Delphi Meets C# Via COM

by Craig Murphy

Unless you have been living under a rock for the last year or so, you will have noticed that Microsoft has focused its attention firmly in the internet camp. It is doing this in a variety of different ways, most of them involving the moniker '.NET'. As part of this move towards greater internet integration, and perhaps as a result of legal issues with Sun, Microsoft has created (yet another) programming language that claims to be the panacea for all the failings in the programming languages that we all use today. This all-singing, all-dancing language is C# (pronounced C-Sharp). Like the musical notation it is named after, C# is intended to be a cut above C, and a shade below C++. Thus, C# has all the good bits of C and C++, but none of the bad bits. Allegedly.

C# as a language was designed by Anders Hejlsberg and Scott Wiltamuth. You should recognise one of those names: Anders was a key architectural figure during the design and build of Delphi. As you might expect, there are some Delphi-isms in C#. During 2000, Microsoft submitted the C# language to the European Computer Manufacturers Association (ECMA) Technical Committee. The ECMA TC comprises various groups, one of which has a mission to 'standardise the syntax and semantics of a general-purpose, cross-platform, vendor-neutral scripting language'. The same group created ECMAScript, a standardised version of Microsoft's JScript and Netscape's JavaScript. Assuming that the ECMA TC accepts and standardises C#, vendors such as Borland will be able to implement the language themselves (they will even be able to extend the language if they so wish).

This article is about interoperability between Delphi and the .NET platform: using C# code in a Delphi environment and vice versa. Over the course of this article I will be using Visual Studio .NET Beta 1: at the time of writing (May 2001) this was the most recent version. By the time you read this, Beta 2 may be available: it is expected to ship at TechEd Europe 2001 (early July), so you may find some minor changes and improvements. Similarly, if you are reading this article and are using the 'Release To Manufacturing' (RTM) version, you should *expect* that some areas will have changed. A detailed discussion about VS.NET and the .NET Framework is beyond the scope of this article. However, you will find a good introduction in Steve Scott's *Perspectives* column in Issue 68 (April 2001).

The remainder of this article will cover three areas. First, a .NET and C# preamble. Then, using C# (.NET) objects in your Delphi application. Finally, I'll discuss using your Delphi objects in a .NET application.

Motivation

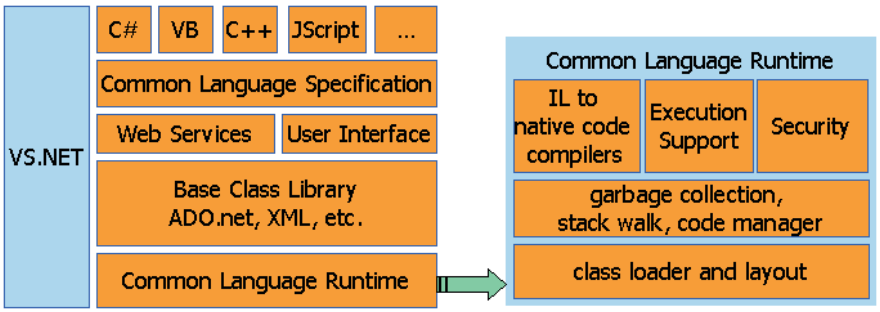
You are probably asking yourself the question: 'Why would I want to build a Delphi application that uses a C# object?' The answer is quite simple: integration with your client's application. Greater integration brings with it the obvious opportunity for business-to-business (B2B) transactions: after all, you have a mechanism for interacting with your client's data and systems.

Similarly, it is very possible that you may wish to use Delphi (or C++Builder) objects in the .NET platform. This is a scenario that is a little easier to imagine. I am sure we all have a lot of tried and tested business logic embedded inside COM objects. So, if we are able to re-use existing COM objects in .NET applications, we will be able to build our .NET applications a lot faster, because we are not starting to write the application from scratch. If we are re-using code, we may continue to maintain it, thus our existing investment is protected. Equally, we do not have to migrate an entire application either: re-use allows us to gradually move our application piece by piece.

Introducing C#

C# has enjoyed a lot of attention since it was announced last year, and a lot has been written about it already. Therefore, this section is not going to be a C# tutorial (difficult to justify in a Delphi/Kylix publication!). Instead, it will be a very brief overview. If you wish to learn more about C#, there are a few websites and books mentioned in the *Resources* sidebar at the end of the article.

C# is an object-oriented language that allows the development of plain vanilla Windows applications, class libraries, web services and Windows services. It is a language that offers productivity and safety. Productivity, because it is a well-thought out mix of C and C++,



➤ *Figure 1: The bigger picture.*

therefore there is a multitude of existing developers who can quickly get up to speed with C#. Safety, because C# is, by default, type-safe. I use the phrase 'by default' because it is possible to mark sections of C# code as being 'unsafe': such code can manipulate pointers and perform those (dodgy) typecasts that C/C++ programmers claim makes them so productive!

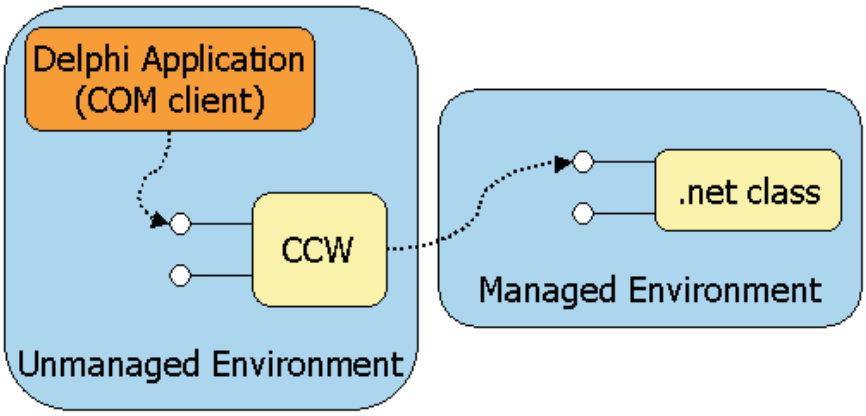
Positioning C# In .NET

You may find .NET rather overwhelming, after all it is technically a whole new platform. A new platform heralds a new architecture, a new way of doing things, and .NET is no exception. Continuing the current trend, .NET brings with it a barrel-load of new acronyms to confuse and amuse! C# sits alongside the other '.NET languages', most of which should be familiar to you, at least in name: Visual Basic, Visual C++ and JScript. No doubt you will have read (with amusement) that COBOL is considered a .NET language.

The .NET compilers (of which C# is one) create Microsoft Intermediate Language (MSIL), often referred to as IL. IL is converted into native code in three ways: Just-In-Time (JIT), on demand, or when the application is loaded.

The need for IL stems from the notion of a Common Language Runtime (CLR). The CLR has to support more than one development language, therefore it has to manage a wide range of data types. This is where it gets a little complex: if we are to achieve complete interoperability between all the IL-emitting compilers it is imperative that we use only the data types and features that are common to all the languages. To ease the complexity, the Common Language Specification (CLS) provides a set of rules that we can use to help ensure that the code we write is 'CLS-compliant'. The CLR and CLS are geared up to support a variety of languages. As you might imagine, porting your existing code and applications to the equivalent .NET language may require a moderate amount of rework.

The idea behind IL is not new, it is similar to the p-code that traditional compilers generate. Whilst you might be thinking that there is a performance overhead, the ability to choose when the IL is converted into native code reduces that overhead. Couple this with the primary benefit that IL is not bound to any particular processor and I think you will agree that IL has some definite advantages. Note that IL is converted into native code, it is never interpreted, thus we dispense with the performance hit that is typically associated with interpreters. Once IL has been converted into native code it can be added to an 'assembly cache' (defined in the *Glossary* sidebar), meaning that any perceived performance hit is negated.



➤ *Figure 2: Unmanaged meets managed: the CLR creates a CCW at runtime.*

Figure 1 explains where C# fits into the .NET platform. There is a lot there, far too much for one article *[Phew! Ed]*. We will be focusing our attention on C# and the CLR. Particular attention will be given to the Execution Support element: it provides 'wrappers' that allow interoperability between the world of COM and the brave new world of .NET.

From .NET To Delphi

The MSIL code that is generated by the C# compiler is said to execute as *managed code*. Managed code interacts with the CLR and provides the metadata (defined in the *Glossary* sidebar) that the CLR then uses to provide such services as: memory management, language interoperability, garbage collection, versioning and security.

This is in contrast to *unmanaged code*, such as COM components and the Win32 API, which are executed outside the CLR.

We will be creating a C# object, therefore it will be executed as managed code. However, the Delphi application that instantiates the C# object has to run in an unmanaged environment. Essentially, we will need to build a bridge between the managed code and the unmanaged code. Thankfully, the CLR helps us out: it will provide a proxy object that allows

➤ *Figure 3: Unmanaged versus managed.*

Characteristic	Unmanaged Environment (COM)	ManagedEnvironment (.net platform)
Type definition	Type library	Metadata
Identity	Guids	Shared names
Coding model	Interface based	Object based
Versioning	Immutable	Resilient
Error-handling mechanism	HResults	Exceptions
Type safety	Unsafe	Safe by default
Type compatibility	Binary standard	Type standard

unmanaged COM components to use managed (C#) classes. This proxy object is known as a COM Callable Wrapper (CCW). Figure 2 graphically depicts the high-level architecture of the unmanaged-to-managed bridge.

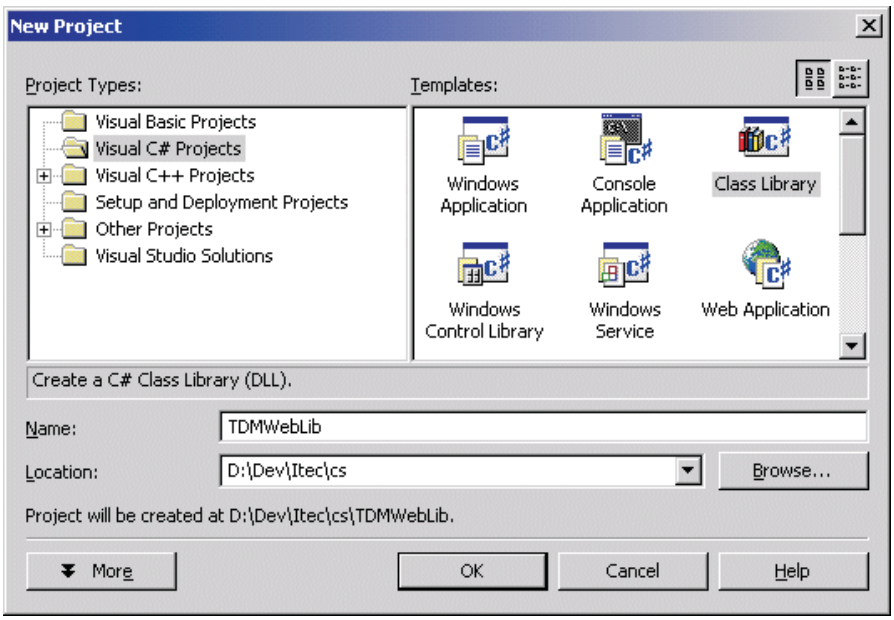
Architecturally, we are able to use managed .NET components in an unmanaged environment. Figure 3 shows key characteristics that differ between unmanaged and managed environments, or rather between the COM environment and the .NET platform. Over the course of this article we will touch on most of these characteristics. We can think of the CCW as an arbitrator: it is capable of providing an abstraction between the managed and unmanaged environments.

Our First C# Object

This is a magazine about Delphi and Kylix, therefore it might prove difficult justifying the inclusion of VS.NET screenshots! Figure 4 is the New Project dialog from VS.NET; the C# object that we are going to create is in fact a C# Class Library. I have chosen to create my project in my D:\Dev\Itec\cs directory; obviously your directory structure will be different.

I will be creating a trivial C# object, but one that is useful nonetheless: a TDMWeb hosting calculator application *[No, I didn't bribe Craig to do this! Ed]*. The logic is simple: a Delphi client application will present the user with the standard TDMWeb packages (Basic, Advanced and Professional), and various options. We all know that the cost of hosting can go up as well as down, so TDMWeb needs control over the costs. The costs are implemented using some C#, as presented in Listing 1. Listing 1 presents three functions, one for each of the packages. In addition, each of these functions takes a parameter that specifies the combination of options that the client has selected. Bit-wise operators are used to create and extract the parameter (the `&` operator is used in C# and the `SHL` operator is used in Delphi). Figure 5 depicts the individual values for each option; obviously they can be combined to allow multiple options, ie 3 represents the first two options, 17 represents the first and last option.

➤ *Figure 4: Creating a C# object using VS.NET.*



➤ *Figure 5: Dealing with TDMWeb package options.*

Integer Representation	Option
1	Register UK domain name (eg .co.uk)
2	Register US domain name (eg .com)
4	Additional 50MB web space (Professional Package only)
8	Easy-Secure: secure form-to-encrypted-email processing
16	Pro-Secure: 20Mb of your own secure server space

What's In A Name?

Figure 3 suggested that shared names are not too dissimilar to GUIDs. Shared names are used to

➤ *Listing 1: Our first trivial C# class.*

```
namespace TDMWebLib
{
  using System;
  /// <summary>
  ///    Summary description for Package.
  /// </summary>
  public class Package
  {
    public Package()
    {
      //
      // TODO: Add Constructor Logic here
      //
    }
      private double optionCost(int iOption)
      {
        long lCost = 0;
        if ((iOption & 1) > 0)   lCost = lCost + 12;
        if ((iOption & 2) > 0)   lCost = lCost + 30;
        if ((iOption & 4) > 0)   lCost = lCost + 30;
        if ((iOption & 8) > 0)   lCost = lCost + 30;
        if ((iOption & 16) > 0)  lCost = lCost + 100;
        return lCost;
      }
      public double Basic(int iOption)
      {
        return 60 + optionCost(iOption);
      }
      public double Advanced(int iOption)
      {
        return 100 + optionCost(iOption);
      }
      public double Professional(int iOption)
      {
        return 150 + optionCost(iOption);
      }
  }
}
```

verify a .NET component's identity and authenticity. A shared name is actually a simple text name (string) and a version number; however, a public key and a digital signature accompany it. We generated a *shared name* using the sn.exe utility, the shared name is stored in the TDMKey.snk file. TDMKey.snk is then compiled into the TDMWebLib assembly (a DLL in this case). Figure 6 depicts the successful output from sn.exe.

➤ *Figure 6: Creating a shared name key.*

```
D:\Dev\Itec\cs\TDMWEB~1>sn -k TDMKey.snk

Microsoft (R) .NET Strong Name Utility.  Version 1.0.2204.21
Copyright (C) Microsoft Corp. 1998-2000

Key pair written to TDMKey.snk
```

Compiling The C# Class

Listing 2 presents a DOS batch file that performs three tasks. First it compiles the TDMWebLib using the command-line compiler. Then it creates a type library for the TDMWebLib and registers it. Finally, it installs TDMWebLib in the global assembly cache.

The global assembly cache is a repository that stores the assemblies used by .NET clients. Obviously, the word 'cache' gives the game away; however, the benefits of using the global assembly cache are two-fold.

First, the CLR performs an integrity check on each and every assembly that a .NET client requests. Assemblies that are in the global assembly cache have their integrity checked before they are placed in the cache; therefore there is a performance gain to be had by placing assemblies in the cache.

The second benefit is versioning: the global assembly cache allows more than one version of the same assembly. This might be useful if you need to work with a 'release' assembly and a 'test' assembly at the same time.

Figure 7 demonstrates Listing 2 being executed: you can see that the first line compiles the C# files and uses the shared name key file that we created earlier.

➤ *Figure 7: Compiling the C# object; building a type library; adding the assembly to the global cache.*

```
D:\Dev\Itec\cs\TDMWEB~1>C

D:\Dev\Itec\cs\TDMWEB~1>csc /a.keyfile:TDMKey.snk /target:library TDMWebLib.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.


D:\Dev\Itec\cs\TDMWEB~1>regasm /tlb:TDMWebLib.tlb TDMWebLib.dll
RegAsm - .NET Assembly Registration Utility Version 1.0.2204.21
Copyright (C) Microsoft Corp. 2000.  All rights reserved.
Types registered successfully
Assembly exported to 'TDMWebLib.tlb' and registered successfully

D:\Dev\Itec\cs\TDMWEB~1>gacutil -i TDMWebLib.dll

Microsoft (R) .NET Global Assembly Cache Utility.  Version 1.0.2204.21
Copyright (C) Microsoft Corp. 1998-2000

Assembly successfully added to the cache
```

➤ *Listing 2: The .bat file that builds our C# assembly.*

```
csc /a.keyfile:TDMKey.snk /target:library TDMWebLib.cs
regasm /tlb:TDMWebLib.tlb TDMWebLib.dll
gacutil -i TDMWebLib.dll
```

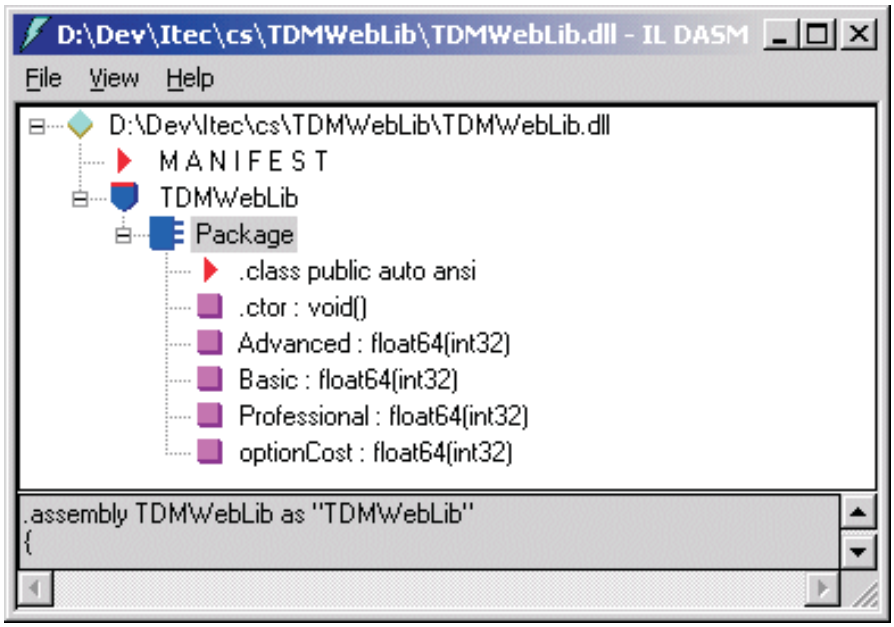
A Brief Look At IL

I am not going to profess to be an IL expert; however, I believe that the best way to learn is by hands-on exposure. The .NET Framework SDK installs an IL Disassembler (IL DASM) that allows us to peek inside .NET assemblies. Figure 8 presents a screenshot of the IL DASM tool in use: if you are comfortable with type libraries most of it should be familiar. Notice that there will have to be a type conversion between `integer` and `int32`, and between `double` and `float64`. It is the CCW that performs this task for us. Listing 3 presents the IL for the `Basic` method: whilst not rocket science, it does give you an idea of what is going on under the hood.

I am not going to dwell on an analysis of the IL code, but the basic flow of events is this:

➢ Load register 8 with the value 60: this is the cost of a basic TDMWeb package.

➤ *Figure 8: Viewing the metadata for the TDMWebLib object.*



- Call the method `optionCost` to obtain a cost for the options.
- Add the cost for the options to the package cost.

Importing The TDMWebLib Type Library

We used the regasm.exe tool to build a type library for the TDMWebLib object. Regasm.exe also registers the type library for us. Importing the type library for use in Delphi is the simple matter of choosing the `Import Type Library` option from the `Project` menu, as shown in Figure 9. Scrolling through the list of available type libraries, we should find the TDMWebLib (Version 0.1) type library. Clicking on the `Create Unit` button creates the source file TDMWebLib_TLB.pas, which can then be used in an unmanaged environment. Listing 4 presents a fragment of the Pascal file that is created: the function names should look familiar.

Once the type library has been imported, you will probably have noticed that, in addition to our three functions, `Basic`, `Advanced` and `Professional`, we also have `Equals`, `GetHashCode` and `GetType`. There is also a property: `ToString`.

➤ *Figure 9: Importing TDMWebLib for use in Delphi.*

```
.method public hidebysig instance float64
  Basic(int32 iOption) il managed
{
  // Code size       22 (0x16)
  .maxstack  3
  .locals (float64 V_0)
  IL_0000:  ldc.r8     60.
  IL_0009:  ldarg.0
  IL_000a:  ldarg.1
  IL_000b:  call       instance float64 TDMWebLib.Package::optionCost(int32)
  IL_0010:  add
  IL_0011:  stloc.0
  IL_0012:  br.s       IL_0014
  IL_0014:  ldloc.0
  IL_0015:  ret
} // end of method Package::Basic
```

➤ *Listing 3: IL for the Basic managed method.*

```
_Package = interface(IDispatch)
  ['{F9A70A0E-8A4D-325D-BBB9-D786F1FBA61C}']
  function  Get_ToString: WideString; safecall;
  function  Equals(obj: OleVariant): WordBool; safecall;
  function  GetHashCode: Integer; safecall;
  function  GetType: _Type; safecall;
  function  Basic(iOption: Integer): Double; safecall;
  function  Advanced(iOption: Integer): Double; safecall;
  function  Professional(iOption: Integer): Double; safecall;
  property ToString: WideString read Get_ToString;
end;
```

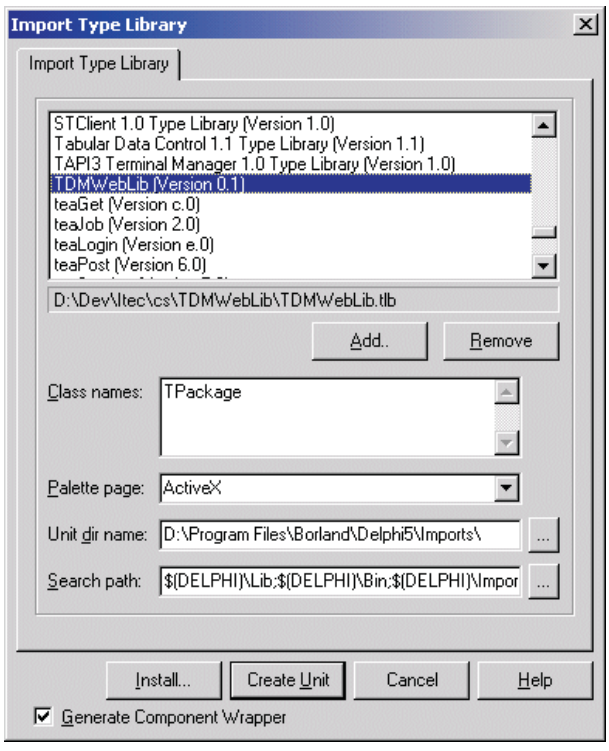
➤ *Listing 4: A snippet of the Pascal type library that is created.*

These methods and properties are part of the .NET base class `System.Object`. The CLR enforces a Common Type System, thus ensuring that each and every managed class inherits from `System.Object`. It is not too dissimilar to Delphi's TObject.

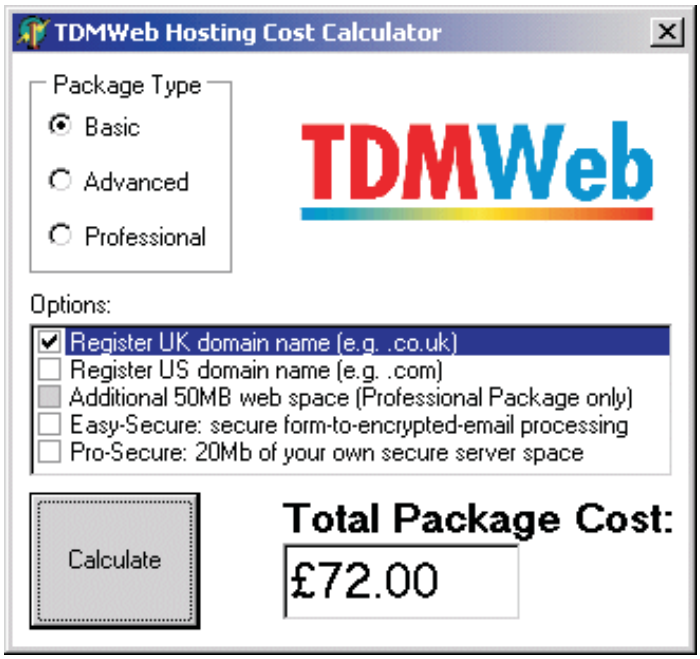
Figure 1 depicted Execution Support as part of the CLR. If you take a closer look at the type libraries that are created, you will notice some reference to MSCorEE.dll. This is the execution engine that is responsible for creating the CCW for the TDMWebLib C# object.

Putting It All Together

Now that we have a compiled C# object and the associated type library, we need an application to allow us to try out our venture in the managed world of C#. Listing 5 presents the Delphi code required



➤ *Figure 10: Why pay an arm and a leg? [Especially as it's now even cheaper... Ed]*



to do just that. Figure 10 presents the form that will be used to test our logic. There is some trivial client-side (Delphi) logic that ensures that Option 3 is only available if the Professional package type is selected but, apart from that, the client-side processing is virtually nil: the hard work is being performed by the CLR (it creates a CCW) and our C# object (TDMWebLib). The beauty of this approach will be realised when TDMWeb drops its prices (again!): simply modify the TDMWebLib, recompile, re-register the assembly, and hey presto: all of the applications that rely on the TDMWebLib object will start to use the new prices.

This example uses early binding: obviously this brings with it compile-time benefits of code completion and type safety. However, if you must use late binding, then the following code snippet should point you in the right direction:

```
Uses
  ComObj;
..
var obj : OleVariant;
...
obj    := CreateOleObject(
  'TDMWebLib.Package');
dCost := obj.Basic(1);
```

➤ *Listing 5: Testing our C# class using early binding.*

From Delphi To .NET

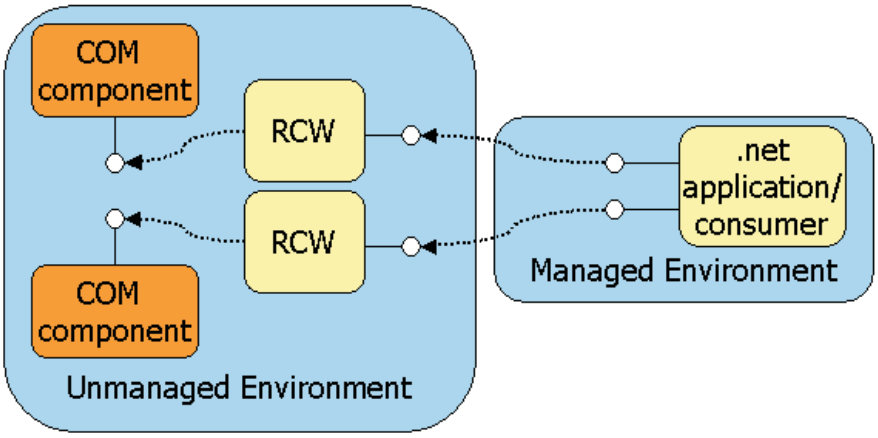
I have demonstrated that we can use managed .NET components in unmanaged environments via the 'bridge' that is the CCW. However, bridges are for travelling in two directions (unless you are in London!), therefore it must be possible to use unmanaged components in a managed environment.

Not surprisingly, you can. The new acronym for this is RCW: the Runtime Callable Wrapper. Figure 11 graphically explains the architecture that the RCW permits.

The RCW, like the CCW, is an arbitrator that is capable of performing some reasonably complicated marshalling 'out of the box'. For instance, the RCW knows how to seamlessly cast managed data types into unmanaged data types. It is able, to take just one example, to convert a `BSTR` into a `String` type on the fly.

The VS.NET IDE offers type library import functionality that is similar to Delphi's `Import Type Library` menu option. Traditionally, VS.NET uses the `Add Reference` menu option from the `Project` menu. It is perfectly possible to add a reference to our Delphi COM object using the `Add Reference` dialog; however, because we may wish to use the command-line C# compiler, it is best that we create a managed type library first. Creating the managed type library requires the use of another command-line utility: tlbimp.exe. Listing 6 presents the Delphi implementation of TDMWebLib. Figure 12 demonstrates tlbimp.exe in use: essentially we are taking our unmanaged Delphi DLL (see

➤ *Figure 11: Managed meets unmanaged: the CLR creates a RCW for each COM object.*




```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, CheckLst, ExtCtrls;
type
  TForm1 = class(TForm)
    Image1: TImage;
    rgPackage: TRadioGroup;
    clbOptions: TCheckListBox;
    Label1: TLabel;
    btnCalculate: TButton;
    edtCost: TEdit;
    Label2: TLabel;
    procedure btnCalculateClick(Sender: TObject);
    procedure rgPackageClick(Sender: TObject);
  private
  public
  end;
var
  Form1: TForm1;
implementation
uses
  TDMWebLib_TLB;
{$R *.DFM}
procedure TForm1.btnCalculateClick(Sender: TObject);
var
  obj : _Package;
  cost : double;
  i,options : integer;
begin
  // Create an instance of the C# TDMWebLib object...
  obj := CoPackage.Create;
  options:=0;
  // figure out which options are set...
  for i:=0 to clbOptions.Items.Count - 1 do
    if clbOptions.checked[i] then
      options := (1 SHL i) + options;
  // Simple validation, and we're off...
  if (options > 0) then
    case rgPackage.ItemIndex of
      0: cost := obj.Basic(options);
      1: cost := obj.Advanced(options);
      2: cost := obj.Professional(options);
    else
      ShowMessage('You must select a Package!');
    end
  else
    ShowMessage('You must select at least one option!');
  edtCost.text:=Format('%m',[cost]);
end;
procedure TForm1.rgPackageClick(Sender: TObject);
var i : integer;
begin
  edtCost.Text:='';
  // Clear the currently selected options...
  for i:=0 to clbOptions.items.count - 1 do
    clbOptions.Checked[i]:=false;
  // 50MB is only available for the Professional Package...
  clbOptions.ItemEnabled[2]:=
    (Sender as TRadioGroup).ItemIndex = 2;
end;
end.
```

```
unit Package;
interface
uses
  ComObj, ActiveX, TDMWebLib_TLB, StdVcl;
type
  TPackage = class(TAutoObject, IPackage)
  protected
    function Basic(iOption: Integer): Double; safecall;
    function Advanced(iOption: Integer): Double; safecall;
    function Professional(iOption: Integer): Double;
      safecall;
  end;
implementation
uses
  ComServ;
function optionCost(iOption : Integer) : Double;
var
  lCost : Double;
begin
  lCost := 0;
    if ((iOption AND 1) > 0) then  lCost := lCost + 12;
    if ((iOption AND 2) > 0) then  lCost := lCost + 30;
    if ((iOption AND 4) > 0) then  lCost := lCost + 30;
    if ((iOption AND 8) > 0) then  lCost := lCost + 30;
    if ((iOption AND 16) > 0) then lCost := lCost + 100;
  Result := lCost;
end;
function TPackage.Basic(iOption: Integer): Double;
begin
  Result := 60 + optionCost(iOption);
end;
function TPackage.Advanced(iOption: Integer): Double;
begin
  Result := 100 + optionCost(iOption);
end;
function TPackage.Professional(iOption: Integer): Double;
begin
  Result := 150 + optionCost(iOption);
end;
initialization
  TAutoObjectFactory.Create(ComServer, TPackage,
Class_Package,
    ciMultiInstance, tmApartment);
end.
```

➤ *Listing 6: It works both ways.*

```
D:\Dev\Itec\cs\Delphi>TLBIMP TDMWEBLIB.DLL /OUT:TDMWEBLIB_MGD.DLL
TlbImp - TypeLib to .NET Assembly Converter Version 1.0.2204.21
Copyright (C) Microsoft Corp. 2000.  All rights reserved.
Typelib imported successfully to TDMWEBLIB_MGD.DLL
```

➤ *Figure 12: Building a type library for use in a managed environment.*

➤ *Listing 7: C# front-end that uses the Delphi pricing calculator.*

```
namespace WinApp
{
  using System;
  using System.Drawing;
  using System.Collections;
  using System.ComponentModel;
  using System.WinForms;
  using System.Data;
  /// <summary>
  ///    Summary description for Form1.
  /// </summary>
  public class Form1 : System.WinForms.Form
  {
    /// <summary>
    ///    Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components;
      private System.WinForms.Label label2;
      private System.WinForms.CheckedListBox clbOptions;
      private System.WinForms.Label label1;
      private System.WinForms.TextBox edtCost;
      private System.WinForms.RadioButton rbBasic;
      private System.WinForms.RadioButton rbProfessional;
      private System.WinForms.RadioButton rbAdvanced;
      private System.WinForms.GroupBox groupBox1;
      private System.WinForms.Button btnCalculate;
    public Form1()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();
      //
      // TODO: Add any constructor code after
      // InitializeComponent call
      //
    }
    /// <summary>
    ///    Clean up any resources being used.
    /// </summary>
    public override void Dispose()
    {
      base.Dispose();
      components.Dispose();
    }
    /// <summary>
    ///    Required method for Designer support - do not
    ///    modify the contents of this method with the code
    ///    editor.
    /// </summary>
    private void InitializeComponent()
      {
        // Removed for conciseness; it's on the companion
        // disk that accompanies this magazine
      }
      protected void rbAdvanced_CheckedChanged (object
        sender, System.EventArgs e)
      {
        clbOptions.SetItemCheckState(2,
          CheckState.Indeterminate);
      }
      protected void rbBasic_CheckedChanged (object sender,
        System.EventArgs e)
      {
        clbOptions.SetItemCheckState(2,
          CheckState.Indeterminate);
      }
      protected void rbProfessional_CheckedChanged (object
        sender, System.EventArgs e)
      {
        clbOptions.SetItemCheckState(2,
          CheckState.Unchecked);
      }
      protected void Form1_Activated (object sender,
        System.EventArgs e)
      {
        rbBasic.Checked = true;
        clbOptions.SetItemCheckState(2,
          CheckState.Unchecked);
      }
      protected void clbOptions_SelectedIndexChanged (object
        sender, System.EventArgs e)
      {
        if ((clbOptions.SelectedIndex == 2) &&
          (!rbProfessional.Checked))  {
          clbOptions.SetItemCheckState(2,
            CheckState.Indeterminate);
        }
      }
      protected void btnCalculate_Click (object sender,
        System.EventArgs e)
      {
        // Create an instance of the Delphi TDMWebLib
        // package calculator
        TDMWebLib.Package obj = new TDMWebLib.Package();
        double dCost = 0;
        int i, iOptions = 0;
        for (i=0;i<clbOptions.Items.Count;i++) {
          if (clbOptions.GetItemChecked(i) &&
            (clbOptions.GetItemCheckState(i)!=
            CheckState.Indeterminate)) {
            iOptions = (1 << i) + iOptions;
          }
        }
        if (rbBasic.Checked)  {
          dCost = obj.Basic(iOptions);
        }
        else
        if (rbAdvanced.Checked) {
          dCost = obj.Advanced(iOptions);
        }
        else
        if (rbProfessional.Checked) {
          dCost = obj.Professional(iOptions);
        }
        edtCost.Text = '£' + dCost.ToString();
      }
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    public static void Main(string[] args)
    {
      Application.Run(new Form1());
    }
  }
}
```

Listing 6) and are creating a DLL that can be used in a managed environment. Going back to VS.NET, it is TDMWEBLIB_MGD.DLL that I chose to import for this example (via the `Browse` option). The VS.NET `Add Reference` IDE is fairly intuitive, so I shall refrain from presenting a screenshot.

Now that we have a reference to our managed DLL, what does it give us? Figure 13 presents the VS.NET Object Browser. We can see that our three TDMWebLib methods are available for use. Listing 7 puts the methods to good use: it mimics the operation of the Delphi equivalent (Listing 5). Like Delphi, VS.NET offers the usual benefits gained by early binding, most notably code completion, as is demonstrated by Figure 14. Figure 15 presents a screenshot of the C# version of Figure 10, albeit slightly less polished.

The C# code that actually uses the Delphi options costing methods is nothing more than a few lines. As you would expect, we have to create a new instance of the TDMWebLib Package and then

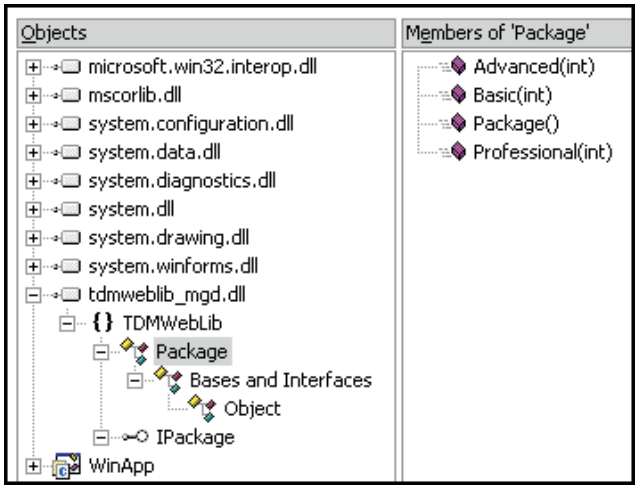


Figure 13: Looking at our Delphi object in VS.NET's Object Browser.

Figure 14: Early binding gives us code completion in VS.NET.

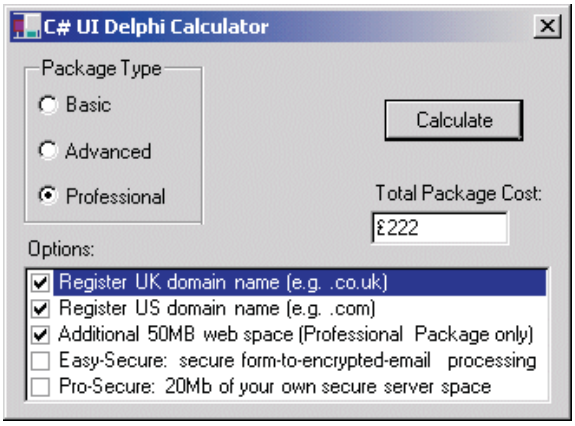
```
TDMWebLib.Package obj = new TDMWebLib.Package();

dCost = obj.
```

Glossary

Assembly: A collection of functionality built, versioned, and deployed as a single implementation unit (one or multiple files). An assembly is the primary building block of a .NET application. All managed types and resources are marked either as accessible only within their implementation unit or as exported for use by code outside that unit. In the runtime, the assembly establishes the name scope for resolving requests and the visibility boundaries are enforced. The runtime can determine and locate the assembly for any running object because every type is loaded in the context of an assembly.

Assembly cache: A machine-wide code cache used for side-by-side storage of assemblies. There are two parts to the cache: the global assembly cache contains assemblies that are explicitly installed to be shared among many applications on the computer; the download cache stores code downloaded from internet or intranet sites, isolated to the application that triggered the download so that code downloaded on behalf of one application/page does not impact other applications.

Metadata: Information that describes every element managed by the runtime: an assembly, loadable file, type, method, and so on. This can include information required for debugging and garbage collection, as well as security attributes, marshalling data, extended class and member definitions, version binding, and other information required by the runtime.



➤ *Figure 15: Calling the Delphi options calculator from a C# application.*

execute the methods, as shown in this snippet:

```
TDMWebLib.Package obj =
  new TDMWebLib.Package();
...
dCost = obj.Basic(iOptions)
```

So, two-way interoperability is achievable: our Delphi code base is safe and we do not need to re-write it all at once.

Summary

Over the course of this article we have essentially mixed two development environments. Currently, we are all developing for the same environment. We use competing tools, whether it is Delphi and C++Builder on the one hand, or Visual Basic and Visual C++ on the other. With the advent of .NET, things are about to change.

We cannot just switch off development for unmanaged environments: progression to the managed world must be gradual. Existing Delphi/C++Builder code is not going to vanish overnight, therefore interoperability is vitally important. Interoperability brings with it the advantage of language independence: the 'my language is better then your language' war is over. Your language might well be better than mine (perhaps improved readability over tightness of code?). However, at the end of the day, the languages achieve the same net result *[or should that be .NET result? Ed]*.

At the time of writing there are 17 development languages lined up for inclusion in the .NET platform. In the future, multi-language projects will become a reality. No longer will single-language projects be the norm. There is no reason why 'skills' and 'language experience' should dictate the direction a project takes. For example: why should a developer with cost-estimating skills have to learn VB.NET just to fit in with the other team members? .NET languages allow the cost-estimator to work in whatever language he or she is familiar with. Language integration allows C# and VB.NET (for example) to be used together in a team project. Thus, teams are brought together for their business skills instead of the traditional approach where teams comprise of developers with specific programming language skills.

Given the profusion of Delphi developers in the world today, let's hope that Borland is looking at how we can use some or all of these skills in the .NET environment.

Resources

Useful URLs

Deep Inside C#: An Interview with Microsoft Chief Architect Anders Hejlsberg: http://windows.oreilly.com/news/hejlsberg_0800.html

C# Standardisation: http://msdn.microsoft.com/net/ecma

Language comparison: http://genamics.com/developer/ csharp_comparative.htm

Useful Reading

Introducing .NET, James Conard et al, Wrox Press, ISBN 1-861004-89-3

C# Programming With the Public Beta, Burton Harvey et al, Wrox Press, ISBN 1-861004-87-7

Craig Murphy works as an Enterprise Developer for Currie & Brown (www.currieb.com) whose primary business is quantity surveying, cost management and project management. He can be contacted by email at Craig@isleofjura.demon.co.uk