], tn);
45: end;

```

Listing 15.1 is actually part of a larger reflection application. It performs some of the same functions as the Reflection utility provided with Delphi for .NET.

An excellent tool for reflecting metadata is Lutz Roeder's Reflector. This utility can not only reflect and display metadata, but it can also disassemble your code into IL or decompile it into other .NET languages. Currently, it supports C#, Visual Basic .NET, and Delphi. You can obtain this utility at <http://www.aisto.com/roeder/dotnet/>

The TWinForm.ReflectAssembly() method shown here takes a reference to an Assembly class instance, which is really a reference to an Assembly class that was previously loaded using the following code:

```

if (OpenFileDialog1.ShowDialog = System.Windows.Forms.DialogResult.OK) then
begin
    Cursor.Current := Cursors.WaitCursor;
    try
        Assem := Assembly.LoadFrom(OpenFileDialog1.FileName);
        ReflectAssembly(Assem);
    finally
        Cursor.Current := Cursors.Default;
    end;
end;

```

This code loads an assembly specified by OpenFileDialog1. It is necessary to load the assembly in order to reflect its metadata.

This code simply populates a TreeView with information about the Assembly. Table 15.1 describes the methods used in Listing 15.1.

Table 15.1 Assembly Class Members

Member	Description
FullName	Retrieves the display name of the assembly. The display name shows the major and minor versions, build and revision numbers, name, culture and public key or public key token.
Location	Retrieves the physical location of the assembly containing the manifest.
GlobalAssemblyCache	Indicates whether the assembly was loaded from the GAC by returning true. Otherwise, false is returned.
GetCallingAssembly()	Gets the assembly whose method invoked the currently executing method.
GetCustomAttributes()	Gets an array of custom attributes defined on this assembly.
GetReferencedAssemblies()	Gets an array of AssemblyName types for any assemblies referenced by this assembly.
GetTypes()	Gets all types that are defined in this assembly.
GetModules()	Gets the modules that make up this assembly.

Table 15.1 is a partial list of the methods and properties in the Assembly class. You can look in the .NET documentation for the complete reference to the Assembly class.

You can see that, in some cases, the metadata is directly obtained through a property or a method. Some methods return an array of items such as the `GetReferencedAssemblies()` and `GetModules()` methods. In these cases, as Listing 15.1 illustrates, you assign the result of these methods to an appropriate array and then reflect the items in the array.

In lines 41-44, the array list of modules is obtained from the assembly object. Each module is then passed to a `ReflectModule()` method, which walks through the module's metadata. The following section discusses this topic.

Reflecting a Module

Listing 15.2 is an example of reflecting Module classes of an assembly.

Listing 15.2 Reflecting a Module

```

1: procedure TWinForm.ReflectModule(aModule: Module; aTn: TreeNode);
2: var
3:   arType: array of System.Type;
4:   arAttrib: array of System.Object;
5:   arFields: array of FieldInfo;
6:   arMeth: array of MethodInfo;
7:   i: integer;
8:   tn: TreeNode;
9:   tni: TTreeNode;
10: begin
11:   tn := aTn.Nodes.Add(aModule.Name);
12:   tn.Nodes.Add('Fully Qualified Name: '+aModule.FullyQualifiedName);
13:   tn.Nodes.Add('Is Resource: '+ TObject(aModule.IsResource).ToString);
14:
15:   // Examine custom attributes
16:   tni := tn.Nodes.Add('Custom Attributes');
17:   arAttrib := aModule.GetCustomAttributes(true);
18:   for i := Low(arAttrib) to High(arAttrib) do
19:     tni.Nodes.Add(arAttrib[i].ToString);
20:
21:   // Examine fields
22:   tni := tn.Nodes.Add('Fields');
23:   arFields := aModule.GetFields;
24:   for i := Low(arFields) to High(arFields) do
25:     tni.Nodes.Add(arFields[i].ToString);
26:
27:   // Examine modules
28:   tni := tn.Nodes.Add('Methods');
29:   arMeth := aModule.GetMethods;
30:   for i := Low(arMeth) to High(arMeth) do
31:     tni.Nodes.Add(arMeth[i].ToString);
32:
33:   // Examine types
34:   arType := aModule.GetTypes;
35:   tn := tn.Nodes.Add('Types');
36:   for i := Low(arType) to High(arType) do
37:     ReflectType(arType[i], tn);
38: end;

```

Listing 15.2 is called from the `ReflectAssembly()` method shown in Listing 15.1 for each module of an assembly. Similar to Listing 15.1, this method consists of calls to members of the module class that return information about itself. This information is then populated in the `TreeView`.

Table 15.2 describes the methods used in Listing 15.2.

Table 15.2 Module Class Members

Member	Description
<code>FullyQualifiedName</code>	Retrieves the fully qualified name including the path to this module.
<code>IsResource()</code>	Indicates whether the object is a resource.
<code>GetCustomAttributes()</code>	Retrieves an array of Custom Attributes for this module.
<code>GetFields()</code>	Returns an array of the type <code>FieldInfo</code> for fields defined for this module.
<code>GetMethods()</code>	Returns an array of the type <code>MethodInfo</code> for methods defined for this module.
<code>GetTypes()</code>	Returns an array of <code>System.Type</code> for types defined in this module.

Table 15.2 is a partial list of the methods and properties in the `Module` class. You can look in the .NET documentation for the complete reference to the `Module` class.

Listing 15.2 performs the same type of functions against Module class as for the Assembly class in Listing 15.1. Various methods and properties are invoked to obtain information about the module. In some cases, this is specific information about the module; whereas in others, an array of certain objects is obtained that must be enumerated and examined.

It is possible to have flat global “methods” (or global routines and global fields defined in a module), but these are not CLS compliant and cannot be defined in neither C#, Visual Basic .NET, nor Delphi. Examining methods and fields is shown in Listing 15.2; however, it unlikely that you would ever encounter them.

Lines 34-37 in Listing 15.2 retrieve an array of types defined in the module. Each type from this list is then passed to the ReflectType() method, which, as the name implies, reflects the details of the specific type. This is discussed in greater detail in the following section.

Reflecting Types

This section shows you how to drill down further into the types defined in an assembly and how to reflect those types.

Listing 15.3 illustrates the processes of reflecting member information for types.

Listing 15.3 Reflecting a Type's Member Information

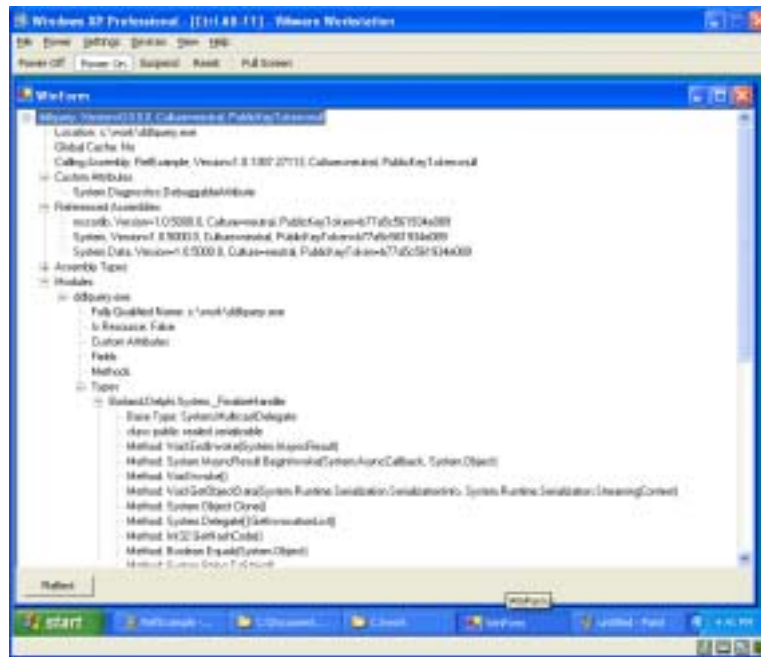
```
1: function GetDeclareString(st: System.Type): String;
2: begin
3:   Result := '';
4:   if st.IsAbstract then Result := Result + 'abstract ';
5:   if st.IsArray then Result := Result + 'array ';
6:   if st.IsClass then Result := Result + 'class ';
7:   if st.IsContextful then Result := Result + 'context ';
8:   if st.IsInterface then Result := Result + 'interface ';
9:   if st.IsPointer then Result := Result + 'pointer ';
10:  if st.IsPrimitive then Result := Result + 'primitive ';
11:  if st.IsPublic then Result := Result + 'public ';
12:  if st.IsSealed then Result := Result + 'sealed ';
13:  if st.IsSerializable then Result := Result + 'serializable ';
14:  if st.IsValueType then Result := Result + 'record ';
15: end;
16:
17: procedure TForm1.ReflectType(aType: System.Type; aTn: TreeNode);
18: var
19:   arMembInfo: array of MemberInfo;
20:   arType: array of System.Type;
21:   i: integer;
22:   tn, fn: TreeNode;
23: begin
24:   tn := aTn.Nodes.Add(aType.FullName);
25:   tn.Nodes.Add('Base Type: '+aType.BaseType.FullName);
26:   tn.Nodes.Add(GetDeclareString(aType));
27:   arMembInfo := aType.GetMembers;
28:
29:   for i := Low(arMembInfo) to High(arMembInfo) do
30:     begin
31:       fn := tn.Nodes.Add(System.String.Format('{0}: {1}',
32:         arMembInfo[i].MemberType, arMembInfo[i]));
33:     end;
34:
35:   arType := aType.GetNestedTypes;
36:   for i := Low(arType) to High(arType) do
37:     ReflectType(arType[i], tn);
38: end;
```

Listings 15.1, 15.2, and 15.3 are all within the same application on the CD. Listing 15.3 consists of two methods, GetDeclareString() and ReflectType(). This listing shows a partial usage of the various methods and properties available for reflecting member information.

The first section of the ReflectType() method (lines 24-25) of Listing 15.3 adds each member's full name and base type to the Treeview. It then passes the member to the GetDeclareString() function. GetDeclareString() builds a string based on the member's access level and other declarative characteristics and adds the string to the TreeView.

The second section retrieves an array of MemberInfo references for the member of the specified type by calling the GetMembers() method. It then adds the type specification for the member and the name of the member to the TreeView (lines 35-36, Listing 15.3).

Figure 15.1 illustrates the output for this program used in Listings 15.1-15.3 in reflecting another assembly.



Runtime Invocation of a Type's Members (Late Binding)

Late binding is a process by which an application can invoke a member of a class (including creating that class) without knowing at compile time of that class or its methods. This is a useful technique, particularly when dealing with add-in functionality to applications. Consider the Delphi for .NET development environment. It must somehow have to determine information about the classes to allow you to visually design applications. It is reflection's capability to perform late binding that enables the IDE to work with classes it does not know about that exist in various assemblies.

To illustrate a simple add-in scheme, the following example consist of two assemblies[md]both of which define a class containing a single method. The method is given the same name for simplicity. An invoking application loads each assembly separately and invokes the methods on both classes. Listings 15.4 and 15.5 list the assemblies.

Listing 15.4 First Assembly Example

```

1:  library asmRemInv1;
2:  type
3:    TClass1 = class
4:    public
5:      procedure WriteMessage(aMsg: String);
6:    end;
7:
8:  procedure TClass1.WriteMessage(aMsg: String);
9:  begin
10:    Console.WriteLine('In asmRemInv1.TClass1: '+aMsg);
11:  end;
12:
13: end.
```

Listing 15.5 Second Assembly Example

```

1:  library asmRemInv2;
2:  type
3:    TClass1 = class
4:    public
5:      procedure WriteMessage(aMsg: String);
6:    end;
7:
8:  procedure TClass1.WriteMessage(aMsg: String);
9:  begin
10:    Console.WriteLine('In asmRemInv2.TClass1: '+aMsg);
11:  end;
12:
13: end.
```

You will notice that both assemblies contain the same class, identically defined. The idea here is that the using application will be able to invoke either of these classes in the same manner. The implementation of these classes might differ. This is similar to using an Interface to define the signature of the class. Consider a class that performs a sorting scheme for instance. These classes could have been developed to perform two different types of sorting, such as a bubble and insertion sort. The calling application should not have to know about the implementation.

At this level, the calling application shown in Listing 15.6 must know something about the classes it is going to invoke, such as the class name and the method of the class. It is possible for the application to know nothing of the class and still be capable to query, through reflection, various methods, determine their parameters, and invoke those methods. The utility of this capability, although powerful, is difficult to determine. At some level, the invoking application should know what it is trying to do with the class it is dynamically invoking.

Listing 15.6 illustrates how these classes are invoked from the calling application.

Listing 15.6[em]Invoking Application

```
1:  program InvProject;
2:
3:  {$APPTYPE CONSOLE}
4:
5:  uses
6:    System.Reflection,
7:    SysUtils;
8:
9:  var
10:   Assem: Assembly;
11:   BFlags: BindingFlags;
12:
13:   procedure InvokeType(aNameSpace: String);
14:   var
15:     obj: System.Object;
16:     Meth: MethodInfo;
17:     t: System.Type;
18:     ParamTypes: array of System.Type;
19:   begin
20:     t := Assem.GetType(aNameSpace+'.TClass1');
21:     Console.WriteLine(t.ToString);
22:     obj := Activator.CreateInstance(t);
23:     Console.WriteLine(obj.ToString);
24:
25:     t.InvokeMember('WriteMessage', BFlags, nil, obj,
26:       ['This is the message']);
27:
28:     // An alternative method is shown below.
29:     SetLength(ParamTypes, 1);
30:     ParamTypes[0] := TypeOf(System.String);
31:     Meth := t.GetMethod('WriteMessage', ParamTypes);
32:     Console.WriteLine(Meth.ToString);
33:     Meth.Invoke(obj, ['This is the message']);
34:   end;
35:
36: begin
37:   BFlags := BindingFlags.DeclaredOnly or BindingFlags.Public or
38:     BindingFlags.Instance or BindingFlags.InvokeMethod;
39:
40:   try
41:     Assem := Assembly.LoadFrom('asmRemInv1.dll');
42:     InvokeType('asmRemInv1');
43:
44:     Assem := Assembly.LoadFrom('asmRemInv2.dll');
45:     InvokeType('asmRemInv2');
46:   except
47:     on E: Exception do
48:       Console.WriteLine('Error: '+E.Message);
49:     end;
50:   Console.ReadLine();
51: end.
```

Listing 15.6 first loads the assemblies and then calls the `InvokeType()` function, which creates the class from the assembly and then calls its `WriteMessage()` function. This code makes use of a few constructs, which I will explain.

The `Activator` class is used to create the instance of the class itself. `Activator` can be used to create instances of classes locally, remotely, or from a COM object using some of its alternate methods. For instance, Listing 15.6 could have used the `Activator.CreateInstanceFrom()` method to create the class without having to explicitly load the assembly. This method takes the classname, a named assembly file, and a constructor name. You might look at the documentation on the `Activator` class to examine its other methods.

Once an instance of the class is created, its method, `WriteMessage()`, is called by using the `InvokeMember()` function.

Table 15.3 describes the parameters required by the `InvokeMember()` function.

Table 15.3 `InvokeMember`'s Parameters

Parameter	Description
name	A string parameter that specifies the name of the member to invoke.
invokeAttr	A <code>BindingFlags</code> type that specifies how members are to be searched for.
binder	A <code>Binder</code> type that specifies how to bind (match) members and their arguments.
target	The target type instance (<code>System.Object</code>) on which to invoke the member.
args	An array of arguments to pass to the member method.
modifiers	An array of <code>ParameterModifier</code> objects[md]each of which represents attributes associated with the corresponding item in the args array
namedParameters	An array containing the names of the parameters that passed as values in the args array.
culture	A <code>CultureInfo</code> that is used to specify the globalization locale.

Note that the `InvokeMember()` method has three overloaded variants depending on which of these parameters are required. In Listing 15.6, only the method name, `invokeAttr`, `binder`, `target`, and `args` parameters are required.

The `invokeAttr` parameter is a bitmask that consists of one or more `BindingFlags` values that control binding and determine how the specified member is to be searched. Some of these flags specify the operation type, such as the `BindingFlags.InvokeMethod` flag, which indicates that a method is going to be called. Other flags specify the access to the member to search on. For instance, `BindingFlags.Public` specifies to include public members in the search. If non-public members are to be included in the search, you would include the `BindingFlags.NonPublic` flag. You will see more use of these flags in the following section. It would be a good idea to be familiar with the various flags that may be used with this parameter. You might look it over in the .NET documentation.

Invoking the Member's Types for Efficiency

Lines 29 to 33 in Listing 15.6 show an alternate way to invoke the method of the type by calling the type's `GetMethod()` function. This function returns a `MethodInfo` instance, which contains an `Invoke()` method that invokes the specified method.

This method is more efficient than calling `InvokeMember()` if you will be calling it repeatedly. The reason is that `InvokeMember()` must perform the search for the member and then bind to it when found every time it is called. By obtaining a reference to the member's type, you can directly access the member without the search/bind operation. Table 15.4 describes the various methods that return a reference to the member type and which of its method to call to invoke that member.

Table 15.4 Accessing Members Through Their Types

Method	Returns	How to Access
<code>GetConstructor()</code>	<code>ConstructorInfo</code>	<code>Invoke()</code>
<code>GetEvent()</code>	<code>EventInfo</code>	<code>AddEventHandler()</code> <code>RemoveEventHandler()</code>
<code>GetField()</code>	<code>FieldInfo</code>	<code>GetValue()</code> <code>SetValue()</code>
<code>GetMethod()</code>	<code>MethodInfo</code>	<code>Invoke()</code>
<code>GetProperty()</code>	<code>PropertyInfo</code>	<code>GetValue()</code> <code>SetValue()</code>

Another Example of Member Invocation

Listing 15.7 illustrates another example of invoking the members of a type through reflection.

Listing 15.7 Member Invocation

```
1: program InvMemb;
2: uses
3:   SysUtils,
4:   System.Reflection;
5: {$APPTYPE CONSOLE}
6:
7: type
8:   TOnWriteEvent = procedure of object;
9:
10:  TMyType = class(TObject)
11:  private
12:    FIntProp: Integer;
13:    FStrProp: String;
14:    FOnWriteEvent: TOnWriteEvent;
15:  public
16:    PublicStr: String;
17:    constructor Create(aStr: String; aInt: Integer;
18:      aStr2: String); override;
19:    procedure WriteSomething;
20:    procedure WriteProc;
21:    property IntProp: Integer read FIntProp write FIntProp;
22:    property StrProp: String read FStrProp write FStrProp;
23:    property OnWriteEvent: TOnWriteEvent read FOnWriteEvent
24:      write FOnWriteEvent;
25:  end;
26:
27:  constructor TMyType.Create(aStr: String; aInt: Integer; aStr2: String);
28:  begin
29:    inherited Create;
30:    PublicStr := aStr;
31:    FIntProp := aInt;
32:    FStrProp := aStr2;
33:    OnWriteEvent := WriteProc;
34:  end;
35:
36:  procedure TMyType.WriteSomething;
37:  begin
38:    Console.WriteLine('PublicStr: '+PublicStr);
39:    Console.WriteLine('FIntProp: '+FIntProp.ToString);
40:    Console.WriteLine('FStrProp: '+FStrProp);
41:    if Assigned(FOnWriteEvent) then
42:      FOnWriteEvent;
43:  end;
44:
45:  procedure TMyType.WriteProc;
46:  begin
47:    Console.WriteLine('—In WriteProc');
48:  end;
49:
50:  var
51:    tp: System.Type;
52:    obj: System.Object;
53:    BFlags: BindingFlags;
54:    parmArray: array[0..2] of System.Object;
55:    s: String;
56:    i: Integer;
57:  begin
58:    BFlags := BindingFlags.DeclaredOnly or BindingFlags.Public
59:      or BindingFlags.Instance;
60:
61:    tp := TypeOf(TMyType);
62:    parmArray[0] := 'hello';
63:    parmArray[1] := System.Object(23);
64:    parmArray[2] := 'world';
65:
66:    try
67:      // create an instance of the type
68:
69:      obj := tp.InvokeMember('.ctor', BFlags or BindingFlags.CreateInstance,
70:        nil, nil, parmArray);
71:      Console.WriteLine(obj.ToString);
```

```

72:
73:     // Call the type's method
74:     (obj as TMyType).WriteSomething;
75:     // or
76:     tp.InvokeMember('WriteSomething', BFlags or BindingFlags.InvokeMethod,
77:         nil, obj, nil);
78:
79:     // Set/Get a field
80:     tp.InvokeMember('PublicStr', BFlags or BindingFlags.SetField, nil, obj,
81:         ['67 Camaro']);
82:
83:     s := String(tp.InvokeMember('PublicStr', BFlags or
84:         BindingFlags.GetField, nil, obj, nil));
85:
86:     Console.WriteLine(s);
87:
88:     // Set/Get a property
89:     tp.InvokeMember('IntProp', BFlags or BindingFlags.SetProperty, nil,
90:         obj, [23]);
91:     tp.InvokeMember('StrProp', BFlags or BindingFlags.SetProperty, nil,
92:         obj, ['Coffee']);
93:
94:     i := Integer(tp.InvokeMember('IntProp', BFlags or
95:         BindingFlags.GetProperty, nil, obj, nil));
96:     s := String(tp.InvokeMember('StrProp', BFlags or
97:         BindingFlags.GetProperty, nil, obj, nil));
98:
99:     Console.WriteLine('Get/Set Property: {0}, {1}', [i, s]);
100:
101: except
102:     on E: Exception do
103:         Console.WriteLine(E.Message);
104:     end;
105:
106:     Console.ReadLine;
107: end.

```

In this example, the class is defined in the same application for simplicity. This example illustrates how through reflection, one can invoke various members of a given class, specifically the TMyType defined in lines 10 - 25.

TMyType declares various members such as a constructor, a procedure, a field, and a few properties.

TMyType.WriteSomething() simply writes the values contained in the various member to the console.

TMyType is created using a different technique than that previously illustrated in Listing 15.6. Instead of using the Activator class to create the class, we call the type's InvokeMember() function. This example passes as the member name, .ctor, which is the .NET name of the constructor. In fact, when you declare a constructor and name it Create(), the Delphi for .NET compiler will generate a constructor named ".ctor" with the appropriate parameters (See Figure 15.2). In reality, the first parameter is ignored. According to the .NET documentation, when you specify the BindingFlags.CreateInstance flag, it instructs Reflection to create an instance of the specified type and call the constructor that matches the arguments provided. It is included here for illustrative purposes.

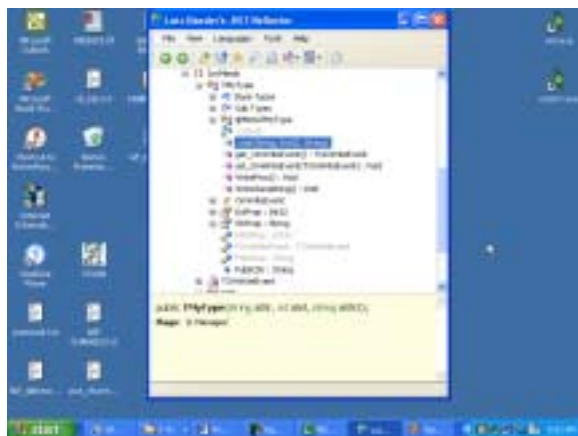


Figure 15.2 Definition of TMyType.

Note also that `BindingFlags.CreateInstance` is included in the flags passed to `InvokeMember()`. By passing this flag, `InvokeMember()` will find the constructor that matches the arguments specified and will invoke that constructor and return an instance of the class. `InvokeMember()` also accepts the array of parameters required by `TMyType`'s constructor.

The remaining code simply illustrates how to access the various members of `TMyType` through the `InvokeMember()` function and by passing the appropriate flag along with the parameter containing the `BindingFlags`.

Emitting MSIL Through Reflection

The .NET framework provides you with the capability to create code on-the-fly through the `System.Reflection.Emit` namespace. The code that you create through `Emit` is MSIL code. Essentially, the functionality in this namespace gives you the ability to dynamically create an assembly and a module within that assembly. You can dynamically define any number of classes and methods within the assembly's module, and then invoke those methods at runtime (which will automatically be JIT compiled to very efficient machine code). You can even save the assembly to disk for later use.

In addition to having the capability to generate MSIL code through `System.Reflection.Emit`, the .NET CodeDom technology provided by the `System.CodeDom` and `System.CodeDom.Compiler` namespaces supply the ability to generate code in languages such as C#, Visual Basic .NET, and Delphi (assuming that you have the proper licensed assemblies installed).

So why would anybody want to have the ability to generate code on-the-fly? There are several ideas thrown around for having this ability. First, visual design tools, such as Delphi for .NET and Microsoft Visual Studio.NET, generate code on-the-fly: Although this is in the form of a .NET language, the idea is the same. The possibility of an end user design tool that generates executable code is not unthinkable. This could also be used to generate more powerful "scripts" for your ASP.NET application. It will be interesting to see where and how far, practically, such a technology will go.

Tools/Classes for Emitting MSIL

Basically four types of tools exist that you will need to generate MSIL code. These are Info classes from the `System.Reflection` namespace, builder classes, and the `ILGenerator` and `OpCodes` classes.

Info classes are defined in the `System.Reflection` namespace. Examples of these are `MethodInfo`, `EventInfo`, `FieldInfo`, and so on.

Builder classes are defined in the `System.Reflection.Emit` namespace. These classes are used to define and create specific parts of the assembly, such as the assembly itself, the module, and a method. Table 15.5 describes the various builder classes.

Table 15.5 Builder Classes from `System.Reflection.Emit`

Builder Class	Description
<code>AssemblyBuilder</code>	Used to define and build a dynamic assembly.
<code>ConstructorBuilder</code>	Used to define and build a dynamic constructor for a class.
<code>CustomAttributeBuilder</code>	Used to define and build a dynamic custom attribute.
<code>EnumBuilder</code>	Used to define and build a dynamic enumeration type.
<code>EventBuilder</code>	Used to define and build a dynamic event for a class.
<code>FieldBuilder</code>	Used to define and build a dynamic field for a class.
<code>LocalBuilder</code>	Used to define and build a dynamic local variable for a method.
<code>MethodBuilder</code>	Used to define and build a dynamic method for a class.
<code>ModuleBuilder</code>	Used to define and build a dynamic module for an assembly.
<code>ParameterBuilder</code>	Used to define and associate parameters.
<code>PropertyBuilder</code>	Used to define and build dynamic properties for a class.
<code>TypeBuilder</code>	Used to define and build dynamic classes and records (value types).
The <code>ILGenerator</code>	is a special class that is used to generate MSIL code. Recall that MSIL is the input to the JIT Compiler.
<code>OpCodes</code>	are specific MSIL instructions and are represented by the <code>OpCodes</code> class in the <code>System.Reflection.Emit</code> namespace.

The Emitting Process

Emitting code is not that different from writing code in the sense that the tasks involved are somewhat sequential. For instance, you first create your assembly. You then create the module. Finally, you create your classes and define the specifics of those classes such as members, methods, and so on. This process will best be illustrated through an example.

A System.Reflection.Emit Example

Listing 15.8 illustrates the tasks involved in using the classes defined in System.Reflection.Emit to generate MSIL code.

Listing 15.8 Example of Using System.Reflection.Emit

```
1: program EmitDemo;
2:   {$APPTYPE CONSOLE}
3:
4:   uses
5:     System.Reflection,
6:     System.Reflection.Emit;
7:
8:   var
9:     appDmn: AppDomain;
10:    asmName: AssemblyName;
11:    asmBuilder: AssemblyBuilder;
12:    modBuilder: ModuleBuilder;
13:    typBuilder: TypeBuilder;
14:    methBuilder: MethodBuilder;
15:    ilGen: ILGenerator;
16:    newType: System.Type;
17:
18:   procedure CreateWLCall;
19:   var
20:     parms: array of System.Type;
21:     methInfo: MethodInfo;
22:     wlTypes: array of System.Type;
23:     SysConsoleType : System.Type;
24:   begin
25:     SetLength(Parms, 1);
26:     Parm[0] := typeof(System.String);
27:
28:     methBuilder := typBuilder.DefineMethod('Main', MethodAttributes.Public
29:       or MethodAttributes.Static, typeof(integer), System.Type.EmptyTypes);
30:
31:     ilGen := methBuilder.GetILGenerator;
32:
33:     SetLength(wlTypes, 1);
34:     wlTypes[0] := typeof('System.String');
35:     methInfo := typeof(System.Console).GetMethod('WriteLine', wlTypes);
36:
37:     if methInfo = nil then
38:       raise Exception.Create('unable to find WriteLine');
39:
40:
41:     ilGen.Emit(OpCodes.Ldstr, 'Delphi for .NET');
42:     ilGen.Emit(OpCodes.Call, methInfo);
43:     ilGen.Emit(OpCodes.Ldc_I4_0);
44:     ilGen.Emit(OpCodes.Ret);
45:   end;
46:
47:   begin
48:     asmName := AssemblyName.Create;
49:     asmName.Name := 'D4DNeDevGuideAsm';
50:
51:     appDmn := AppDomain.CurrentDomain;
52:     asmBuilder := appDmn.DefineDynamicAssembly(AsmName,
53:       AssemblyBuilderAccess.Save);
54:
55:     modBuilder := asmBuilder.DefineDynamicModule(asmName.Name,
56:       'd4dmasm.exe');
57:     typBuilder := ModBuilder.DefineType('Class1');
58:     CreateWLCall;
59:
60:     asmBuilder.SetEntryPoint(methBuilder, PEFileKinds.ConsoleApplication);
61:
62:     newType := typBuilder.CreateType();
63:     asmBuilder.Save('d4dmasm.exe');
64:     System.Console.Write('done');
65:   end.
```

Listing 15.8 generates the IL code for a simple assembly containing a class with a main method that writes a statement to the Console.

The main block of Listing 15.8 performs the functions of creating the dynamic assembly, the module, and the type for this class. As part of this process, the `CreateWLCall ()` function is invoked (line 58), which performs the process of emitting the code for the method itself.

Dynamic entities are typically defined within the context of another entity. For instance, an assembly is created within the context of an `AppDomain`. A module is created within the context of an assembly. A type is created within the context of a module, and so on. Listing 15.8 reflects this structure.

As stated, a dynamic assembly must be created within the context of an `AppDomain`, so this example uses the current `AppDomain` (line 51). To create the assembly, the `AppDomain.DefineDynamicAssembly()` method is invoked. There are several overloaded versions of this method that return an `AssemblyBuilder`. The version used in this example takes `AssemblyName` and `AssemblyBuilderAccess` parameter types. `AssemblyBuilderAccess` is an enumeration type whose values dictate the access level for the dynamic assembly. Valid values are `Run`, `RunAndSave`, and `Save`, which represent an assembly that can be executed, executed and saved, or just saved, respectively. The example here will just save the assembly to disk, so the value passed is `Save`.

The next task is to build the module (lines 55-56). This is done by calling the `AssemblyBuilder.DefineDynamicModule()` method. This method also has overloaded variations, and it returns a `ModuleBuilder` class. The version used here passes the assembly name and the name of the file to which the assembly will be persisted. The method to actually save the assembly is called later, and it must use the same filename used here.

Once the `ModuleBuilder` is obtained, types can be defined within it. Line 57 creates a class named "Class1", and the next line calls the `CreateWLCall()` procedure, which creates a method for this class[md]more on this shortly.

Once the method is created and associated with a `MethodBuilder` class, it is set as the entry point for the assembly by calling the `AssemblyBuilder.SetEntryPoint()` method (line 60). This method takes the `MethodBuilder` class and a `PEFileKinds` type as a parameter. `PEFileKinds` is an enumeration type that specifies what type of PE file is generated from the dynamic assembly. Its values can be `ConsoleApplication`, `DLL`, and `WindowApplication`. This example generates a console application.

Finally, lines 62 and 63 create the type and save it to disk. At this point, one can actually execute the file.

The `MethodBuilder()` procedure is where the `ILGenerator` and `OpCodes` are used.

Lines 28 - 29 invoke the `TypeBuilder.DefineMethod()` method, which returns a `MethodBuilder` class. `DefineMethod()` takes the method name, method attributes, the return type, and an array of parameter types. When defining a method with no parameters, you can pass the `System.Type.EmptyTypes` type, which is a predefined empty array of type.

Given a `MethodBuilder` class, the code can now emit the code for the method. This is really just the process of writing out the IL code that you would type if you were coding in MSIL. In this case, you can use information from types through reflection to generate the IL code. Line 31 retrieves an `ILGenerator` instance. The first `OpCode` specified is `LdStr`, which loads a string reference onto the stack. It then uses reflection to find the method info for the `System.Console.WriteLine()` method[md]specifically, the version that takes a single string parameter. `System.Type.GetMethod()` is used to obtain the `MethodInfo` class for this method. With `MethodInfo`, `ILGenerator.Emit()` defines the `Call` `OpCode` on for the `WriteLine()` method (line 42). Last, the method is returned by emitting the `Ret` `OpCode` (line 44) after placing its value of zero on the stack (line 43)

The `ILGenerator` class has a helper method, `EmitWriteLine()`, that can be used to simplify emitting the MSIL for a `WriteLine()` method. Listing 15.8 takes the longer approach for illustrative purposes.

To order this book:

From Amazon via the DG website at £28.11 (UK) - £25.36 + p&p £2.16 + £0.59 per item
http://www.amazon.co.uk/exec/obidos/ASIN/0672324431/qid=1083772168/sr=2-2/ref=sr_2_3_2/202-6003699-4455066#product-details

or £29.20 (£36.50 less 20% with free p&p in Europe) by special offer from the publishers before June 1st
<http://www.pearson-books.com/voucher/> Voucher code: ZL004D

Delphi.NET Developers Guide also contains a chapter by Brian Long.

Serving Up RSS using Delphi 8 (part 2)

by Craig Murphy

In my last instalment in this series of articles about RSS, Delphi 8 and blogging, I described how I have set about trying to make some pages on my website more dynamic via the use of some simple PHP scripts and RSS files. Part 1 was a little disjointed and was really setting the scene and direction for this and future articles.

Over the course of this article I will be using Delphi 8 (with update 2 applied). I will be describing how I used some of the ideas from part 1 to automate the creation of my Developers Group downloads page at <http://www.craigmurphy.com/bug>. A rather cool side-effect of this will be the creation of an RSS feed for my Developers Group downloads.

Where are we going?

Since the publication of part 1, some of you may have noticed the presence of a little orange icon on my website. If you point your browser to <http://www.craigmurphy.com/bug/bug.xml> you will find an RSS 2.0 feed that contains references to all of my BUG/DDG/DG source code and presentations. Believe it or not, I actually use that very same feed to build the content at <http://www.craigmurphy.com/bug>. This means that when I re-publish bug.xml those users who have syndicated my DG content receive an update when their RSS aggregator polls my site. However, what's a great timer-saver for me is the fact that I do not have to touch the HTML page at <http://www.craigmurphy.com/bug>.

So, over the course of this article I will be describing how I create bug.xml. In the next article in this series, I will demonstrate the PHP code that is required to create HTML from bug.xml – it is this HTML that is rendered in your browser when you visit: <http://www.craigmurphy.com/bug>.

First Things First

We know that RSS feeds are nothing more than a specific XML format. It can assumed that we will need a means of loading and saving the RSS feed – it's an XML file, so we can make use of .NET's XmlDocument object found in the System.XML namespace.

So, given the following variable declaration:

```
uses System.Xml;  
...  
var xmlDoc : XmlDocument;
```

We could write:

```
xmlDoc := XmlDocument.Create;  
xmlDoc.Load(FileName);
```

Assuming that FileName referenced a valid XML document (or an RSS feed), xmlDoc will provide us with access to the XML document via a Document Object Model (DOM). If you are unsure about the DOM, visit <http://www.craigmurphy.com/bug> and look for "Using OpenXML and the XDOME".

More usefully however, given an XmlDocument, we have can use the SelectSingleNode method with an XPath expression to gain granular access to specific nodes with XmlDocument.

I've chosen to wrap XML node selection up and have created a method, __GetNode, to help me out.

__GetNode takes an XML Document and an XPath expression. It will either return the node's textual value or an empty string if the node doesn't exist. It's basic, but let's get things working before we set about improving them.

Here's how we might make use of __GetNode:

```
Title := __GetNode(xmlDoc, 'rss/channel/title');
```

__GetNode's implementation simply calls SelectSingleNode:

RAIZE SOFTWARE

CodeSite

When your code isn't doing what you expect, find out why with CodeSite



- New for CodeSite 3...**
- CodeSite Projects
 - Automatic Message Filtering
 - Automatic Views
 - XML Support
 - .NET Support
 - Custom Views
 - Enhanced Thread Support
 - Plus much more...

CodeSite helps developers locate problems in their code by enabling them to send detailed information from within their application code to a specialized receiver.

Send Any Information

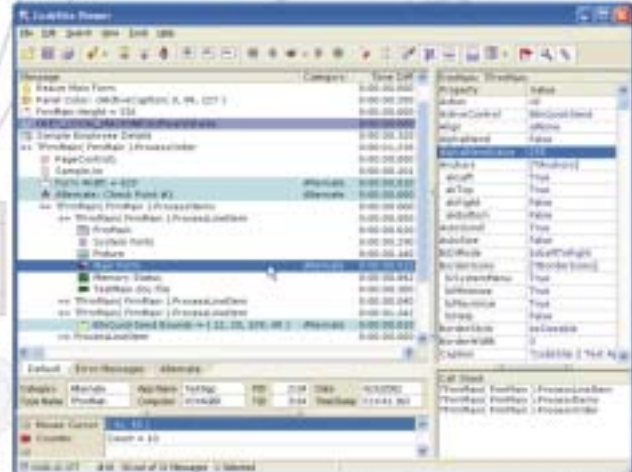
CodeSite supports sending snapshots of all kinds of data, including all standard types, objects, string lists, variants, bitmaps, files, memory statistics, registry entries, custom data, and much more.

Receive the Messages

CodeSite messages can be sent to a variety of destinations including the CodeSite Viewer, a log file, a remote computer, and even a web server.

Analyze the Messages

The CodeSite Viewer (screen shot) provides a powerful and highly-flexible environment for analyzing CodeSite messages.



Raize DropMaster

OLE Drag-and-Drop made easy

Simply Powerful

DropMaster encapsulates the complexity of OLE inter-application drag-and-drop in 4 easy-to-use components.



Drag Sources

Easily support dragging text, images, or custom data formats to other applications.

Drop Targets

Accept plain text, rich text, file lists, URL links, images, and custom data formats from other applications that supports OLE drag and drop.

Customize the Process

Special events are provided so that the drag and drop process can be customized.

Real World Examples

DropMaster comes with more than 30 example projects that demonstrate the extreme power and flexibility of these components.

Examples Include:

- Accept messages dragged from Outlook, Outlook Express, Netscape, and Eudora.
- Dragging URLs from an application to a browser or the desktop.
- Dragging multiple custom formats
- Dragging and creating multiple files
- Accepting OLE object links
- Dragging JPEG images
- Dragging custom content to Explorer folders or the desktop

Raize Components

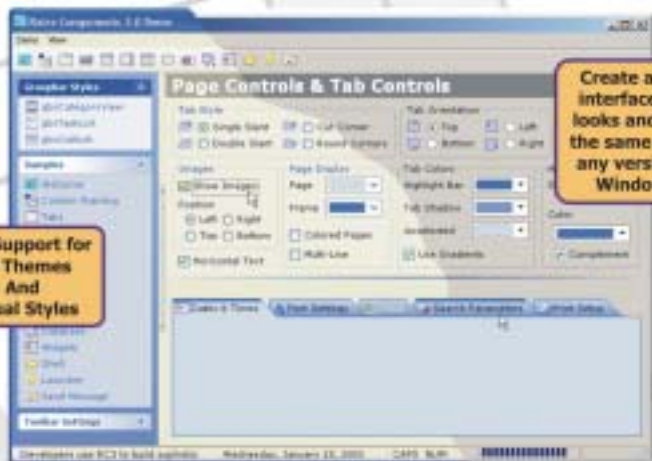
Build sophisticated user interfaces in less time with less effort

User Interface Design System
Raize Components includes **more than 120** general-purpose native VCL controls for use in Delphi and C++Builder.

Component Innovations
Raize Components also features one-step installation, automatic help integration, and dynamic component registration.

Powerful and Easy to Use
RC3 also includes **more than 100** component designers focused on simplifying user interface development.

Source Code Included
Complete source code for all components, packages, and design editors is provided at no additional charge.



Full Support for XP Themes And Visual Styles

Create a user interface that looks and feels the same under any version of Windows.

www.raize.com

```

function __GetNode(nd : XmlNode; xpath : string) : string;
var Node : XmlNode;
begin
  Node := nd.SelectSingleNode(xpath);
  if Node <> nil then
    __GetNode := Node.InnerText
  else
    __GetNode := '';
end;

```

Now that we know how to load and work with an RSS feed, let's set about building a class that will provide us with a simple, but extensible means of working with RSS feeds.

Programmatic Handling of an RSS Document

I know that we looked at an RSS feed in part 1 of this series of articles however, for the sake of completeness, here's a snippet of the bug.xml:

```

<?xml version="1.0"?>
<rss version="2.0">
<channel>
  <title>CraigMurphy.com - Developers Group Downloads</title>
  <link>http://www.craigmurphy.com/bug</link>
  <description>I write articles and give presentations for the UK Borland User Group. From
    time to time I need to make source code or PowerPoint slides available - this is
    where to find them.</description>
  <language>en-uk</language>
  <copyright>Copyright 2001-2004 Craig Murphy</copyright>
  <managingEditor>craig.bug.rss@craigmurphy.com</managingEditor>
  <webMaster>craig.bug.rss@craigmurphy.com</webMaster>
  <pubDate>Sun, 7 Mar 2004 22:00:00 GMT</pubDate>
  <lastBuildDate>Sun, 7 Mar 2004 22:00:00 GMT</lastBuildDate>
  <generator></generator>
  <docs>http://blogs.law.harvard.edu/tech/rss</docs>
  <image>
    <title>Craig Murphy.com: Developers Group Downloads</title>
    <link>http://www.craigmurphy.com/</link>
    <url>http://www.craigmurphy.com/images/logo.gif</url>
  </image>

  <item>
    <title>Using OpenXML and the XDOM</title>
    <link>http://www.craigmurphy.com/bug</link>
    <description>Each archive contains the original article as a PDF, source code and GIF
      files for each of the figures &lt;br /&gt;
      &lt;a href="http://www.craigmurphy.com/bug/xdom/OpenXML1.zip"&gt;Download&lt;/a&gt; the Part
      1 (136K): XDOM basics, creating, loading, saving&lt;br /&gt;
      &lt;a href="http://www.craigmurphy.com/bug/xdom/OpenXML2.zip"&gt;Download&lt;/a&gt; the Part
      2 (115K): Traversing XML using a TreeWalker and an Iterator&lt;br /&gt;
      &lt;br /&gt;Thanks are due to the &lt;a href="http://www.richplum.co.uk/"&gt;Developers
      Group&lt;/a&gt; for their kind permission to reproduce these here.
    </description>
    <author>craig.bug.rss@craigmurphy.com</author>
    <comments>http://www.craigmurphy.com/bug</comments>
    <pubDate>Sun, 7 Mar 2004 22:00:00 GMT</pubDate>
  </item>
  ...
</channel>

```

An RSS feed has a root element <channel> that contains a number of elements describing the RSS feed followed by zero or more <item> elements.

We are going to need a means of representing each of the <item> elements. Keeping things simple, TRSSItem goes some way to allowing us to represent an <item> element:

```

TRSSItem = class
  public
    Title,
    Link,
    Description,
    Author,
    Comments,
    PubDate : string;
end;

```

TRSS provides us with a wrapper for the entire RSS feed. It deals with the RSS-specific elements and the variable number of <item> elements. Using the System.Collection namespace, the ArrayList object provides us with a means of working with a collection of objects of any type (although we will stick to items of type TRSSItem).

```
TRSS = class
private
  rssItems : ArrayList;
public
  Title,
  Link,
  Description,
  Language,
  Copyright,
  ManagingEditor,
  WebMaster,
  PubDate,
  LastBuildDate,
  Generator,
  Docs,
  Img,
  ImgTitle,
  ImgLink,
  ImgURL : string;

  constructor Create(FileName : string); overload;
  function SaveFeed(FileName : string) : Boolean;

  function GetRSSItem(itemNo : integer) : TRSSItem;
  function GetRSSItemCount : integer;
  procedure DeleteRSSItem(itemNo : integer);
  procedure AddRSSItem(RSSItem : TRSSItem);
end;
```

TRSS also provides us with a handful of management methods:

Create takes a filename representing an existing RSS feed – it loads the XML document populating an ArrayList containing TRSSItem instances representing the <item> elements.

SaveFeed simply takes the rssItems collection and saves them as an RSS 2.0 feed format, which is nothing more than vanilla XML.

GetRSSItemCount returns the number of <item> elements that were found in the RSS feed.

GetRSSItem takes an integer as a parameter and returns the TRSSItem representing the said <item>.

DeleteRSSItem also takes an integer as a parameter; it deletes the said RSS <item>.

AddRSSItem takes an instance of TRSSItem and adds it to the collection rssItems.

Using TRSS

Assuming that an RSS feed already exists, TRSS can load it using the following code:

```
Var rss : TRSS;
...
rss := TRSS.Create('bug.xml');
```

There is a lot to the Create method. It has to understand how to work with XML and has to honour the RSS XML format. Luckily, using Delphi 8 and .NET, this is not such a burden. Indeed, .NET's XmlDocument class provides us with access to an XML document's elements and attributes using XPath expressions – this means we can write some rather concise code that accesses the specific RSS elements. We touched on this earlier in this article when we looked at the __GetNode method – more on this later.

Here is the code for the Create method:

```
constructor TRSS.Create(FileName : string);
var xmlDoc : XmlDocument;
  xmlNodes : XmlNodeList;
  Node : XmlNode;
  rssItem : TRSSItem;
  str : string;
  NodeEnum : IEnumerator;
```

```

function __GetNode(nd : XmlNode; xpath : string) : string;
var Node : XmlNode;
begin
  Node := nd.SelectSingleNode(xpath);
  if Node <> nil then __GetNode := Node.InnerText
  else __GetNode := '';
end;

begin
  inherited Create;

  xmlDoc := XmlDocument.Create;
  xmlDoc.Load(FileName);

  Title      := __GetNode(xmlDoc, 'rss/channel/title');
  Link       := __GetNode(xmlDoc, 'rss/channel/link');
  Description := __GetNode(xmlDoc, 'rss/channel/description');
  Language   := __GetNode(xmlDoc, 'rss/channel/language');
  Copyright  := __GetNode(xmlDoc, 'rss/channel/copyright');
  ManagingEditor := __GetNode(xmlDoc, 'rss/channel/managingEditor');
  Language   := __GetNode(xmlDoc, 'rss/channel/language');
  WebMaster  := __GetNode(xmlDoc, 'rss/channel/webMaster');
  PubDate    := __GetNode(xmlDoc, 'rss/channel/pubDate');
  LastBuildDate := __GetNode(xmlDoc, 'rss/channel/lastBuildDate');
  Generator  := __GetNode(xmlDoc, 'rss/channel/generator');
  Docs       := __GetNode(xmlDoc, 'rss/channel/docs');
  ImgTitle   := __GetNode(xmlDoc, 'rss/channel/image/title');
  ImgLink    := __GetNode(xmlDoc, 'rss/channel/image/link');
  ImgURL     := __GetNode(xmlDoc, 'rss/channel/image/url');

  rssItems := ArrayList.Create;

  xmlNodes := xmlDoc.SelectNodes('rss/channel/item');

  NodeEnum := xmlNodes.GetEnumerator;
  while NodeEnum.MoveNext() do begin
    rssItem := TRSSItem.Create;
    Node := NodeEnum.Current As XmlNode;

    str:=Node.OuterXml;

    rssItem.Title      := __GetNode(Node, 'title');
    rssItem.Link       := __GetNode(Node, 'link');
    rssItem.Description := __GetNode(Node, 'description');
    rssItem.Author     := __GetNode(Node, 'author');
    rssItem.Comments   := __GetNode(Node, 'comments');
    rssItem.PubDate    := __GetNode(Node, 'pubDate');

    rssItems.Add(rssItem);
  end;
end;

```

SelectSingleNode is a useful function (specific to MSXML and the .NET framework) that takes XPath expressions to allow us essentially to get the information held in specific nodes. You can see that I have wrapped its functionality up in the __GetNode method (whenever I write a method that is local to another method I tend to prefix the method with a double underscore).

Create also uses an enumerator to iterate over the <item> nodes found in the RSS feed. Whilst we had access to enumerators in Win32 (pre-Delphi 8) versions of Delphi, their use was not as commonplace as it is/will be in Delphi 8, largely because of their extensive usage in the .NET framework. It is likely that heavy users of COM in a Win32 environment will be familiar with the use of enumerators. As you can see from the while loop in the Create method, their .NET syntax is pleasant and not as littered as their Win32 counterpart.

Iterating Over RSSItems

The overloaded TRSS Create method builds an ArrayList of TRSSItems representing the <item> elements found in the RSS feed. Typically we will want to iterate over that collection, possibly adding them to a ListView for example. The following code snippet demonstrates how we might achieve this:

```

var colNo, itemNo : integer;
    rssItem : TRSSItem;
    lvItem : ListViewItem;
...

```

```

lvRSS.Items.Clear;
for itemNo := 0 to rss.GetRSSItemCount - 1 do begin
  rssItem := rss.GetRSSItem(itemNo);

  lvItem := lvRSS.Items.Add(rssItem.Title);
  lvItem.SubItems.Add(rssItem.PubDate);
end;

```

Adding New <items>

```

procedure TRSS.AddRSSItem(rssItem: TRSSItem);
var newrssItem : TRSSItem;
begin
  newrssItem := TRSSItem.Create;

  newrssItem.Title      := rssItem.Title;
  newrssItem.Link       := rssItem.Link;
  newrssItem.Description := rssItem.Description;
  newrssItem.Author     := rssItem.Author;
  newrssItem.Comments   := rssItem.Comments;
  newrssItem.PubDate    := rssItem.PubDate;

  rssItems.Insert(0, newrssItem);
end;

```

Saving an RSS Feed

Now that we have seen how to add new RSS items, we need to be able to save the RSS items for subsequent upload to a website. Like the load (Create) method, the SaveFeed method has to know how to work with .NET's XmlDocument class and know about the RSS 2.0 XML format.

I have kept the implementation of the SaveFeed method fairly straightforward, preferring to get it working before I strengthen it and refactor it into something a little more elegant. Its operation is perhaps obvious: create an XML document, create the RSS elements <rss>, <channel> and an <item> for each item in the rssItems collection. For good measure, this method relies on its own internal properties and methods, notably, GetRSSItem and GetRSSItemCount.

```

function TRSS.SaveFeed(FileName: string): Boolean;
var xmlDoc : XmlDocument;
  xmlDec : XmlDeclaration;
  xmlRoot, newNode, channelNode, imageNode : XmlNode;
  xmlAtt : XmlAttribute;
  itemNo : integer;

  procedure __AddElement(destNode : XmlNode; NodeName, NodeContent : string);
  begin
    newNode := xmlDoc.CreateNode(XmlNodeType.Element, NodeName, '');
    newNode.InnerText := NodeContent;
    destNode.AppendChild(newNode);
  end;

  procedure __AddRSSItem(rssItem : TRSSItem);
  var itemNode : XmlNode;
  begin
    itemNode := xmlDoc.CreateNode(XmlNodeType.Element, 'item', '');
    __AddElement(itemNode, 'title',      rssItem.Title);
    __AddElement(itemNode, 'link',       rssItem.Link);
    __AddElement(itemNode, 'description', rssItem.Description);
    __AddElement(itemNode, 'author',     rssItem.Author);
    __AddElement(itemNode, 'pubDate',    rssItem.PubDate);
    channelNode.AppendChild(itemNode);
  end;

begin
  xmlDoc := XmlDocument.Create;

  xmlDec := xmlDoc.CreateXmlDeclaration('1.0', '', '');
  xmlDoc.AppendChild(xmlDec);

  xmlRoot := xmlDoc.CreateElement('rss');
  xmlAtt := xmlDoc.CreateAttribute('version');
  xmlAtt.InnerText := '2.0';
  xmlRoot.Attributes.Append(xmlAtt);

```

```

xmlDoc.AppendChild(xmlRoot);

channelNode := xmlDoc.CreateNode(XmlNodeType.Element, 'channel', '');

__AddElement(channelNode, 'title', Title);
__AddElement(channelNode, 'link', Link);
__AddElement(channelNode, 'description', Description);
__AddElement(channelNode, 'language', Language);
__AddElement(channelNode, 'copyright', Copyright);
__AddElement(channelNode, 'managingEditor', ManagingEditor);
__AddElement(channelNode, 'webMaster', WebMaster);
__AddElement(channelNode, 'pubDate', PubDate);
__AddElement(channelNode, 'generator', Generator);
__AddElement(channelNode, 'docs', Docs);

imageNode := xmlDoc.CreateNode(XmlNodeType.Element, 'image', '');
__AddElement(imageNode, 'title', ImgTitle);
__AddElement(imageNode, 'link', ImgLink);
__AddElement(imageNode, 'url', ImgURL);
channelNode.AppendChild(imageNode);

for itemNo := 0 to GetRSSItemCount - 1 do
  __AddRSSItem( GetRSSItem(itemNo) );

xmlRoot.AppendChild(channelNode);

xmlDoc.Save(FileName);

SaveFeed := True;
end;

```

Listing 1 presents the TRSS class that has been discussed here. If you visit <http://www.craigmurphy.com/bug> you will be able to download the full source code for a simple RSS management application. It is this application that I will be covering in forthcoming articles, so expect the application to grow arms and legs over the coming months.

Summary

Over the course of this article I have demonstrated how we can use Delphi 8 to work with an RSS formatted XML document. Whilst we didn't cover very much ground, we did see how to load and save XML documents using Delphi 8. We also have the bare bones of a reusable RSS class: we can create an RSS feed/file, add items, delete items and save the feed as an XML file.

In my next article I will add some gloss to this application – after all, the code presented here is far from perfect! We'll see how to generate RSS feeds from scratch and we'll take a look at how we can augment this application to handle more than one RSS feed. I deliberately glossed over some important aspects of RSS, such as the RFC822 date format. All RSS dates must adhere to this standard and whilst the code in this article matches that standard, it's not very flexible.

Also in the next article will be more coverage about how to include HTML tags in an RSS feed without the need to “escape” them using “<” and “>”.

For those of you who are already blogging (writing a web log/diary), I am sure it has not escaped your attention that many blogs are served up via an RSS feed. To that end, you will find a few notable blogs listed at the end of this article.

I know that I am repeating myself, but I am finding that RSS is an excellent way of staying on top of things: news comes to me rather than me having to seek out the news. Elsewhere in this publication you will find me rambling on about my e-mail/link (favourites/bookmarks) manager/newsgroups applications that I use. I discussed a product called LinkMan – it has a “Daily Links” feature which provides a dumping ground for URLs that I would like to visit daily (or more realistically, frequently). It is still a chore visiting a handful of sites each day to see if there is new content. If there is an RSS feed, let the new content come to you!

Lastly...

As I write this I learn of the untimely death of friend and colleague Jon Jenkinson. I had the pleasure and privilege of essentially being mentored by Jon during my early presentations at BUG meetings. My first ever article for the BUG magazine was reviewed by Jon – his positive (and negative!) feedback has shaped and influenced my writing style – permanently.

Jon was also a catalyst in my presence at DCon 2001 as a “yellow shirt”. For some reason this is my most fond memory of Jon: a yellow shirt, black trousers, bouncing from room to room, sneaking the occasional fag in between times!

Jon was one of the nicest blokes you could ever expect to meet; he would go out of his way to help and was always there to give a very frank and honest answer to any question posed. It upsets me that I didn't manage to visit Jon and Debra more often – JJ, you are greatly missed.



Resources

- [1] Implementing RSS using PHP: <http://www.richplum.co.uk/magazine/bug200401/eJanFeb2004.pdf>
 - [2] RSS Aggregators: <http://www.hebig.org/blogs/archives/main/000877.php>
 - [3] A readable version of the RSS 2.0 specification: <http://feedvalidator.org/docs/rss2.html>
 - [4] RSS (and Atom 0.3) feed validation: <http://feedvalidator.org>
- RSSReader (requires .NET): <http://www.rssreader.org>
- FeedReader: <http://www.feedReader.com/>

RSS News Feeds

- <http://www.richplum.co.uk/DG.xml>
- <http://www.craigmurphy.com/bug/bug.xml>
- <http://community.borland.com/article/0,1410,32010,00.html>
- <http://news.borland.com/>

Notable Blogs

- Anders Ohlsson: http://homepages.borland.com/aohlsson/blog_beta/index.html
- Allen Bauer: <http://homepages.borland.com/abauer/>

Craig is an author, developer, speaker, project manager and is a Certified ScrumMaster. He specialises in all things XML, particularly SOAP and XSLT. Craig is evangelical about .NET, C#, Test-Driven Development, Extreme Programming, agile methods and Scrum. He can be reached via e-mail at: bug@craigmurphy.com, or via his web site: <http://www.craigmurphy.com> (where you can also find the source code and PowerPoint files for all of Craig's articles, reviews and presentations).

Listing 1 – A simple class that manages an RSS feed

```
unit ClsRSS;

interface

uses System.Collections;

type
  TRSSItem = class
  public
    Title,
    Link,
    Description,
    Author,
    Comments,
    PubDate : string;
  end;

  TRSS = class
  private
    rssItems : ArrayList;
  public
    Title,
    Link,
    Description,
    Language,
    Copyright,
```

```

ManagingEditor,
WebMaster,
PubDate,
LastBuildDate,
Generator,
Docs,
Img,
ImgTitle,
ImgLink,
ImgURL : string;

constructor Create(FileName : string); overload;
function SaveFeed(FileName : string) : Boolean;

function GetRSSItem(itemNo : integer) : TRSSItem;
function GetRSSItemCount : integer;
procedure DeleteRSSItem(itemNo : integer);
procedure AddRSSItem(RSSItem : TRSSItem);
procedure UpdateRSSItem(itemNo : integer; RSSItem : TRSSItem);
end;

implementation

uses System.Xml;

(* _____ *)

procedure TRSS.AddRSSItem(rssItem: TRSSItem);
var newrssItem : TRSSItem;
begin
    newrssItem := TRSSItem.Create;

    newrssItem.Title := rssItem.Title;
    newrssItem.Link := rssItem.Link;
    newrssItem.Description := rssItem.Description;
    newrssItem.Author := rssItem.Author;
    newrssItem.Comments := rssItem.Comments;
    newrssItem.PubDate := rssItem.PubDate;

    rssItems.Insert(0, newrssItem);
end;

(* _____ *)

constructor TRSS.Create(FileName : string);
var xmlDoc : XmlDocument;
xmlNodes : XmlNodeList;
Node : XmlNode;
rssItem : TRSSItem;
str : string;
NodeEnum : IEnumerator;

function __GetNode(nd : XmlNode; xpath : string) : string;
var Node : XmlNode;
begin
    Node := nd.SelectSingleNode(xpath);
    if Node <> nil then __GetNode := Node.InnerText
    else __GetNode := '';
end;

begin
    inherited Create;

    xmlDoc := XmlDocument.Create;
    xmlDoc.Load(FileName);

    Title := __GetNode(xmlDoc, 'rss/channel/title');
    Link := __GetNode(xmlDoc, 'rss/channel/link');
    Description := __GetNode(xmlDoc, 'rss/channel/description');
    Language := __GetNode(xmlDoc, 'rss/channel/language');
    Copyright := __GetNode(xmlDoc, 'rss/channel/copyright');
    ManagingEditor := __GetNode(xmlDoc, 'rss/channel/managingEditor');
    Language := __GetNode(xmlDoc, 'rss/channel/language');
    WebMaster := __GetNode(xmlDoc, 'rss/channel/webMaster');
    PubDate := __GetNode(xmlDoc, 'rss/channel/pubDate');
    LastBuildDate := __GetNode(xmlDoc, 'rss/channel/lastBuildDate');

```

```

Generator      := __GetNode(xmlDoc, 'rss/channel/generator');
Docs           := __GetNode(xmlDoc, 'rss/channel/docs');
ImgTitle       := __GetNode(xmlDoc, 'rss/channel/image/title');
ImgLink        := __GetNode(xmlDoc, 'rss/channel/image/link');
ImgURL         := __GetNode(xmlDoc, 'rss/channel/image/url');

rssItems := ArrayList.Create;

xmlNodes := xmlDoc.SelectNodes('rss/channel/item');

NodeEnum := xmlNodes.GetEnumerator;
while NodeEnum.MoveNext() do begin
    rssItem := TRSSItem.Create;
    Node := NodeEnum.Current As XmlNode;

    str:=Node.OuterXml;

    rssItem.Title      := __GetNode(Node, 'title');
    rssItem.Link       := __GetNode(Node, 'link');
    rssItem.Description := __GetNode(Node, 'description');
    rssItem.Author     := __GetNode(Node, 'author');
    rssItem.Comments   := __GetNode(Node, 'comments');
    rssItem.PubDate    := __GetNode(Node, 'pubDate');

    rssItems.Add(rssItem);
end;
end;

(* _____ *)

procedure TRSS.DeleteRSSItem(itemNo: integer);
begin
    rssItems.RemoveAt(itemNo);
end;

(* _____ *)

function TRSS.GetRSSItem(itemNo: integer): TRSSItem;
begin
    GetRSSItem := rssItems.Item[itemNo] As TRSSItem;
end;

(* _____ *)

function TRSS.GetRSSItemCount: integer;
begin
    GetRSSItemCount := rssItems.Count;
end;

(* _____ *)

function TRSS.SaveFeed(FileName: string): Boolean;
var xmlDoc : XmlDocument;
    xmlDec : XmlDeclaration;
    xmlRoot, newNode, channelNode, imageNode : XmlNode;
    xmlAtt : XmlAttribute;
    itemNo : integer;

    procedure __AddElement(destNode : XmlNode; NodeName, NodeContent : string);
    begin
        newNode := xmlDoc.CreateNode(XmlNodeType.Element, NodeName, '');
        newNode.InnerText := NodeContent;
        destNode.AppendChild(newNode);
    end;

    procedure __AddRSSItem(rssItem : TRSSItem);
    var itemNode : XmlNode;
    begin
        itemNode := xmlDoc.CreateNode(XmlNodeType.Element, 'item', '');
        __AddElement(itemNode, 'title',      rssItem.Title);
        __AddElement(itemNode, 'link',      rssItem.Link);
        __AddElement(itemNode, 'description', rssItem.Description);
        __AddElement(itemNode, 'author',    rssItem.Author);
        __AddElement(itemNode, 'pubDate',   rssItem.PubDate);
        channelNode.AppendChild(itemNode)
    end;
end;

```

```

begin
  xmlDoc := XmlDocument.Create;

  xmlDec := xmlDoc.CreateXmlDeclaration('1.0', '', '');
  xmlDoc.AppendChild(xmlDec);

  xmlRoot := xmlDoc.CreateElement('rss');
  xmlAtt := xmlDoc.CreateAttribute('version');
  xmlAtt.InnerText := '2.0';
  xmlRoot.Attributes.Append(xmlAtt);
  xmlDoc.AppendChild(xmlRoot);

  channelNode := xmlDoc.CreateNode(XmlNodeType.Element, 'channel', '');

  __AddElement(channelNode, 'title', Title);
  __AddElement(channelNode, 'link', Link);
  __AddElement(channelNode, 'description', Description);
  __AddElement(channelNode, 'language', Language);
  __AddElement(channelNode, 'copyright', Copyright);
  __AddElement(channelNode, 'managingEditor', ManagingEditor);
  __AddElement(channelNode, 'webMaster', WebMaster);
  __AddElement(channelNode, 'pubDate', PubDate);
  __AddElement(channelNode, 'generator', Generator);
  __AddElement(channelNode, 'docs', Docs);

  imageNode := xmlDoc.CreateNode(XmlNodeType.Element, 'image', '');
  __AddElement(imageNode, 'title', ImgTitle);
  __AddElement(imageNode, 'link', ImgLink);
  __AddElement(imageNode, 'url', ImgURL);
  channelNode.AppendChild(imageNode);

  for itemNo := 0 to GetRSSItemCount - 1 do
    __AddRSSItem( GetRSSItem(itemNo) );

  xmlRoot.AppendChild(channelNode);

  xmlDoc.Save(FileName);

  SaveFeed := True;
end;

(* _____ *)

procedure TRSS.UpdateRSSItem(itemNo: integer; RSSItem: TRSSItem);
begin
  with TRSSItem(rssItems[itemNo]) do begin
    Title := RSSItem.Title;
    Link := RSSItem.Link;
    Description := RSSItem.Description;
    Author := RSSItem.Author;
    Comments := RSSItem.Comments;
    PubDate := RSSItem.PubDate;
  end;
end;

end.

```

Listing 1 – A simple class that manages an RSS feed

