

White Paper

Developing for Mono with Delphi Prism

Brian Long Consultancy & Training Services Ltd

December 2009

Corporate Headquarters

100 California Street, 12th Floor

San Francisco, California 94111

EMEA Headquarters

York House

18 York Road

Maidenhead, Berkshire

SL6 1SF, United Kingdom

Asia-Pacific Headquarters

L7. 313 La Trobe Street

Melbourne VIC 3000

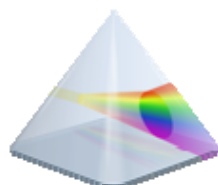
Australia

CONTENTS

Introduction	- 3 -
Microsoft .NET, ECMA and Mono	- 3 -
Cross-platform Development	- 4 -
Licensing Considerations	- 6 -
Getting Started	- 6 -
Compiler.....	- 6 -
Linux and OS X.....	- 6 -
Mono	- 7 -
Mono Source.....	- 7 -
Xcode and Interface Builder	- 8 -
Shared Folder	- 8 -
First Foray with Console Applications.....	- 8 -
Platform & Runtime Identification	- 11 -
Application Deployment	- 14 -
Leave As Is	- 14 -
Change the Behavior of .exe Files.....	- 14 -
Scripts	- 14 -
Bundled Executables	- 15 -
Mac OS X Application Bundles.....	- 16 -
Data Access	- 17 -
The Problem of GUI Applications.....	- 20 -
GUI Toolkits	- 21 -
WinForms.....	- 21 -
Mac Application Icons.....	- 24 -
GTK#.....	- 25 -
Tweak the GTK# Project Code.....	- 27 -
GTK# Examples.....	- 28 -
Simple GTK# Example	- 28 -
Dialog Example.....	- 30 -
TreeView Example	- 31 -
GTK# Bundled Executable	- 36 -
GTK# Mac OS X Application Bundle	- 38 -

Cocoa#	- 39 -
Monobjc	- 39 -
.nib Files	- 40 -
Interface Builder	- 41 -
Simple Text Editor	- 42 -
Monobjc and Snow Leopard	- 43 -
Correct Closure	- 44 -
Interacting Controls Examples	- 47 -
Color Chooser Example	- 51 -
Cocoa UI Techniques - Error Indication By Window Shake	- 55 -
Cocoa UI Techniques - Confirmation By Slide-in Sheet	- 56 -
Summary	- 58 -
Acknowledgements	- 58 -

INTRODUCTION



Delphi Prism is well known as the .NET Object Pascal compiler created by RemObjects Software and marketed by Embarcadero Technologies. The compiler at the heart of the product is RemObjects Oxygene, formerly RemObjects Chrome. Most commonly Delphi Prism is used by Delphi developers to build regular .NET applications of various types, including Windows applications (with the .NET WinForms library or the newer Windows Presentation Foundation, or WPF, library), console applications, windows service applications, ASP.NET web server applications or

Windows Communication Foundation (WCF) service libraries.

This paper leaves traditional .NET development to one side and instead explores how Delphi Prism lets you take your existing .NET and Delphi skills and break out of the Windows world by leveraging the Mono project. This allows you to build cross-platform applications that broaden your reach and gain access to users of Linux and Mac OS X. Launching into the world of cross-platform development will inevitably result in stumbling across some pitfalls, but we will try and foresee as many of the more common issues as we can and see how to avoid or overcome them.

The sample code shown in the paper will be made available on Code Central.

MICROSOFT .NET, ECMA AND MONO

Microsoft released .NET 1.0 in 2002 and it provided a great platform to develop for. Not only did the platform provide built-in security, consistent exception handling scheme, garbage collection, and ability to mix and match code from various languages in one managed runtime environment, but it held the promise of cross-platform development in a more tangible form than we had encountered before. After all, .NET applications were actually compiled to a common Intermediate Language (IL), which the platform translated to native instructions through a JIT (Just-In-Time compilation) process, thereby opening the door for various potential underlying hardware architectures.

As far as this Microsoft offering was concerned, their initial fulfillment of cross-platform support involved ensuring that .NET ran on Windows 98, Me, NT 4.0, 2000, and XP, and later versions of Windows as time went on. This definition of cross-platform being 'different implementations of Windows running on the same hardware' didn't really come as much of a treat – it was after all assumed to be a given by Windows developers. However, all was not lost.

Early in the .NET development cycle, Microsoft saw the potential of what they were working towards and released the specifications for the major parts of its .NET infrastructure to the European Computer Manufacturer's Association (ECMA) for standardization. The first standard of what was called the Common Language Infrastructure, or CLI, was released in December 2001. This later became an ISO standard in April 2003. You can download the latest standards document from the ECMA web site at <http://www.ecma-international.org/publications/standards/Ecma-335.htm>, and I would recommend doing so – it provides a formalized definition of the platform you develop for as a .NET developer. The standard is split into various sections, or partitions, and covers the architecture of the virtual

machine (or execution engine), metadata, the base class libraries (BCL), the Intermediate Language.

So Microsoft's .NET platform is an implementation of the CLI. Microsoft calls the virtual machine component of .NET the Common Language Runtime or CLR, and its .NET class library (FCL, or Framework Class Library) is a significant superset of the BCL, so .NET can be seen as an implementation of the CLI (CLR + BCL) plus a large additional set of libraries. Note that the CLI specification is also available as two annotated books from Addison-Wesley, split between the platform and the BCL: *The Common Language Infrastructure Annotated* by James S. Miller & Susann Ragsdale (Addison-Wesley, 2004) and *.NET Framework Standard Library Annotated Reference, Volume 1: Base Class Library and Extended Numerics Library* by Brad Abrams (Addison-Wesley, 2004).

Microsoft proceeded with additional implementations of the CLI, thereby actually making .NET a cross-platform product. The .NET Compact Framework for mobile and embedded devices emerged towards the end of 2002. Earlier the same year a shared source implementation of the CLI, SSCLI (codenamed Rotor) was released under an academic license. This huge source base shared a lot with the commercial .NET platform, but employed a Platform Abstraction Layer (PAL) to allow it to target Windows, FreeBSD or Mac OS X.



More interesting was the Mono project (<http://www.mono-project.com>), originally from Ximian, now from Novell. This project was started before .NET 1.0 was released and Mono 1.0 was released in 2004. Mono attempts to be not only a complete implementation of the CLI and also as much of the additional non-standardized parts of .NET as possible on as many platforms as possible. The JIT compiler targets include x86, x86-64, SPARC, ARM, and PowerPC and you can download installation kits for Windows, OS X and various Linux distros.

As well as matching what .NET offers, Mono goes above and beyond that and takes advantage of many Linux and OS X specific technologies. There are libraries that support specific Unix-related calls and much support for various graphical toolkits.

Delphi Prism supports the Mono platform and we will look at how we can embrace this support and move into the world of Linux and OS X without abandoning too much of what we know.

There are other implementations of the CLI available, including the DotGNU project's Portable.NET, but they fall outside the scope of this paper.

CROSS-PLATFORM DEVELOPMENT

Before diving into the code, let's have a think about what is meant by cross-platform development. This is important as there are various interpretations that could be made.

Regular .NET applications are already cross-platform as they will work on Microsoft Windows machines running on either 32-bit or 64-bit machines. That could be described as cross-architecture. Many .NET applications will actually run un-altered on the Mono platform. This could be described as cross-CLI. A given managed application might run under several operating systems. This could be described as cross-OS. And of course there are permutations

of these; you need to decide on your requirements and work towards solving them using the tools at your disposal. The thrust of this paper will be to promote code re-use (cross-CLI) and look at the feasibility of cross-OS code, which by default will be cross-architecture to one extent or another.

Of course, the moment you start contemplating having code that runs on any OS other than Windows (which in this context will be Linux or Mac OS X, both Unix-based), you need to be conscious of what will immediately break in existing Windows-centric code. There are obvious examples found when interacting with the file system, such as the separator between directories in a path (back slash on Windows and forward slash on Unix), or the separator between paths in a path list (semi-colon on Windows, colon on Unix). These are easy pitfalls to avoid if you conscientiously use the facilities in the BCL that shield you from the specific platform option, namely `System.IO.Path.DirectorySeparatorChar` and `System.IO.Path.PathSeparator`. Slightly more interesting is the fact that Unix-based file systems are typically case-sensitive. That one could take some thinking about if you do a lot of file manipulation.

The more knotty concerns crop up when thinking about the application's GUI and whether or not you are interested in maintaining consistency with the look and feel of the OS you are targeting. Can you create one UI and have it work everywhere, or should you instead think of building specific UIs for specific OSs? We'll look at the options later (see *GUI Toolkits* on page - 21 -).

And what about native code that you've called through the P/Invoke or COM Interop mechanisms? Well, P/Invoke is fully supported and so if you can find suitable implementations of the native libraries with suitable functions exported, you can probably conditionalise the code to call one function or another. A P/Invoke function won't be loaded by the interop layer in the runtime unless the method that calls it is invoked. So as long as you keep platform-specific P/Invoke calls wrapped up in their own functions, with different functions calling the native functions on different platforms, you can work through this one. The Mono class libraries call many native functions in order to work. Mono wraps up the Unix-specific P/Invokes in the `Mono.Unix.Native.Syscall` class in the `Mono.Posix.dll` assembly. Incidentally, you can get a list of all the P/Invoke calls contained in an assembly by using the Mono tool `monodis`, passing it the `--implmap` command-line switch.

COM Interop, however, is another matter, what with COM being a mostly Windows-specific technology. That said, Mono supports COM Interop to some extent on the Windows platform (see [http://mono-project.com/COM Interop](http://mono-project.com/COM%20Interop)).

You may be wondering whether it is feasible to work out programmatically whether you are running on Windows or on Linux or on OS X, or whether you can tell if you are running under Mono or .NET. It is indeed possible to do all these things as we shall see when start building applications. However, conditionalising your code based on CLI is considered poor form, although it is accepted that sometimes it is necessary, not least of which for when you happen upon a bug or limitation in Mono. Do remember that Mono is under constant development and is always playing a game of catch-up with .NET. It is in very good shape, but there are still holes here and there.

If you have a library of code that you are thinking of taking to Mono and you want to see how well it is likely to fare in terms of how many of the .NET library calls are implemented on Mono

then you should use the MoMA (Mono Migration Analyzer) tool (<http://www.mono-project.com/MoMA>). This scans an assembly and produces a report based on some definition files that are generated for specific releases of Mono.

LICENSING CONSIDERATIONS

An important factor to take into consideration when building applications with open source libraries such as Mono, the Mono tools, GTK# and Monobjc is the license model they use and subsequent licensing implications for the whole project.

The Mono tools work under the “viral” GPL v2 (GNU General Public License, <http://www.opensource.org/licenses/gpl-license.html>) license, which means if you incorporate them into your projects, your projects have to use the GPL (meaning it has to be open software). Fortunately this ends up as irrelevant; you just need to use the odd Mono tool during development - you are unlikely to need to include them in your own software.

The Mono runtime and the Monobjc libraries are licensed under the LGPL v2 (GNU Library General Public License, <http://www.gnu.org/copyleft/library.html>), which means you can use them in your projects without being forced to make your project use the GPL.

The Mono class libraries are licensed under the MIT X11 license (<http://www.opensource.org/licenses/mit-license.html>), which is designed to be rather like the LGPL, but avoid technical loopholes where you might have been forced to make your application LGPL just by using it and inheriting from classes within the framework.

In summary, if you use any of these Mono-based libraries in your project, you can distribute them without being forced to make your own applications use the LGPL - you can use whatever license you choose. However the author is by no means a lawyer and you should clearly invest some personal research in this area to ensure you do not get caught out.

GETTING STARTED

Ok, enough of the preamble. Let's make sure you have all the tools you'll need for our journey beyond Windows.

COMPILER

Firstly, the compiler - well that's Delphi Prism running in Visual Studio (or from a command-line, running either under .NET or under Mono if you prefer). In 2010 it will also support running within the Mono development tool MonoDevelop, but for now we'll stick with Visual Studio.

LINUX AND OS X

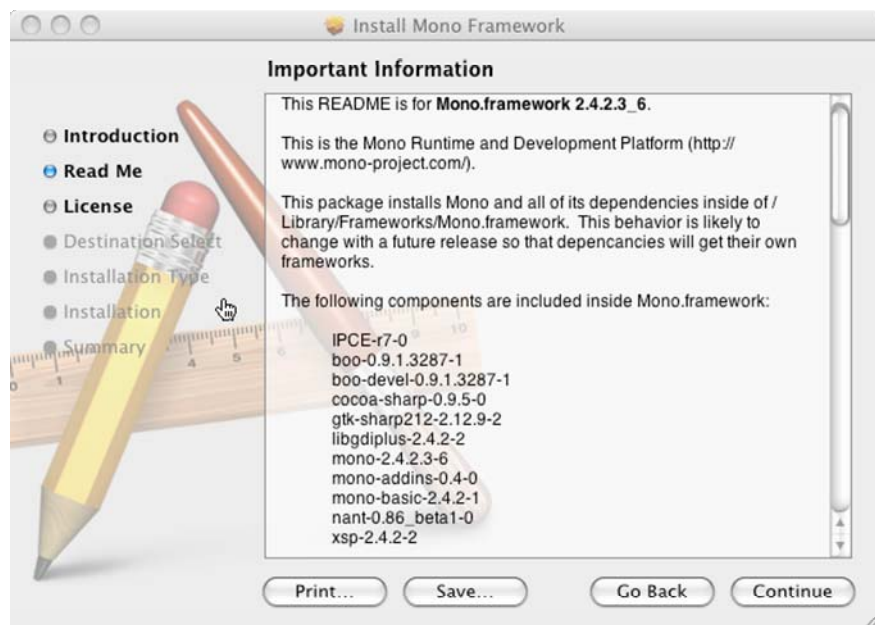
Now, assuming your cross-platform requirement isn't simply restricted to running under a different CLI implementation then you'll either be running an Apple Mac with OS X, and/or have Linux running somewhere. For the examples here I am using the latest (at the time of writing) version of OS X, 10.6 (or Snow Leopard). I am also running the Debian-based Linux distro, Ubuntu. Most of the work was done with the April 2009 release of Ubuntu (version 9.04 aka Jaunty Jackalope).

MONO

When Delphi Prism is installed it also installs a copy of Mono on Windows, currently version 2.4 (installed into *C:\Program Files\Mono-2.4*).

Ubuntu comes with the core parts of Mono pre-installed. Ubuntu 9.04 comes with version 2.0.1 of Mono installed in */usr/lib/mono* and */usr/bin*. Ubuntu 9.10 includes Mono 2.4.2.3. For some Linux distros you can find packages to download at <http://www.go-mono.com/mono-downloads>, and for others the distro will already have it installed, as Ubuntu does. The fact that Ubuntu only has the core Mono libraries installed will come to bite us later, but we'll see how to resolve the problems we see. In the case of other distros, there may be similar problems, but then there will also be similar solutions to research.

OS X does not come with Mono pre-installed so the appropriate version needs to be pulled down from Mono downloads link above - there are versions for both Intel and PowerPC architectures there. On the Mac it installs into */Library/Frameworks/Mono.framework* and */usr/bin*. The Mac installer looks like this:



MONO SOURCE

It's certainly not a requirement in order to work with Mono, but it should be noted that you can pull down the source code for as many parts of Mono as you like. This can be quite educational. I prefer to use Subversion and check-out the source directly from the SVN tree as documented at http://www.mono-project.com/Compiling_Mono_From_SVN so for the Mono main class library source this will pull down around 110,000 files (totaling slightly more than half a gigabyte) into a new directory called *mcs*:

```
svn co http://anonsvn.mono-project.com/source/trunk/mcs
```

XCODE AND INTERFACE BUILDER

If you are planning on developing for the Mac and ultimately want native looking applications, then you will have to install Apple's free Xcode development tools. These include a development environment (Xcode) and, more importantly, a UI design tool (Interface Builder). You can either find these as an Optional Install on the OS X installation DVD, or you can download the installer package from <http://developer.apple.com/tools/xcode> after registering for free ADC (Apple Developer Connection) membership.

Apple's recognizable user interface is produced by the Cocoa library. When you get to building applications that use Cocoa you will need to use Interface Builder from the Xcode tools. One of the reasons this paper has been prepared using the latest version of OS X (10.6) is that the last few versions of Interface Builder have each had some major work done to them and so they have things laid out in different places, or not at all. Using the latest version ensures you can all get the benefit of the latest changes.

SHARED FOLDER

The next thing to take into consideration is how you will manage compiling a project on Windows and running it on either Linux or OS X. Some form of shared folder will be required. This can be a folder resident on the Linux or OS X box shared via the network, or via the Shared Folders mechanism if you are running Windows in VM software on the non-Windows box. Alternatively it can be a Windows shared folder that is mounted onto the Linux/Mac file system.

In my case, I used a MacPro as the real machine, and used VMWare to permit Windows XP and Ubuntu Linux to run simultaneously in virtual machines. A subdirectory off my OS X home directory was shared to both VMs so the Windows VM could generate applications on the Mac drive and the Linux VM could see them as well.

FIRST FORAY WITH CONSOLE APPLICATIONS

Okay, the time has come to cut some code. The first thing to do is just prove a point. We'll build a .NET application and ensure it runs under .NET and Mono. We'll then build the same project as a Mono project and prove that it also works under both platforms. We'll start with a simple console application that lists out various pieces of environment information. Note that when you choose File, New, Project... (Ctrl+Shift+N) and look at the available projects there is a Console Application in the main Delphi Prism section, and there is also a Mono Console Application in the Mono section. You can choose either (and indeed can go through both).

The main source of each version of the project looks like this:

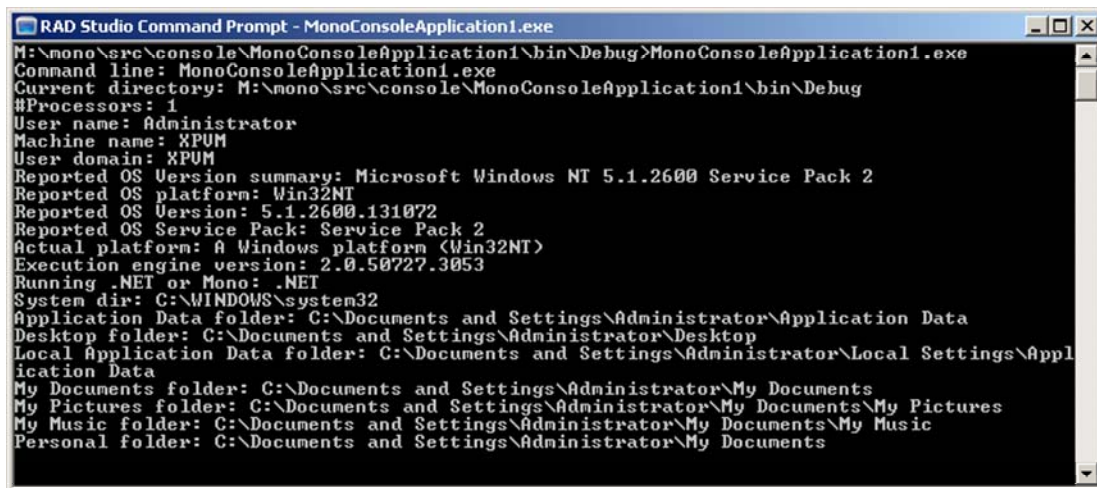
```
class method ConsoleApp.Main;
begin
  Console.WriteLine(String.Format('Command line: {0}', Environment.CommandLine));
  Console.WriteLine(String.Format('Current directory: {0}',
    Environment.CurrentDirectory));
  Console.WriteLine(String.Format('#Processors: {0}', Environment.ProcessorCount));
  Console.WriteLine(String.Format('User name: {0}', Environment.UserName));
  Console.WriteLine(String.Format('Machine name: {0}', Environment.MachineName));
  Console.WriteLine(String.Format('User domain: {0}', Environment.UserDomainName));
  Console.WriteLine(String.Format('Reported OS Version summary: {0}',
    Environment.OSVersion.VersionString));
```

```

Console.WriteLine(String.Format('Reported OS platform: {0}',
    Environment.OSVersion.Platform));
Console.WriteLine(String.Format('Reported OS Version: {0}',
    Environment.OSVersion.Version));
Console.WriteLine(String.Format('Reported OS Service Pack: {0}',
    Environment.OSVersion.ServicePack));
Console.Write('Actual platform: ');
if IsWindows then
    Console.WriteLine(String.Format('A Windows platform ({0})',
        Environment.OSVersion.Platform))
else if IsLinux then
    Console.WriteLine('Linux')
else
    Console.WriteLine('Mac OS X');
Console.WriteLine(String.Format('Execution engine version: {0}',
    Environment.Version));
Console.Write('Running .NET or Mono: ');
if IsMono then
    Console.WriteLine('Mono')
else
    Console.WriteLine('.NET');
Console.WriteLine(String.Format('System dir: {0}', Environment.SystemDirectory));
Console.WriteLine(String.Format('Application Data folder: {0}',
    Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData)));
Console.WriteLine(String.Format('Desktop folder: {0}',
    Environment.GetFolderPath(Environment.SpecialFolder.Desktop)));
// More of the same sorts of calls...
Console.ReadLine;
end;

```

The code uses some helper methods but we'll come back to those; they allow you to establish the runtime or OS for when you desperately need it. For now just note that when you compile either the .NET or the Mono version of the project you essentially get the same result - a managed Portable Executable file that can be run wherever Mono or .NET is found. For example, here is the Mono version of the project being run under .NET.

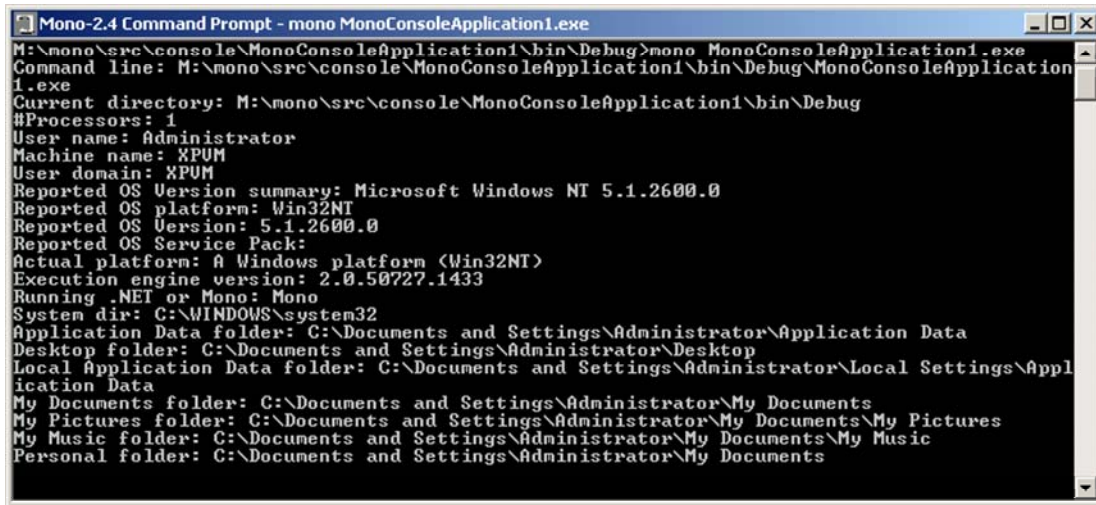


```

RAD Studio Command Prompt - MonoConsoleApplication1.exe
M:\mono\src\console\MonoConsoleApplication1\bin\Debug>MonoConsoleApplication1.exe
Command line: MonoConsoleApplication1.exe
Current directory: M:\mono\src\console\MonoConsoleApplication1\bin\Debug
#Processors: 1
User name: Administrator
Machine name: XPUM
User domain: XPUM
Reported OS Version summary: Microsoft Windows NT 5.1.2600 Service Pack 2
Reported OS platform: Win32NT
Reported OS Version: 5.1.2600.131072
Reported OS Service Pack: Service Pack 2
Actual platform: A Windows platform (Win32NT)
Execution engine version: 2.0.50727.3053
Running .NET or Mono: .NET
System dir: C:\WINDOWS\system32
Application Data folder: C:\Documents and Settings\Administrator\Application Data
Desktop folder: C:\Documents and Settings\Administrator\Desktop
Local Application Data folder: C:\Documents and Settings\Administrator\Local Settings\Appl
ication Data
My Documents folder: C:\Documents and Settings\Administrator\My Documents
My Pictures folder: C:\Documents and Settings\Administrator\My Documents\My Pictures
My Music folder: C:\Documents and Settings\Administrator\My Documents\My Music
Personal folder: C:\Documents and Settings\Administrator\My Documents

```

In addition to the fact that the application helpfully reports it is running under .NET, you should also know that you are running under .NET because to run under Mono you need to feed the application the `mono` command as you can see here:

A screenshot of a Windows command prompt window titled "Mono-2.4 Command Prompt - mono MonoConsoleApplication1.exe". The window shows the output of running the application. The text displayed is as follows:

```
M:\mono\src\console\MonoConsoleApplication1\bin\Debug>mono MonoConsoleApplication1.exe
Command line: M:\mono\src\console\MonoConsoleApplication1\bin\Debug\MonoConsoleApplication1.exe
Current directory: M:\mono\src\console\MonoConsoleApplication1\bin\Debug
#Processors: 1
User name: Administrator
Machine name: XPUM
User domain: XPUM
Reported OS Version summary: Microsoft Windows NT 5.1.2600.0
Reported OS platform: Win32NT
Reported OS Version: 5.1.2600.0
Reported OS Service Pack:
Actual platform: A Windows platform (Win32NT)
Execution engine version: 2.0.50727.1433
Running .NET or Mono: Mono
System dir: C:\WINDOWS\system32
Application Data folder: C:\Documents and Settings\Administrator\Application Data
Desktop folder: C:\Documents and Settings\Administrator\Desktop
Local Application Data folder: C:\Documents and Settings\Administrator\Local Settings\Application Data
My Documents folder: C:\Documents and Settings\Administrator\My Documents
My Pictures folder: C:\Documents and Settings\Administrator\My Documents\My Pictures
My Music folder: C:\Documents and Settings\Administrator\My Documents\My Music
Personal folder: C:\Documents and Settings\Administrator\My Documents
```

Note: on Windows, the Mono *bin* directory is not forced onto the system path. You can either remedy this or use the Mono Command Prompt in the Mono folder in the Start menu.

Regardless of runtime (CLI) the application dutifully reports details of the Windows build. Interestingly, Mono reports its execution engine (virtual machine) version as being exactly the same as the .NET CLR, presumably for compatibility.

Here's the application running on Linux. The platform version here is the Linux kernel version number. Notice the *Personal* or *My Documents* directory - that home directory is typically abbreviated to `~` when working at the command prompt and writing scripts in Unix-based OSs.



```
brian@ubuntu: ~/dev/mono/src/console/MonoConsoleApplication1/bin/Debug
File Edit View Terminal Help
brian@ubuntu:~/dev/mono/src/console/MonoConsoleApplication1/bin/Debug$ mono MonoConsoleApplication1.exe
Command line: /mnt/hgfs/dev/mono/src/console/MonoConsoleApplication1/bin/Debug/MonoConsoleApplication1.exe
Current directory: /mnt/hgfs/dev/mono/src/console/MonoConsoleApplication1/bin/Debug
#Processors: 1
User name: brian
Machine name: ubuntu
User domain: ubuntu
Reported OS Version summary: Unix 2.6.28.16
Reported OS platform: Unix
Reported OS Version: 2.6.28.16
Reported OS Service Pack:
Actual platform: Linux
Execution engine version: 2.0.50727.42
Running .NET or Mono: Mono
System dir:
Application Data folder: /home/brian/.config
Desktop folder: /home/brian/Desktop
Local Application Data folder: /home/brian/.local/share
My Documents folder: /home/brian
My Pictures folder: /home/brian/Pictures
My Music folder: /home/brian/Music
Personal folder: /home/brian
```

And here it is running on OS X, so you can see how the directory structures differ.



```
Terminal — mono — 109x23
MacPro:Debug brian$ mono MonoConsoleApplication1.exe
Command line: /Users/brian/dev/mono/src/console/MonoConsoleApplication1/bin/Debug/MonoConsoleApplication1.exe
Current directory: /Users/brian/dev/mono/src/console/MonoConsoleApplication1/bin/Debug
#Processors: 4
User name: brian
Machine name: MacPro.local
User domain: MacPro.local
Reported OS Version summary: Unix 10.0.0.0
Reported OS platform: Unix
Reported OS Version: 10.0.0.0
Reported OS Service Pack:
Actual platform: Mac OS X
Execution engine version: 2.0.50727.1433
Running .NET or Mono: Mono
System dir:
Application Data folder: /Users/brian/.config
Desktop folder: /Users/brian/Desktop
Local Application Data folder: /Users/brian/.local/share
My Documents folder: /Users/brian
My Pictures folder: /Users/brian/Pictures
My Music folder: /Users/brian/Music
Personal folder: /Users/brian
```

PLATFORM & RUNTIME IDENTIFICATION

Before moving on there is the matter of the helper functions used in the code to be addressed. They help identify runtime and OS. The runtime detection calls are straightforward - Mono's mscorlib.dll will always have a `Mono.Runtime` type in it, so:

```
class method ConsoleApp.IsMono: Boolean;
begin
    exit &Type.GetType( 'Mono.Runtime' ) <> nil
end;
```

```

class method ConsoleApp.IsDotNet: Boolean;
begin
    exit not IsMono()
end;

```

Note: This code uses a Delphi Prism extension to `Exit` where you can specify a function's return value and exit in one statement, rather like the `return` statement in C and C#. In this example its use over simply assigning to `Result` is irrelevant, however in later examples it proves useful when used amidst other logic - it saves having to assign a value to `Result` and then call `Exit` (or use other convoluted conditional logic to avoid further code executing).

For the platform routines, you might be forgiven for thinking a quick peek at `System.Environment.OSVersion.Platform` would sort that out - after all, the value is from the `System.PlatformID` enumeration which has all the platforms of interest in it. However some history is useful here. When .NET 1.0 came out, `PlatformID` only defined Windows-related values: `Win32NT`, `Win32S`, `Win32Windows` & `WinCE`. .NET 2 added the `Unix` value, and then .NET 3.5 added `MacOSX` and `Xbox`. This poses a problem for Mono on the Mac. As you can see above, on OS X it returns `Unix`; returning the correct `MacOSX` value (which was added to Mono) was considered but at the time it broke too much code.

Detecting a generic Unix platform is also an issue; the original Mono kept a true representation of `PlatformID` and so on Unix `Platform` returned an integer value 128. When .NET 2 added the `Unix` value (that had an integer value of 4) they switched to that, and now recently `MacOSX` (value 6) has been added.

So, the generally recommended way of determining if you are on a Windows or Unix platform is to check `System.IO.Path.DirectorySeparatorChar`, which will either be `'\'` or `'/'`. The approved way of working out what type of Unix you are on (Linux or OS X) is to take advantage of the Unix `uname` command. In a terminal window, `uname` will return either `'Darwin'` (the FreeBSD derivative that OS X is based on) or `'linux'`. How do you call such a command from your program? Well, running an external process is overkill - you would typically add a reference to `Mono.Posix.dll` and call `Mono.Unix.Native.Syscall.uname()`, being careful not to make the call on Windows:

```

class method ConsoleApp.IsWindows: Boolean;
begin
    Result := System.IO.Path.DirectorySeparatorChar = '\';
end;

class method ConsoleApp.IsLinux: Boolean;
begin
    if IsWindows then
        exit False;
    var buf: Mono.Unix.Native.Utsname;
    if Mono.Unix.Native.Syscall.uname(out buf) = 0 then
        exit string.Compare(buf.sysname, 'linux', True) = 0;
    Result := false;
end;

```

```

class method ConsoleApp.IsOSX: Boolean;
begin
    if IsWindows then
        exit False;
    var buf: Mono.Unix.Native.Utsname;
    if Mono.Unix.Native.Syscall.uname(out buf) = 0 then
        exit string.Compare(buf.sysname, 'darwin', True) = 0;
    Result := false;
end;

```

Note: The code above uses the Delphi Prism syntax extension to allow variables to be declared wherever they are required, rather than being forced to use the var section above a method's code block.

If you use any of the Mono libraries and you are expecting to deploy on Windows, particularly to .NET users without Mono, then you should ensure the Mono assembly references have their Copy Local option set to True, to get all dependent files copied to your *bin* directory. This allows you to readily see what needs to be deployed to users.

An alternative approach, just for completeness and to show that cross-platform code can make use of P/Invokes, involves talking directly to the Unix library libc.

```

ConsoleApp = class
private
    [DllImport('libc')]
    class method uname(buf: IntPtr): Integer; external;
    ...
end;
...
class method ConsoleApp.RunningOnUnix: Boolean;
begin
    // .NET 1.x didn't have a Unix value in System.PlatformID enum, so Mono
    // just used value 128.
    // .NET 2 added Unix to PlatformID, but with value 4
    // .NET 3.5 added MacOSX with a value of 6
    exit Integer(Environment.OSVersion.Platform) in [4, 6, 128];
end;

class method ConsoleApp.RunningOnLinux: Boolean;
begin
    exit RunningOnUnix and InternalRunningLinuxInsteadOfOSX
end;

class method ConsoleApp.RunningOnOSX: Boolean;
begin
    exit RunningOnUnix and not InternalRunningLinuxInsteadOfOSX
end;

class method ConsoleApp.InternalRunningLinuxInsteadOfOSX: Boolean;
begin
    // based on Mono cross-platform checking code in:
    // mcs\class\Managed.Windows.Forms\System.Windows.Forms\XplatUI.cs
    if not RunningOnUnix then
        raise new Exception('This is not a Unix platform!');
    var Buf: IntPtr := Marshal.AllocHGlobal(8192);
    try
        if uname(buf) <> 0 then
            // assume Linux of some sort
            exit True
    end;
end;

```

```

    else
        //Darwin is the Unix variant that OS X is based on
        exit Marshal.PtrToStringAnsi(Buf) <> 'Darwin'
    finally
        Marshal.FreeHGlobal(Buf);
    end;
end;

```

Note: the difference between a Mono project and a .NET project, as far as the Delphi Prism compiler is concerned, is really down to which set of assemblies to link against. In the case of the Mono project the project file contains a directive to point at the Mono framework assemblies.

APPLICATION DEPLOYMENT

One thing you will have noticed when running an application under the Mono runtime is the requirement to pass the executable name as a parameter to the `mono` command. This could well be a bit of an issue, given it is more work for a user than is normally required at a command-line. Fair enough, if the application ends up in an Applications menu somewhere and the command-line is obscured this maybe not so bad, but there is no such menu on a Mac. So let's look at the options you have for deploying applications and having them executed by the user.

LEAVE AS IS

The first option is to do nothing and require users to run your application via `mono`. This may be okay for some command-line tools, particularly if the recipient is familiar with using Mono applications and tools already, but it's not really very satisfactory.

CHANGE THE BEHAVIOR OF .EXE FILES

It is possible to set up a specific response in Linux when you invoke a file conforming to the Portable Executable file format (.exe files) and run them via `mono`, via the `binfmt` kernel module. However this is discouraged as it may interfere with other applications. For example VMWare Fusion sets up a similar mechanism to run .exe files in the context of a guest Windows system.

SCRIPTS

The next best thing would be to supply a shell script with your application whose sole purpose is to run `mono` and pass your application along. On Windows this would be a batch file (.bat) or command script (.cmd), although the issue is almost moot on Windows, given that users will most commonly launch applications from the folders in the Start menu. For command-line tools, however (such as those supplied with Mono itself) batch files are still appropriate.

On Unix systems shell scripts have no file extension. You might briefly consider writing such scripts from the comfort of Visual Studio's editor, but that wouldn't get you very far. Firstly there is the issue of line endings - Windows defaults to using carriage return and line feed characters, whereas Unix only uses line feeds. But more importantly you will need to test your scripts, so you should dive in and use a Unix text editor.

On a Mac, the faint-hearted will migrate towards the graphical TextEdit application, although there are various terminal-based editors: `ed`, `pico` (actually `nano`), `vi` (actually `vim`) and the

infamous emacs. Hardened Unix geeks will argue the merits of choosing either `vi` or `emacs`, but MacVim is a good GUI-based version of the old `vim` editor, available from <http://code.google.com/p/macvim>. You should also run `vimtutor` to get a handle on how to use it.

On Ubuntu, `ed`, `pico` (actually `nano`) and `vi` are installed by default. `vim`, `vimtutor` and `emacs` are available - run the command and you are told how to install them.

Given a Mono application called `Blah.exe`, a suitable, if minimal script, would look like the following, and perhaps be called simply `blah`, to match the general lower-case scheme found on the case-sensitive Unix systems:

```
#!/bin/sh
mono Blah.exe $@
```

The first line of the script is a shell execution directive and says the script should be run by `sh`, the basic shell (which on OS X maps onto `bash`, the Bourne-Again shell). The second line runs `mono`, passes the Mono application to it and the `$@` says to pass all the command-line parameters to the script along to the program.

Note: UK Mac keyboards don't have a `#` key on them. To type `#`, use `Alt+3`.

Note: In order for a shell script to be directly executed it must be marked as executable, otherwise you will get a *Permission denied* error. This is done by running the command: `chmod +x blah`

Alternatively you can run the script via the shell explicitly, for example: `sh blah`

Note: if your shared folder that you compile your projects to happens to be a Windows folder, mounted into the Linux or OS X file system, then you don't see the above problem. Since Windows file systems don't understand Unix execute bits all files are marked executable by default. This can be a handy time saver, but you should remember the underlying point about ensuring your scripts are marked as executable.

Note: If you're in the same directory as the script you won't be able to directly execute the shell by simply typing its name. Execution requires the script to be on the path and the current directory is not typically on the path in Unix-based systems (for security reasons) and so the path needs to be explicitly mentioned: `./blah`

Note: the Mono tools are managed applications and use scripts located in a directory on the path in order to be called with a single word.

BUNDLED EXECUTABLES

Another option is to bundle everything required for the application into a single executable. The Mono tool `mkbundle` does just this. It looks at the executable and identifies dependent assemblies (and their config files), assuming you pass the `--deps` command-line switch, and encodes them as raw assembly language files, which are then assembled into object files by the GNU assembler. It then creates a C file and compiles it all together into one self-contained application. The application still uses various native libraries when launched, but all the

managed assemblies are all compiled into the one image. We'll look at using this approach later (see page - 36 -).

Note that `mkbundle` does offer the `--static` option, which will statically link the Mono libraries into the target executable file, rather than dynamically linking to them. This will mean you will be forced to use an LGPL license for your application and so this option is generally best avoided. `mkbundle` does warn you of this issue if you use the `--static` switch by printing out this message: *Note that statically linking the LGPL Mono runtime has more licensing restrictions than dynamically linking. See <http://www.mono-project.com/Licensing> for details on licensing.*

MAC OS X APPLICATION BUNDLES

OS X offers another way of tidying away pesky additional resources required by an application: application bundles (or packages). If you've used a Mac and launched applications in Finder you will have seen that they appear to be singular monolithic entities, thereby making installing and uninstalling applications a breeze. However this appearance is an illusion; each application is a directory structure containing various files and resources as needed by the application.

In Finder you can right-click on an application and the context menu will have a *Show Package Contents* menu item. Choosing this launches another Finder window showing the directory structure within the application bundle, or package.

The actual directory that looks like a monolithic application has an `.app` extension to it, though Finder hides this fact. Beneath this is a *Contents* directory. In here is a *MacOS* directory containing the executable program, a *Resources* directory and a file called *Info.plist*. This latter item is a property list and describes the application bundle in a formalized manner. There may be other files as required in the bundle directory structure but that covers the main suspects. Note that application bundles provide the best means of setting an application icon, as displayed by the Dock and the Task Switcher; the icon file will be in the *Resources* directory and will be referenced by an entry in the property list file.

Clearly Finder is happy opening applications set up as bundles, but what about command-line access? Given an application set up as a bundle you can use the `open` command, which is essentially what Finder uses:

- if you know the application bundle directory name (such as `/Applications/iChat.app`) you can pass it as a parameter to `open`, for example:

```
open /Applications/iChat.app
```

- if the application is installed in the main `/Applications` directory and you know its name, such as MacVim (if installed), you can use the `-a` switch, which allows parameters to be passed, for example:

```
open -a MacVim ~/.bash_profile
```

Note: `open` also has the `-e` switch which will automatically open the passed in file with TextEdit.

When you start building OS X applications that have various support files, an application bundle is the natural choice. Indeed when you build applications based on Cocoa you are pretty much

forced into using bundles. To help us along, Mono has a tool called `macpack` that takes an assembly, various resources, and an icon file and generates an application bundle. The `-m` switch allows you to specify whether this app bundle is for a WinForms app, Cocoa app, X11 app or console app, as there are certain differences in what needs to be generated.

We'll come back to the subject of app bundles later (see page - 38 -).

DATA ACCESS

Mono supports ADO.NET functionality, despite it being outside the scope of the ECMA CLI specification. Over and above the standard ADO.NET namespaces there are also many more Mono-specific namespaces for various additional databases. See http://www.mono-project.com/Database_Access for full details. It's interesting to note that, for example, the `System.Data.SqlClient` namespace that provides Microsoft SQL Server support requires no native client as it is written in fully managed code.

Note: Mono includes a convenient database querying tool called SQL# (invoked with `sqlsharp`) that allows you to set a provider and connection string and test out query strings from a SQL prompt. It's not on Ubuntu by default but can be added readily by running: `sudo apt-get install mono-devel`. `sudo` is a command that allows you to run commands as the super-user, root.

A commonly used database is the open source MySQL (<http://www.mysql.com>) and you can access this in various ways. In regular .NET you might try the MySQL ODBC driver with code like this to run a SELECT query:

```
uses System.Data.Odbc;
...
class method ConsoleApp.GetSomeData(): string;
const
    rootPassword = '';
    //ODBC connection string
    connectionString = 'Driver={MySQL ODBC 3.51 Driver};Server=localhost;' +
        'User=root;Password=' + rootPassword + ';Option=3;';
    selectSQL = 'SELECT * FROM Customers WHERE Town LIKE '%che%'';';
    widths: Array of Integer = [3, 22, 12, 6];
var
    results: StringBuilder := new StringBuilder();
begin
    using sqlConnection: OdbcConnection := new OdbcConnection(connectionString) do
    begin
        sqlConnection.Open();
        SetupData(sqlConnection);
        var dataAdapter: OdbcDataAdapter := new OdbcDataAdapter(selectSQL, sqlConnection);
        var dataTable: DataTable := new DataTable('Results');
        dataAdapter.Fill(dataTable);
        for each row: DataRow in dataTable.Rows do
        begin
            results.Append('|');
            for I: Integer := 0 to dataTable.Columns.Count - 1 do
            begin
                var Width := 20;
                if I <= High(widths) then
                    Width := widths[I];
                results.AppendFormat('{0,' + Width.ToString() + '}|', row.Item[I]);
```

```

    end;
    results.Append(Environment.NewLine);
end;
TearDownData(sqlConnection);
end;
Result := results.ToString()
end;

```

This works fine in .NET and gives identical results in Mono running on Windows:

```

Z:\dev\mono\src\console\ConsoleDBApplication\bin\Debug>ConsoleDBApplication.exe
1|      John Smith| Manchester| 0001|
2|      John Doe| Dorchester| 0002|
3|      Fred Bloggs| Winchester| 0003|
4|    Walter P. Jabsco| Ilchester| 0004|
5|      Jane Smith| Silchester| 0005|
6| Raymond Luxury-Yacht| Colchester| 0006|

Press Enter to continue...

```

Note: The code above uses the Delphi Prism `for` each iteration syntax using a variable `row` that is local to the loop, and is declared in the loop syntax itself.

Note: Rather than force a particular character sequence for the end of each row, the platform-suitable character sequence is obtained from `System.Environment.NewLine`.

So the code works, but ODBC is a technology prevalent on Microsoft operating systems and doesn't show up that much elsewhere. Mono supports talking to ODBC, if present, but what happens on non-Windows platforms? The answer is you get an error *DllNotFoundException: libodbc.so* on Linux and the somewhat impenetrable *OdbcException: ERROR [1]* on OS X.

You can get ODBC to work on Unix-based platforms with help from the unixODBC project (<http://www.unixodbc.org>). There are many native drivers listed at <http://www.unixodbc.org/drivers.html> but perhaps ODBC is best bypassed.

An alternative is to use a better cross-platform driver such as the MySQL Connector/Net, available from <http://dev.mysql.com/downloads/connector/net>. The download links imply Windows support, but really the archive contains a managed assembly `mysql.data.dll` that you need to reference from your project (remember to set the Copy Local option to True in the Properties for this assembly reference, unless you plan to install it in the GAC (Global Assembly Cache) with the `gacutil` tool). Formal installation instructions are at <http://dev.mysql.com/doc/refman/5.1/en/connector-net-installation-unix.html>. With this driver the code changes to:

```

uses MySql.Data.MySqlClient;
...
class method ConsoleApp.GetSomeData(): string;
const
    rootPassword = '';
    //MySQL Connector/Net connection string

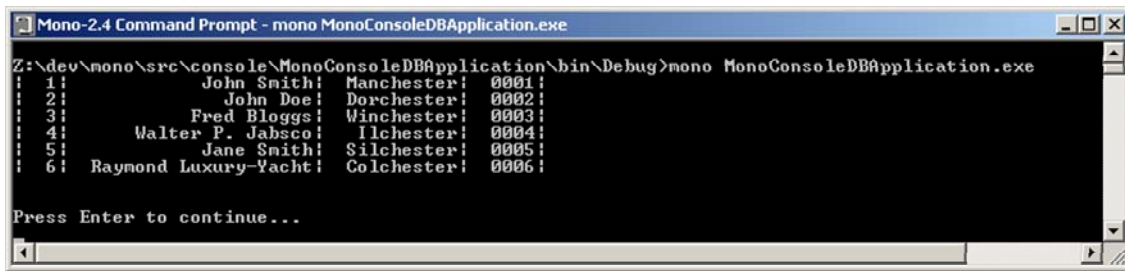
```

```

connectionString = 'Server=localhost;Username=root;Password=' + rootPassword;
selectSQL = 'SELECT * FROM Customers WHERE Town LIKE '%che%'';
widths: Array of Integer = [3, 22, 12, 6];
var
  results: StringBuilder := new StringBuilder();
begin
  using sqlConnection: MySqlConnection := new MySqlConnection(connectionString) do
  begin
    sqlConnection.Open();
    SetupData(sqlConnection);
    var dataAdapter: MySqlDataAdapter :=
      new MySqlDataAdapter(selectSQL, sqlConnection);
    var dataTable: DataTable := new DataTable('Results');
    dataAdapter.Fill(dataTable);
    for each row: DataRow in dataTable.Rows do
    begin
      results.Append('|');
      for I: Integer := 0 to dataTable.Columns.Count - 1 do
      begin
        var Width := 20;
        if I <= High(widths) then
          Width := widths[I];
        results.AppendFormat('{0,' + Width.ToString() + '}|', row.Item[I]);
      end;
      results.Append(Environment.NewLine);
    end;
    TearDownData(sqlConnection);
  end;
  Result := results.ToString()
end;

```

The code works just the same in Mono on Windows (and also in .NET):



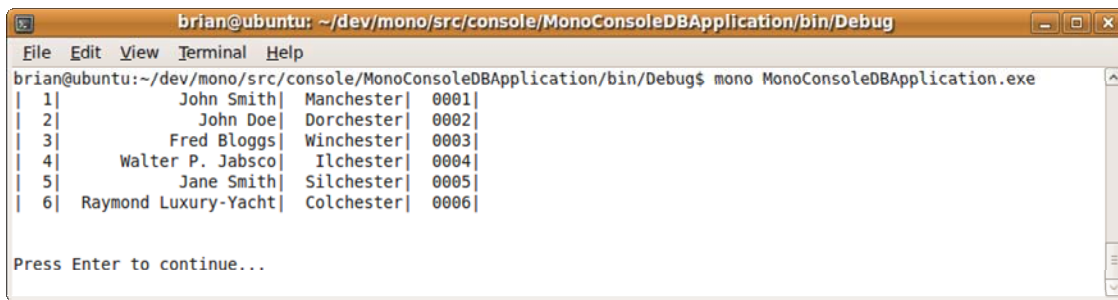
```

Mono-2.4 Command Prompt - mono MonoConsoleDBApplication.exe
Z:\dev\mono\src\console\MonoConsoleDBApplication\bin\Debug>mono MonoConsoleDBApplication.exe
1: John Smith| Manchester| 0001|
2: John Doe| Dorchester| 0002|
3: Fred Bloggs| Winchester| 0003|
4: Walter P. Jabsco| Ilchester| 0004|
5: Jane Smith| Silchester| 0005|
6: Raymond Luxury-Yacht| Colchester| 0006|
Press Enter to continue...

```

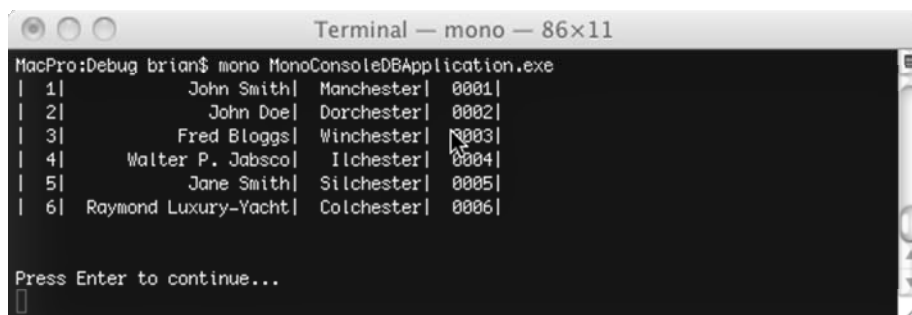
As with Windows, MySQL is not installed by default on Linux or OS X. You can download the Mac version from the MySQL web site (just be sure to read the README carefully for instructions on how to start the service). It will vary between Linux distros, but Ubuntu will install MySQL with: `sudo apt-get install mysql-server`

Data access then works in exactly the same manner on Linux and OSX:



```
brian@ubuntu: ~/dev/mono/src/console/MonoConsoleDBApplication/bin/Debug$ mono MonoConsoleDBApplication.exe
| 1|      John Smith| Manchester| 0001|
| 2|      John Doe|  Dorchester| 0002|
| 3|      Fred Bloggs| Winchester| 0003|
| 4|    Walter P. Jabsco| Ilchester| 0004|
| 5|      Jane Smith| Silchester| 0005|
| 6| Raymond Luxury-Yacht| Colchester| 0006|

Press Enter to continue...
```



```
MacPro:Debug brian$ mono MonoConsoleDBApplication.exe
| 1|      John Smith| Manchester| 0001|
| 2|      John Doe|  Dorchester| 0002|
| 3|      Fred Bloggs| Winchester| 0003|
| 4|    Walter P. Jabsco| Ilchester| 0004|
| 5|      Jane Smith| Silchester| 0005|
| 6| Raymond Luxury-Yacht| Colchester| 0006|

Press Enter to continue...
```

THE PROBLEM OF GUI APPLICATIONS

This is where things get interesting/messy (delete as applicable). Application GUIs are very specific to the OS they run on. Windows applications have a distinct look and feel because they are built with Windows controls using some variant of some version of the *Windows User Experience Interaction Guidelines* (<http://msdn.microsoft.com/en-us/library/aa511258.aspx>) - these guidelines have changed over the years, so Windows applications vary, but they are still readily recognizable.

Linux applications have less of a common look and feel, though those built with the same GUI toolkit typically have some visual consistency.

Mac applications have a definite look and feel thanks to the almost exclusive use of the Cocoa library, as mentioned earlier. Additionally the Apple *Human Interface Guidelines* make very clear how Mac applications should act and look in order to be acceptable (<http://developer.apple.com/mac/library/documentation/UserExperience/Conceptual/AppleHIGuidelines>).

So what happens here with regard to cross-platform solutions, in particular cross-OS? You have to make a choice, partly based on your expected user base, partly based on what you already have. If you have a .NET application you are looking to take cross-platform, then you presumably already have your GUI developed. You might consider a first cut of a cross-platform app by leaving the GUI as a WinForms UI. WinForms is supported by Mono, despite being

outside the ECMA specification. The API is all supported although some minor features do not actually function.

Using your existing WinForms knowledge may also be acceptable if your user base is deemed to be quite small or quite forgiving. The issue here being that your WinForms application will stick out as inconsistent in the Linux GNOME or KDE desktops, or on OS X. Mac users in particular are unwelcoming to non-Cocoa applications but that's a call for you to make.

The alternative is to consider building a different GUI for your application. You could go all-in and build a new application using a different cross-platform UI toolkit, for example GTK# (a Mono layer over GTK+). This will favor Linux users as GTK+ is the toolkit used to build the popular GNOME desktop, but will again look out of place on the Mac. There are other similar toolkits that Mono supports for cross-platform UI development, such as Qyoto (a layer over Qt giving a Linux KDE desktop look and feel) and wxNet (a layer over wxWindows)

A further alternative is to ensure all your business logic is isolated as much as possible in self-contained assemblies with appropriately diverse calling interfaces, and to build different front ends for each target platform. A number of applications take this approach. Clearly this adds considerable learning and development efforts but does result in a native-looking application on each OS.

Something that can guide our choices here a little is to limit ourselves to which toolkits Delphi Prism has project templates for. This includes WinForms and GTK# for cross-OS projects, and Cocoa# and Monobjc for OS X. There is nothing stopping you looking at Qyoto and wxNet or other Mono GUI toolkit layers that may exist, but at least you get a starting point with the project templates.

Let's take a look at these toolkits.

GUI TOOLKITS

WINFORMS

Mono's implementation of WinForms had a couple of false starts, where they tried to layer it over existing graphical toolkits and eventually ran into the limitations of trying to fit a square peg into a round hole. The current implementation uses `System.Drawing` to render all controls, and `System.Drawing` uses a platform-dependent driver to talk with the underlying OS windowing system. Currently, though, the rendering is only done using a Windows theme; Linux and OS X themes might possibly arrive in the future.

In principle you should be able to take your WinForms application from .NET and run it under Mono on Linux or OS X, however it would be worthwhile running it through MoMA (Mono Migration Analyzer) to see if there are foreseeable issues with what your code does.

To check out cross-platform WinForms support you could either take an existing Delphi Prism WinForms .NET project or create a new WinForms project using either the .NET project template or the Mono WinForms project template for Mac OS X. They will both allow us to

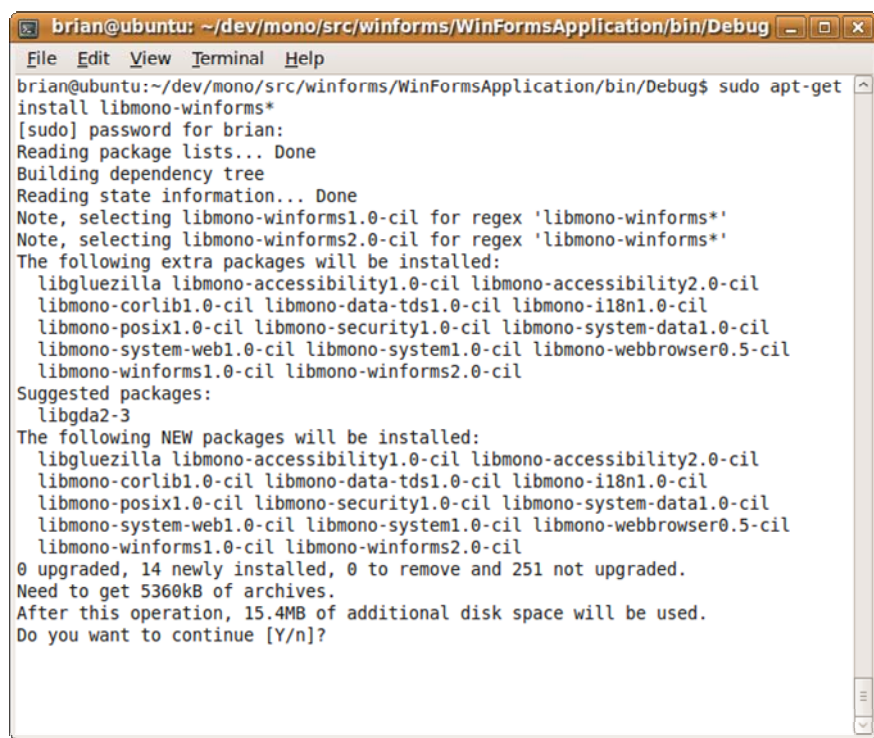
develop and build a WinForms application but clearly there are some differences in the Mac OS X template.

The two key differences are:

- the Mono project will be linked against the Mono framework assemblies
- an application bundle will be created for the application for OS X users. This allows a custom icon to be used by the Mac Dock and Task Switcher, which is good as without this the application is left with a generic executable application icon.

Delphi Prism doesn't directly invoke Mono's macpack tool to generate app bundles, but uses a custom MSBuild target that offers the same functionality.

Let's look at a sample WinForms application that uses a `TreeView` control and a `NotifyIcon` control (we'll see more about how this example came about when we see the GTK# equivalent application in the next section). The first problem we hit is that Mono WinForms support is not installed on Ubuntu (it's not a core part of Mono, what with it not being part of the ECMA specification). This is easily resolvable, as usual:



```
brian@ubuntu: ~/dev/mono/src/winforms/WinFormsApplication/bin/Debug
File Edit View Terminal Help
brian@ubuntu:~/dev/mono/src/winforms/WinFormsApplication/bin/Debug$ sudo apt-get
install libmono-winforms*
[sudo] password for brian:
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting libmono-winforms1.0-cil for regex 'libmono-winforms*'
Note, selecting libmono-winforms2.0-cil for regex 'libmono-winforms*'
The following extra packages will be installed:
  libgluezilla libmono-accessibility1.0-cil libmono-accessibility2.0-cil
  libmono-corlib1.0-cil libmono-data-tds1.0-cil libmono-il8n1.0-cil
  libmono-posix1.0-cil libmono-security1.0-cil libmono-system-data1.0-cil
  libmono-system-web1.0-cil libmono-system1.0-cil libmono-webbrowser0.5-cil
  libmono-winforms1.0-cil libmono-winforms2.0-cil
Suggested packages:
  libgda2-3
The following NEW packages will be installed:
  libgluezilla libmono-accessibility1.0-cil libmono-accessibility2.0-cil
  libmono-corlib1.0-cil libmono-data-tds1.0-cil libmono-il8n1.0-cil
  libmono-posix1.0-cil libmono-security1.0-cil libmono-system-data1.0-cil
  libmono-system-web1.0-cil libmono-system1.0-cil libmono-webbrowser0.5-cil
  libmono-winforms1.0-cil libmono-winforms2.0-cil
0 upgraded, 14 newly installed, 0 to remove and 251 not upgraded.
Need to get 5360kB of archives.
After this operation, 15.4MB of additional disk space will be used.
Do you want to continue [Y/n]?
```

A quick check with MoMA says all is fine with our use of WinForms. I checked with the default current Mono version definitions file, and also pulled down the definitions file for Mono version 2.0 (the one on Ubuntu 9.04).

Now if you run it on Linux it executes and pretty much works; pretty much, but not perfectly, as I will explain.

The application works like this. It starts up and enumerates all the assemblies loaded in its process, and all the types in those assemblies and all the members in each of those types and loads them into a `TreeView`. Whilst loading all this information a progress dialog is on-screen showing what is being loaded. When all is loaded, the progress dialog disappears and the main form with the populated `TreeView` is displayed. That's about it for the main functionality, but in addition the application sets up a `NotifyIcon` control. This shows up in Windows as an icon in the Taskbar Notification Area (often wrongly referred to as the system tray). The notification control is programmed to respond to a mouse click (`Click` event) by toggling the visibility of the form; it also responds to a right-click by popping up a menu with a `Quit` option on it (thanks to the `ContextMenuStrip` property).

In Windows all is well, it responds to the left-click and right-click just fine. However on Linux things aren't so clear-cut. The notification icon responds to a left-click correctly; however its response to a right-click seems at first glance a little odd. Both events trigger for it - you get the popup menu and the visibility changes. Clearly the Mono implementation takes the `Click` event name literally - any type of click triggers the event. This is not the sort of thing that MoMA picks up. I'd be inclined to suggest this was a bug (being inconsistent) in the Linux side of Mono WinForms. Perhaps it was not such a sage move to have both a `Click` event handler and a `ContextMenuStrip` after all.

It also runs on OS X. Kind of; but not entirely. You see, the `NotifyIcon` control is not fully implemented in OS X and when the code tries to set the icon you get:



Again, MoMA did not see this as it is looking more for missing methods more than missing functionality. If you pulled down the Mono class library source (as described earlier) you can see the offending routines in `mscorlib\managed.Windows.Forms\System.Windows.Forms\XplatUICarbon.cs` (note that the Cocoa Mac UI framework is itself based on the Carbon framework). Specifically `SystrayAdd()`, `SystrayChange()` and `SystrayRemove()` all throw a `NotImplementedException` and are marked with the `MonoTODO` attribute, so you need to avoid this can of worms by conditionalising some of the code based on the OS:

```
method MainForm.MainForm_Load(sender: System.Object; e: System.EventArgs);  
begin  
    if not RunningOnOSX then  
        notifyIcon.Icon := Icon;  
    end;  
end;
```

With this done the application runs fine on OS X:



MAC APPLICATION ICONS

Notice the custom Dock icon? This was achieved through the content of the application bundle set up by the Mono WinForms project template. Each template that sets up an app bundle has a default App.icns file (an Apple icon image file) in the project that is sent into the app bundle, and which can be replaced. However if you replace it with a file of a different name you need to make Delphi Prism aware of it as a Mac icon file.

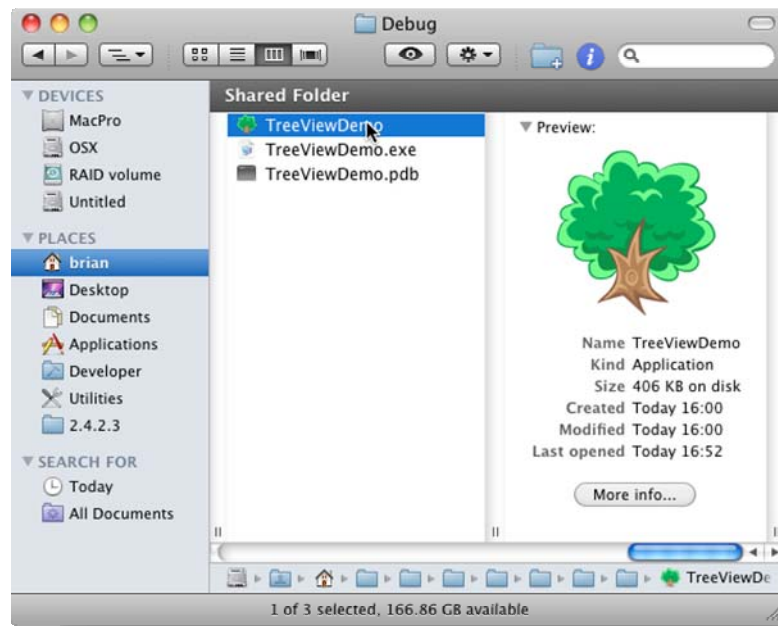
In the current release of Delphi Prism (August 2009) this requires you to edit your project file (the .oxygene file), which is stored in XML format. Nested within the main `PropertyGroup` element you need to add a `MacIconFile` element, for example:

```
<MacIconFile>Properties\Tree.icns</MacIconFile>
```

Note: The Apple Xcode tools include an Icon Composer application in `/Developer/Applications/Utilities` that can be used to build .icns files and .ico files. Note this tool is for building the files, not for editing the images, but if you have suitable .png files for the various resolutions required you can simply drag them into the placeholder areas in an .icns file or .ico file in Icon Composer. There are web sites with free images that can be used to build icons, such as http://www.zoobapps.com/free_icons.

In the case of a WinForms application that can also be executed on other platforms, be sure to set the application icon as you would normally, as well as the `Icon` property for your forms.

Now that you have a custom application icon, not only does it show up in the Dock when the application is running, but of course it is displayed by Finder when looking at the application bundle as well. In the screenshot below you can see the actual WinForms executable (and debug symbol file), and also the application bundle (remember Finder by default hides the .app extension on application bundle directories).



GTK#



GTK+ (<http://www.gtk.org> and <http://www.gtk-osx.org>) is a cross-platform graphical widget toolkit (where a widget is a control). It is an object-oriented version of the original GTK that was built specifically to develop the popular GNU image editor GIMP (GNU Image Manipulation program), hence GTK being the GIMP Tool Kit. GTK# (<http://www.mono-project.com/GtkSharp>) is a .NET binding over GTK+ allowing Mono GTK+ applications to be developed.

Documentation can be found under Gnome Libraries in the Mono documentation (<http://www.go-mono.com/docs>) - use the hierarchy browser on the left of the page. It can also be useful to keep a link to the original GTK+ documentation, which you can find at <http://library.gnome.org/devel/references>.

GTK+ is based on separate libraries maintained by the same team, all represented in GTK#:

- Glib - core library providing the event loop and other runtime functionality
- Pango - text rendering and layout library, supporting internationalization
- GDK - GIMP Drawing Kit - insulates GTK+ from the windowing system
- Cairo - cross-platform 2D graphics rendering system
- ATK - accessibility toolkit

Building a GTK# application can either be thoroughly code-based, or you can use the Glade UI designer (<http://glade.gnome.org>), with binaries available for Windows and OS X. Glade allows you to design the user interface entirely separate from the code, with the definition saved in an XML file. It presents the regular UI design paradigm: lay out controls, or widgets, on one or more windows, set the properties of the controls and set up the names of methods that respond to things happening. In GTK+ and GTK# events are referred to as *signals* and event handlers are referred to as *callbacks*.

Note: Glade XML files are loaded by functionality in libglade. Technically, libglade is heading toward deprecation, to be replaced by GtkBuilder (added to GTK+ in 2007). As of version 3.6, Glade works with both libglade and GtkBuilder files, however GTK# currently only supports libglade. Presumably at some point in the future GtkBuilder support will be added to GTK#, but bear in mind which format to use when using Glade - the Delphi Prism GTK# project template contains a libglade XML file.

Note: When using Glade to edit your UI it is advisable to follow the mantra “save and save often.” I have encountered a number of frustrating Glade crashes that have caused a loss of work.

Note: If using Glade on a Mac you should be very wary of the Delete key on the keyboard when editing text properties. It is instinctive to expect it to delete the character after the cursor, as it does on Windows; however on the Mac it has a tendency to delete the currently selected widget, which may include a whole hierarchy of child widgets.

Note: Use of Glade should be optional very soon as work is under way to have Delphi Prism integrate with MonoDevelop (<http://monodevelop.com>). MonoDevelop also has an XML-based UI design approach for GTK# applications but uses a dedicated GTK#-friendly editor called Stetic, which bypasses both libglade and GtkBuilder.

The basic GTK# application has a Main.pas file that notionally contains the main form. In truth it really contains a class (MainForm) that can act for the main form and as many other forms as you choose. Also included in the project is *Main.glade*, the libglade XML file. This contains a simple definition of a GTK+ window with a VBox container on it and a delete_event callback handler set. The MainForm class looks like this:

```
type
  MainForm = class(System.Object)
  private
    {$REGION Glade Widgets}
    var
      [Widget] window1: Gtk.Window;
    {$ENDREGION}
  public
    constructor(args: array of String);

    method on_window1_delete_event(aSender: Object; args: DeleteEventArgs);
  end;

constructor MainForm(args: array of String);
begin
  inherited constructor;
  Application.Init();
```

```

with lXml := new Glade.XML(nil, 'GtkApplication1.Main.glade', 'window1', nil) do
    lXml.Autoconnect(self);

    Application.Run();
end;

method MainForm.on_window1_delete_event(aSender: Object; args: DeleteEventArgs);
begin
    Application.Quit();
end;

```

Here you can see a declaration for the window (adorned with a `widget` attribute), the callback handler, and the code that sets things in motion. The `Glade.XML` class loads the libglade XML file and creates a user interface defined inside. This particular constructor allows the XML file to be found as a resource in a specified assembly (the first `nil` means the current assembly). The second parameter is the resource name - note that the resource name starts with the assembly name. The third parameter is the widget node to start building the UI from, typically a top level window name, and the last parameter is an XML translation domain, typically `nil`.

As soon as the XML object is created its `Autoconnect()` method is called with a reference to this class passed in. This connects all the signals defined in the glade file with the callback handlers defined in the class, and also wires up fields defined in the class with the `widget` attribute to the same-named widgets in the UI.

TWEAK THE GTK# PROJECT CODE

There are factors that normally cause me to update this setup code. The XML file will always be embedded in my main executable assembly so I use a simpler constructor overload. Also, since I may save my project under a new name, or rename the executable after building it, I remove the literal reference to the assembly name in the XML resource name (which will be invalid as soon as the executable is renamed) and calculate it dynamically instead. Finally, to avoid the compiler complaining about a lack of assignment to the widget variables (they are, after all, assigned behind the scenes) I add in a `nil` assignment to their declarations. All this changes the constructor to as follows. Note that the Glade XML object is created without a variable, and `Autoconnect()` is called directly before leaving it to fend for itself until the Garbage Collector finds it.

```

uses
    ...
    System.Reflection;

type
    MainForm = class(System.Object)
    private
        var AssemblyName: String;
        {$REGION Glade Widgets}
        var
            [Widget] window1: Gtk.Window := nil;
        {$ENDREGION}
        ...
    end;

constructor MainForm(args: array of String);
begin
    inherited constructor;
    Application.Init();

```

```
AssemblyName := &Assembly.GetEntryAssembly().GetName().Name;  
new Glade.XML(  
    AssemblyName + '.Main.glade', 'myGladeWindow').Autoconnect(self);  
Application.Run();  
end;
```

You simply add in as many additional widget variable declarations as you require, based on the widgets you build into your UI in the Glade designer.

Note: In the August 2009 release of Delphi Prism, the GTK# project template references all the various GTK# assemblies but has the Copy Local property set to True for all of them. This has no impact on Windows, but causes problems on Linux and OS X. The thing is, there is no need to copy the GTK# assemblies - they are a part of the Mono distribution. Unfortunately having them sat in the executable's directory without their config files breaks required loading behavior on non-Windows platforms and gives the error: *Unhandled Exception: System.DllNotFoundException: libglib-2.0-0.dll.*

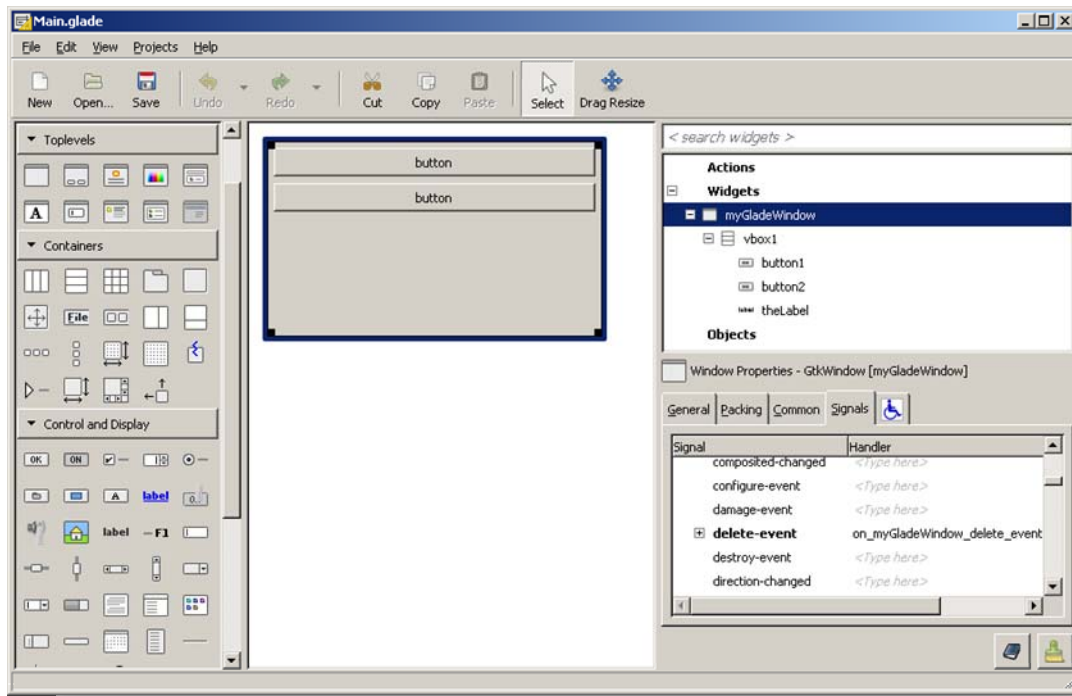
GTK# EXAMPLES

To learn something of how GTK# works (and later, Monobjc) we'll look at how to implement some example programs detailed in a few tutorials scattered about the web; both for the Mono wrapper GTK# and for the original GTK+. This can be quite useful as you can see the differences required to achieve the same results when using Delphi Prism and Mono and compare the steps with the original tutorial. This will help when looking at additional tutorials that you find to learn how to achieve other goals in the future.

SIMPLE GTK# EXAMPLE

The first example to try is a very simple GTK# application and the original tutorial is located at <http://www.box.net/public/aqu7jo4uby>, by Paul Hogan (aka pachjo). This tutorial uses Python as the programming language and appears to use an older version of Glade that offers GTK+ and GNOME as project types over GtkBuilder and Libglade, but again, use Libglade when asked.

Build up the trivial UI in Glade as per the directions - you should be looking at something like this:



Note that there is a `Label` control below the two buttons. Also note that signal handler names can either be entered manually or a reasonably sensible name can be selected from a drop-down list in the *Signals* tab.

This example is quite simple and has both buttons update the label's caption from their clicked signal handler. This is the important code, including the manually entered signal handler methods:

```
type
  MainForm = class(System.Object)
  private
    var AssemblyName: String;
    {$REGION Glade Widgets}
    var
      [Widget] theLabel: Gtk.Label := nil;
    {$ENDREGION}
  public
    constructor(args: array of String);
    method on_myGladeWindow_delete_event(aSender: Object; args: DeleteEventArgs);
    method on_button1_clicked(sender: Object; args: EventArgs);
    method on_button2_clicked(sender: Object; args: EventArgs);
  end;
```

```

constructor MainForm(args: array of String);
begin
    inherited constructor;
    Application.Init();
    AssemblyName := &Assembly.GetEntryAssembly().GetName().Name;
    new Glade.XML(
        AssemblyName + '.Main.glade', 'myGladeWindow').Autoconnect(self);
    Application.Run();
end;

method MainForm.on_myGladeWindow_delete_event(aSender: Object; args: DeleteEventArgs);
begin
    Application.Quit();
end;

method MainForm.on_button1_clicked(sender: Object; args: EventArgs);
begin
    theLabel.Text := 'You pressed button1';
end;

method MainForm.on_button2_clicked(sender: Object; args: EventArgs);
begin
    theLabel.Text := 'You pressed button2';
end;

```

As you can see, it's quite straightforward: standard PME (property, method & event) coding. Hopefully you navigated your way around the Glade designer without too much ado.

DIALOG EXAMPLE

The next tutorial to tackle in GTK# is <http://tadeboro.blogspot.com/2009/04/gtkdialog-tutorial-part-2.html> by Tadej Borovšak aka tadeboro. This takes us through building a confirmation dialog used to confirm the user wishes to quit, and also extending and using the About box template. Other than a bit more time in the Glade designer, following the tutorial requires a couple of signal handlers as follows:

```

method MainForm.on_mainWindow_delete_event(aSender: Object; args: DeleteEventArgs);
begin
    new Glade.XML(AssemblyName + '.Main.glade', 'confirmQuitDialog').Autoconnect(self);
    try
        confirmQuitDialog.Icon := new Pixbuf(nil, AssemblyName + '.Properties.Tux.png');
        args.RetVal := confirmQuitDialog.Run <> 1;
        confirmQuitDialog.Hide;
    finally
        confirmQuitDialog.Destroy;
        confirmQuitDialog := nil;
    end;
end;

```

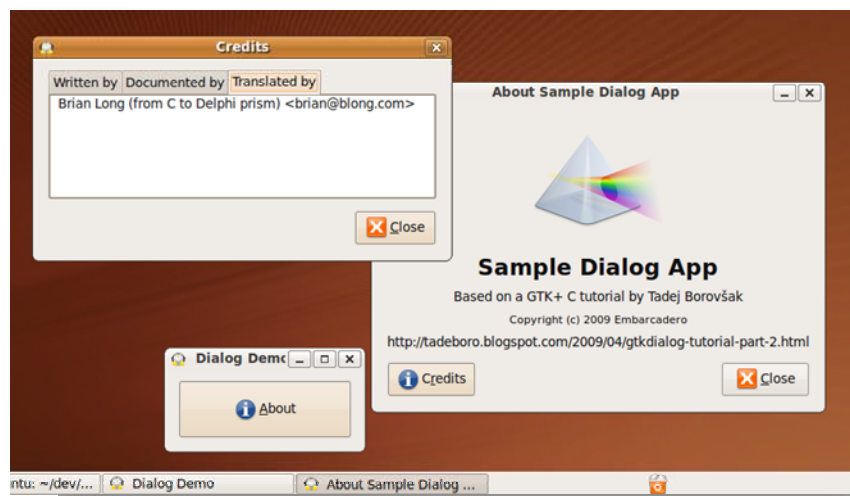


```

method MainForm.on_aboutButton_clicked(aSender: Object; args: EventArgs);
begin
    new Glade.XML(AssemblyName + '.Main.glade', 'aboutDialog').Autoconnect(self);
    try
        aboutDialog.Icon := new Pixbuf(nil, AssemblyName + '.Properties.Tux.png');
        aboutDialog.Logo := new Pixbuf(nil, AssemblyName + '.PrismLogo.png');
        aboutDialog.Run;
        aboutDialog.Hide;
    finally
        aboutDialog.Destroy;
        aboutDialog := nil;
    end;
end;

```

Note that I have also chosen to add a PNG image into the project in the *Properties* directory, and use this as the icon for each window in the application (this shows up on Linux and on Windows). Additionally I added a regular icon file to the application so that when the .exe file is examined on Windows, the icon will be visible.



You can see that the pre-fabricated About box template is quite flexible inasmuch as it automatically offers a Credits screen.

Note: The Glade auto-connect mechanism does a good job of hooking variables up to UI widgets, but occasionally you find a corner case where it fails. For example, in the About dialog template there is an internal VBox `dialog-vbox3`, in which I wanted to load an image from a resource file. However the regular approach to having the image control surface as a variable failed, possibly due to the VBox being an internal control.

TREEVIEW EXAMPLE

The next demo application uses the GTK+ TreeView control. Unfortunately the Glade designer (or more specifically the libglade support) does not support working with the treeview (though the GtkBuilder support does) so we'll deal with it in code. TreeView is interesting in that it forces separation of the UI from the data displayed there; the data is set up in a TreeStore (which implements the `TreeModel` interface), and this is later fed to the TreeView via its

constructor or Model property. Also, the GTK+ TreeView supports columns of information in addition to the displayed hierarchy, something that the WinForms TreeView control does not.

The TreeView class is documented in the Mono documentation library at <http://tinyurl.com/Gtk-TreeView> where you will also find a simple example of its use along with a more advanced example, both written in C#. This ends up being quite easy to port over to Delphi Prism, all the more so if you ignore all the regular UI building code and do those steps in the Glade designer. This is what the main parts of the code end up looking like:

```

type
  MainForm = class(System.Object)
  private
    var AssemblyName: String;
    {$REGION Glade Widgets}
    var
      [Widget] mainWindow: Gtk.Window := nil;
      [Widget] scrolledWindow: ScrolledWindow := nil;
      [Widget] updateDialog: Gtk.Dialog := nil;
      [Widget] dialogLabel: Label := nil;
    {$ENDREGION}
    store: TreeStore;
    method UpdateProgress(format: String; params args: Array of object);
    method ProcessType(parent: TreeIter; t: &Type);
    method ProcessAssembly(parent: TreeIter; asm: &Assembly);
    method PopulateStore;
  public
    constructor(args: array of String);
    method on_window1_delete_event(aSender: Object; args: DeleteEventArgs);
    method on_updateDialog_response(aSender: Object; args: ResponseArgs);
  end;
  ...
implementation

uses
  System.Security;

constructor MainForm(args: array of String);
begin
  inherited constructor;
  AssemblyName := &Assembly.GetEntryAssembly().GetName().Name;
  Application.Init();
  PopulateStore;
  new Glade.XML(AssemblyName + '.Main.glade', 'mainWindow').Autoconnect(self);
  mainWindow.Icon := new Pixbuf(nil, AssemblyName + '.Properties.Tree.png');
  var tv: TreeView := new TreeView(store);
  tv.HeadersVisible := True;
  tv.AppendColumn('Name', new CellRendererText(), 'text', 0);
  tv.AppendColumn('Type', new CellRendererText(), 'text', 1);
  scrolledWindow.Add(tv);
  updateDialog.Destroy;
  updateDialog := nil;
  mainWindow.ShowAll();
  Application.Run();
end;

method MainForm.on_window1_delete_event(aSender: Object; args: DeleteEventArgs);
begin
  Application.Quit();
end;

```

```

method MainForm.PopulateStore;
begin
    if store <> nil then
        Exit;
    store := new TreeStore(typeof(string), typeof(string));
    for each asm: &Assembly in AppDomain.CurrentDomain.GetAssemblies() do
        begin
            var asmName: String := asm.GetName().Name;
            UpdateProgress('Loading {0}', asmName);
            ProcessAssembly(store.AppendValues(asmName, 'Assembly'), asm)
        end;
    end;

method MainForm.UpdateProgress(format: String; params args: Array of object);
begin
    var Text := string.Format(format, args);
    if updateDialog = nil then
        begin
            new Glade.XML(AssemblyName + '.Main.glade', 'updateDialog').Autoconnect(self);
            updateDialog.Icon := new Pixbuf(nil, AssemblyName + '.Properties.Tree.png');
            dialogLabel.Text := Text;
            updateDialog.ShowAll();
        end
    else
        begin
            dialogLabel.Text := Text;
            while Application.EventsPending() do
                Application.RunIteration()
            end;
        end;
    end;

method MainForm.ProcessAssembly(parent: TreeIter; asm: &Assembly);
begin
    var asmName: String := asm.GetName().Name;
    for each t: &Type in asm.GetTypes() do
        begin
            UpdateProgress('Loading from {0}: '#10'{1}', asmName, t.ToString());
            ProcessType(store.AppendValues(parent, t.Name, t.ToString()), t);
        end;
    end;

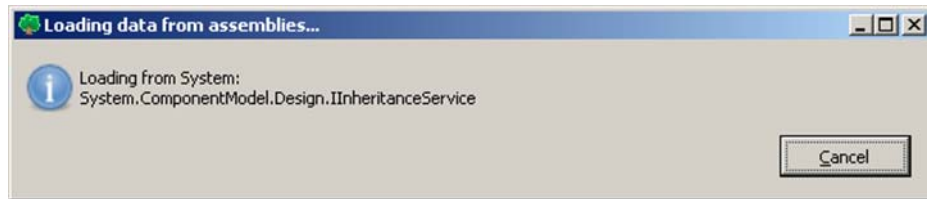
method MainForm.ProcessType(parent: TreeIter; t: &Type);
begin
    for each mi: MemberInfo in t.GetMembers() do
        store.AppendValues(parent, mi.Name, mi.ToString())
    end;

method MainForm.on_updateDialog_response(aSender: Object; args: ResponseArgs);
begin
    //Application.Quit();
    System.Environment.Exit(0);
end;

```

When the application starts up (MainForm constructor) it sets up the `TreeStore` via `PopulateStore()` before doing anything with the main window. This method iterates through each assembly in the process, then each type in the assemblies and each member in each type, adding information about each item into the store using the helper methods `ProcessAssembly()` and `ProcessType()`. Additionally, each step of the way the user is alerted to what is going on by `UpdateProgress()`, which ensures the progress dialog is displayed on

the screen (with an associated window icon) and updates a label on it with the passed in progress message:



An enhanced version of this example contains the GTK+ About dialog, set up with pretty image, credits etc. as well as a menu allowing the user to choose an additional assembly to loop through and add into the tree view. The code for these additions is not very informative, but there is also a `StatusIcon` widget used to add a notification icon into the system, based on the sample code at <http://www.mono-project.com/GtkSharpNotificationIcon>. As usual the code ports over straightforwardly, but in this case I've fixed some shortcomings:

```

constructor MainForm(args: array of String);
begin
    ...
    //Set up a tray icon
    trayIcon := new StatusIcon(new PixBuf(nil, AssemblyName + '.Properties.Tree.png'));
    trayIcon.Visible := True;
    //Show/hide the window when the icon is clicked
    trayIcon.Activate += method begin mainWindow.Visible := not mainWindow.Visible end;
    //Set up a context menu for the icon
    trayIcon.PopupMenu += OnTrayIconPopup;
    trayIcon.Tooltip := 'TreeView Demo Icon';
    //Start the app proper
    mainWindow.ShowAll();
    Application.Run();
end;

method MainForm.OnTrayIconPopup(sender: Object; args: EventArgs);
begin
    //Ensure we don't get lots of popups if we keep clicking in different spots
    if popupMenu = nil then
    begin
        popupMenu := new menu();
        var menuItemQuit: ImageMenuItem := new ImageMenuItem('Quit');
        menuItemQuit.Image := new Gtk.Image(Stock.Quit, IconSize.Menu);
        popupMenu.Add(menuItemQuit);
        //Quit when the menu item is clicked
        menuItemQuit.Activated += method begin ExitNicely end;
    end;
    popupMenu.ShowAll();
    popupMenu.Popup();
end;

```

```

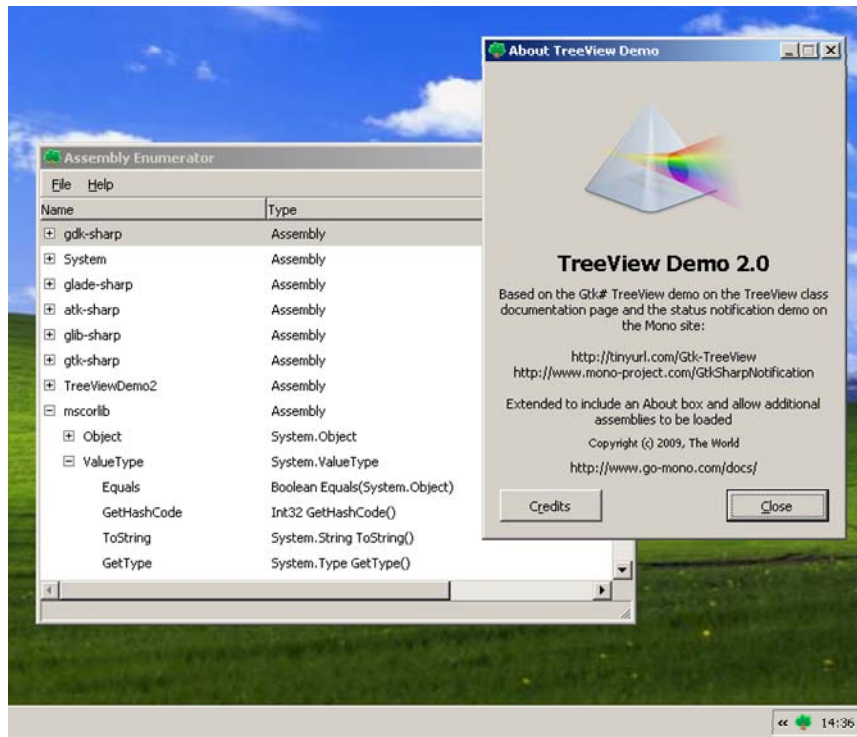
method MainForm.ExitNicely;
begin
    if trayIcon <> nil then
    begin
        trayIcon.Visible := False;
        Application.Quit;
    end;
end;

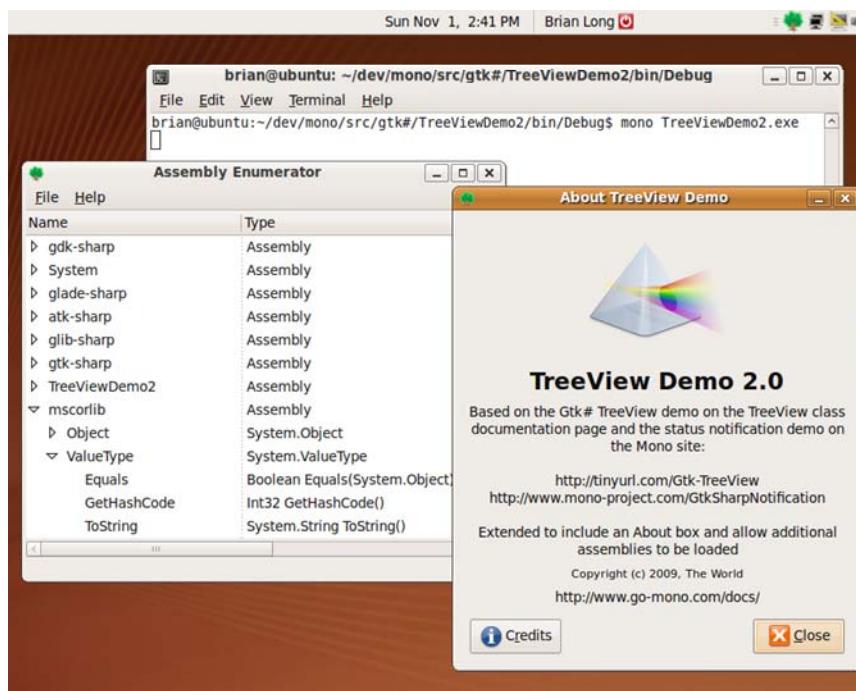
```

Notice the check to ensure you don't create multiple popup menus - without this if you right-click multiple times on the notification icon you would keep getting additional popup menus showing. Also, the termination code has been extended to ensure the notification icon is explicitly removed from the notification area to avoid a ghost icon being left there.

Note: The code above uses some Delphi Prism syntax extensions. The += operator adds a new event handler. In the case of the `Activate` and `Activated` events we are adding an anonymous delegate: a method body declared inline.

This example is now functionally equivalent to the WinForms example we looked at earlier. It operates successfully on Windows and Linux; even the status icon behaves as intended this time. The `StatusIcon`'s `Activate` signal is triggered on a left-click and the `PopupMenu` signal on a right-click.





On OS X the application isn't so good. Resizing the window causes it to jump all over the screen and the `StatusIcon` signal generation is different. A left-click triggers the `PopupMenu` signal and the `Activate` signal seems to not trigger at all. So there are some platform differences again in the case of the notification icon control but other than that GTK# does a good job of cross-platform UI.

GTK# BUNDLED EXECUTABLE

Let's turn one of these GTK# applications into a bundled executable so that it doesn't need to be launched via the `mono` command. This is a suitable operation for any application intended for use on Linux and for console applications on the Mac - GUI applications will use application bundles as discussed next.

Note: `mkbundle` will generate a bundled executable suitable for the platform it was called on. It generates different code based the running OS.

Note: Be careful to ensure your directory path does not contain a `#` in it when running `mkbundle` otherwise the process will fail. For example if my target Mono application is located in `~/dev/mono/src/gtk#/TreeViewDemo2/bin/Debug` and I run `mkbundle` from that directory against the `.exe` file I get this error thanks to the `#` being taken as a comment character in one of the command expansions:

Unhandled Exception: System.IO.FileNotFoundException: Could not load file or assembly '/Users/brian/dev/mono/src/gtk' or one of its dependencies. The system cannot find the file specified.

Things go very smoothly on Linux; you tell `mkbundle` to work out the dependencies and make a program called simply `treeviewdemo` and the job is done.

```
brian@ubuntu: ~/temp
File Edit View Terminal Help
brian@ubuntu:~/temp$ mkbundle TreeViewDemo2.exe --deps -o treeviewdemo
05 is: Linux
Sources: 1 Auto-dependencies: True
embedding: /home/brian/temp/TreeViewDemo2.exe
embedding: /usr/lib/mono/2.0/mscorlib.dll
embedding: /usr/lib/mono/gac/glade-sharp/2.12.0.0_35e10195dab3c99f/glade-sharp.dll
config from: /usr/lib/mono/gac/glade-sharp/2.12.0.0_35e10195dab3c99f/glade-sharp.dll.config
embedding: /usr/lib/mono/gac/glib-sharp/2.12.0.0_35e10195dab3c99f/glib-sharp.dll
config from: /usr/lib/mono/gac/glib-sharp/2.12.0.0_35e10195dab3c99f/glib-sharp.dll.config
embedding: /usr/lib/mono/gac/System/2.0.0.0_b77a5c561934e089/System.dll
embedding: /usr/lib/mono/gac/System.Configuration/2.0.0.0_b03f5f7f11d50a3a/System.Configuration.dll
embedding: /usr/lib/mono/gac/System.Xml/2.0.0.0_b77a5c561934e089/System.Xml.dll
embedding: /usr/lib/mono/gac/System.Security/2.0.0.0_b03f5f7f11d50a3a/System.Security.dll
embedding: /usr/lib/mono/gac/Mono.Security/2.0.0.0_0738eb9f132ed756/Mono.Security.dll
embedding: /usr/lib/mono/gac/gtk-sharp/2.12.0.0_35e10195dab3c99f/gtk-sharp.dll
config from: /usr/lib/mono/gac/gtk-sharp/2.12.0.0_35e10195dab3c99f/gtk-sharp.dll.config
embedding: /usr/lib/mono/gac/gdk-sharp/2.12.0.0_35e10195dab3c99f/gdk-sharp.dll
config from: /usr/lib/mono/gac/gdk-sharp/2.12.0.0_35e10195dab3c99f/gdk-sharp.dll.config
embedding: /usr/lib/mono/gac/Mono.Cairo/2.0.0.0_0738eb9f132ed756/Mono.Cairo.dll
config from: /usr/lib/mono/gac/Mono.Cairo/2.0.0.0_0738eb9f132ed756/Mono.Cairo.dll.config
embedding: /usr/lib/mono/gac/pango-sharp/2.12.0.0_35e10195dab3c99f/pango-sharp.dll
config from: /usr/lib/mono/gac/pango-sharp/2.12.0.0_35e10195dab3c99f/pango-sharp.dll.config
embedding: /usr/lib/mono/gac/atk-sharp/2.12.0.0_35e10195dab3c99f/atk-sharp.dll
config from: /usr/lib/mono/gac/atk-sharp/2.12.0.0_35e10195dab3c99f/atk-sharp.dll.config
Compiling:
as -o temp.o temp.s
cc -gdb -o treeviewdemo -Wall temp.c `pkg-config --cflags --libs mono` temp.o
temp.c: In function 'install_dll_config_files':
temp.c:62: warning: pointer targets in passing argument 2 of 'mono_register_config_for_assembly' differ in signedness
temp.c:64: warning: pointer targets in passing argument 2 of 'mono_register_config_for_assembly' differ in signedness
temp.c:66: warning: pointer targets in passing argument 2 of 'mono_register_config_for_assembly' differ in signedness
temp.c:68: warning: pointer targets in passing argument 2 of 'mono_register_config_for_assembly' differ in signedness
temp.c:70: warning: pointer targets in passing argument 2 of 'mono_register_config_for_assembly' differ in signedness
temp.c:72: warning: pointer targets in passing argument 2 of 'mono_register_config_for_assembly' differ in signedness
temp.c:74: warning: pointer targets in passing argument 2 of 'mono_register_config_for_assembly' differ in signedness
Done
brian@ubuntu:~/temp$
```

As mentioned, it's less important on OS X, but things don't go too smoothly on OS X 10.6. You may notice in the screenshot above that `mkbundle` invokes the GNU assembler `as`, the GNU C compiler `cc` and also `pkg-config`, a tool used to get compiler switches to allow successful linking to various libraries. On OS X (at least on my Mac Pro) it seems the assembler and linker default to an architecture that is incompatible with the generated assembly and C source files. Also, `pkg-config` is not in any directories on the path (though Mono does supply a copy on a directory not on the path, `/Library/Frameworks/Mono.framework/Commands`) and cannot find its data files for some of the required libraries (partly due to a lack of config, partly due to the Glib developer framework not being installed).

Fortunately all of this can be rectified by installing GTK+ for OS X (<http://www.gtk-osx.org>) and modifying some environment variables; I chose to edit my `~/.profile` file (which didn't initially exist) so each new session would have these pre-set. I added these lines to the file:

```
# scripts is where my own scripts are to be found
# Mono commands path is to allow pkg-config to be found

Export PATH=/Library/Frameworks/GLib.framework/Resources/dev/lib/␣
pkgconfig:/Library/Frameworks/Mono.framework/Commands:$PATH

# To allow Mono mkbundle to work

export AS="as -arch i386"
export CC="cc -arch i386"
```



```
export PKG_CONFIG_PATH=/Library/Frameworks/Mono.framework/Versions/Current/lib/pkgconf
ig
```

If you add modify your `.profile` you can force the shell to reload it immediately, to save starting a new shell, with:

```
. ~/.profile
```

This should give successful executable bundling on OS X.

GTK# MAC OS X APPLICATION BUNDLE

For proper deployment on OS X you should look at what's involved in making an application bundle for a GTK# application. In this case we currently have a Mono executable. For a decent app bundle you'll need a Mac icon (.icns file), so I made one called `Tree.icns`. Now it's just a case of invoking `macpack`:

```
macpack -n TreeViewDemo -i Tree.icns -m console TreeViewDemo2.exe
```

Clearly the `-i` switch specifies the icon file, but the other two switches bear some investigation. `-n` specifies the application name that will appear in Finder when you select the application bundle (as shown in the screenshot on page - 25 -), and is also used in the Task Switcher and in the tooltip for the application icon in the Dock when it is running. `-m` allows you to choose the type of application so the generated bundled script can set things up correctly. Values that you might feasibly choose include `winforms`, `cocoa` and `console`, but since the environment variables that would be set up for WinForms or Cocoa are not required, `console` will do just fine.

This is the manual way to generate an application bundle and is currently appropriate for console and GTK# applications on OS X. WinForms applications (as we saw earlier), Cocoa# applications and Monobjc applications have their projects set up to automatically generate an application bundle thanks to the Delphi Prism MacPack MSBuild task invoked via the project file.

If you prefer, you can skip the manual `macpack` invocation and make use of the IDE app bundle generator by modifying your GTK# project file (the XML format `.oxygene` file). You can either load the project file into some text editor of your choice, or you can edit it within Visual Studio by following these steps:

1. Ensure you have saved changes within your project/solution
2. Right-click on the project node in the Solution Explorer window hierarchy and choose `Unload project` (this changes the project node to say it is unavailable)
3. Right-click on the project node and choose `Edit <projectname>.oxygene` to load the XML project file into the Visual Studio editor
4. Edit the file as you require
5. Close the `.oxygene` file in the editor
6. Right-click the project node and choose `Reload project`

Nested within the main `PropertyGroup` element in the project file you need to add a `MacPackMode` element and, if desired, a `MacIconFile` element, for example:

```
<MacPackMode>Console</MacPackMode>
<MacIconFile>Properties\Tree.icns</MacIconFile>
```

You'll also need to import the MSBuild target file that enables the IDE-resident MacPack functionality. You can add this after the other RemObjects import element towards the end of the project file nested within Project element:

```
<Import Project="$(MSBuildExtensionsPath)\RemObjects Software\Oxygene\
  RemObjects.Oxygene.Cocoa.targets" />
```

Note: Having an app bundle means moving and installing the application are straightforward. It also means you can drag your application into the Dock to have a persistently available shortcut on hand.

Note: if generating your app bundle in the IDE, you must ensure the bundled script file has the execute bit set. If you build to a Windows drive that the Mac is sharing this will happen automatically. Otherwise you will have to set it manually with `chmod`, for example:

```
chmod +x TreeViewDemo.app/Contents/MacOS/TreeViewDemo
```

Cocoa#

Cocoa# (<http://cocoa-sharp.com>) is one of a number of projects designed to allow Cocoa applications to be built under Mono. However, it isn't the fastest moving project and has a number of issues waiting to bite you right from the off.

The Delphi Prism version that has been shipping since August 2009 contains a couple of Cocoa# project templates, one for OS X 10.4 (Tiger) and one for 10.5 (Leopard). Unfortunately neither of these seems to run correctly under OS X 10.6 (Snow Leopard). Even in 10.5, the Leopard project exhibits a very noticeable problem inasmuch as the Colors button on the toolbar crashes the application.

This is an issue in the Cocoa# libraries as opposed to any limitation with Delphi Prism. This and other issues were reported some time back to the Cocoa# maintainers, but still remain unfixed. Due to this the Cocoa# project templates will be removed in the next Delphi Prism release.

In short, let's dismiss Cocoa# as an idea good in theory and poor in implementation.

MONOBJC



This is where you spend some time doing real Mac development. Monobjc (<http://www.monobjc.net>) is a bridge technology that allows Mono applications to connect to various Apple Objective-C libraries including the main UI library, Cocoa. API documentation can be found at <http://api.monobjc.net> but of course it is useful to keep the original Cocoa documentation handy. As well as online at <http://developer.apple.com/mac/library/navigation>, you also have a local copy thanks to the Xcode tools:

```
/Developer/Documentation/DocSets/com.apple.adc.documentation.A
```

```
ppleSnowLeopard.CoreReference.docset.
```

Monobjc is not part of Mono but is a project that builds on Mono. The August 2009 version of Delphi Prism ships with Monobjc version 2.0.404.0 (installed in `C:\Program Files\Monobjc-2.0.404.0`) although at the time of writing the current released version is 2.0.413.0 and the current test version is 2.0.436.0. You should get the latest version of the library to develop against.

Let's jump straight in and make a new Monobjc application in Delphi Prism using the Monobjc project template. Once you've created the project and built it, you should locate the automatically generated application bundle in Finder and try and launch the new application to see what it offers.

Note: Finder will try to launch the application but it may well fail, depending on where the directory resides that you built the application to. If it's on a Windows drive that is shared and mounted onto the Mac then all will be well, otherwise you'll have to set the execute bit as discussed in the *Scripts* section on page - 15 -. However in this case the script is bundled into a nested directory in the app bundle. For an example project `MonobjcApplication1`, if you change to the directory containing the app bundle directory (the `bin/Debug` directory under the project directory) then this command will fix it:

```
chmod +x MonobjcApplication1.app/Contents/MacOS/MonobjcApplication1
```

Note: if you find that you are required to keep adding the execute bit in order to run the application through Finder or via `open` then you could perhaps use an alternative method. If you explicitly ask the shell to run the bundled script then the execute bit is not required. For a sample project `MonobjcApp`, if you `cd` to the directory that contains the project directory, then this will launch the application:

```
bash MonobjcApp/bin/Debug/MonobjcApp.app/Contents/MacOS/MonobjcApp
```

Note: The Monobjc project template in Delphi Prism has references to all the Monobjc assemblies and Copy Local is set to True for them all. This covers all eventualities and allows you to access the entire Monobjc bridge library, meaning you can access Cocoa, OpenGL, QuickTime, the PDF framework as well as the AddressBook, Image Kit, Security and Web Kit frameworks. However if you are just building a regular Cocoa application then you might not want all the additional framework assemblies deployed to your application directory. For most Cocoa applications you can delete all the Monobjc references except `Monobjc.dll` and `Monobjc.Cocoa.dll`.

Now you should be able to launch the application with Finder, or from a terminal window with `open`. The application's UI has a toolbar with a Colors button on it that launches a color panel, and a Fonts button that launches a font panel. However there is currently nothing for these controls to edit. There are disabled Print and Customize buttons on the toolbar and there is a reasonably well populated menu system and some of the menu items (such as About, Quit, Page Setup, Print) actually prompt things to happen, but in general the application has no real functionality.

.NIB FILES

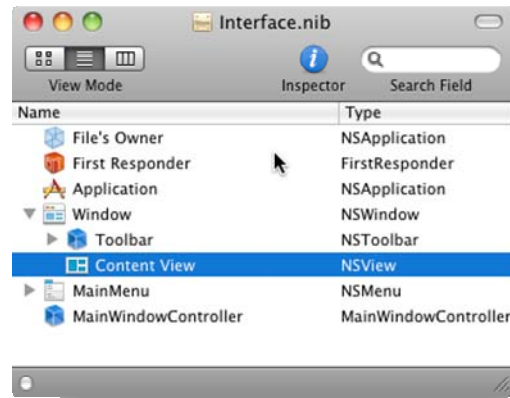
To add controls and build up a UI you need to use Apple's Interface Builder and edit the project's .nib file, which itself is a file bundle. In the project template the .nib bundle is called `Interface.nib` and inside it are the two files `designable.nib` and `keyedobjects.nib`.

.nib is a legacy extension dating back to NeXTSTEP and comes from NeXT Interface Builder. Historically .nib files were binary but more recently they may be stored as XML and optionally have the .xib extension. In the setup you get from the project template, designable.nib is XML and keyedobject.nib is binary. After the application is built the .nib bundle is stored in the *Resources* directory of the application bundle.

INTERFACE BUILDER

Interface Builder can be located in Finder as */Developer/Applications/Interface Builder*. When you open a .nib bundle, Interface Builder displays it in the Document window (Window, Document). The sample project .nib bundle shows up like this:

You can see the window (an `NSWindow` object) listed, along with the embedded toolbar (`NSToolbar`) and a content view (`NSView`) that fills up the rest of the window. These can be edited using the various pages of the Inspector (accessible from the Tools menu if not visible). The menu (an `NSMenu`) is also listed and can be edited in a visual menu designer by double-clicking it in the Document window. A Cocoa application can have multiple .nib files, but one of them will have a representation of the Application (`NSApplication`) object, also present here.

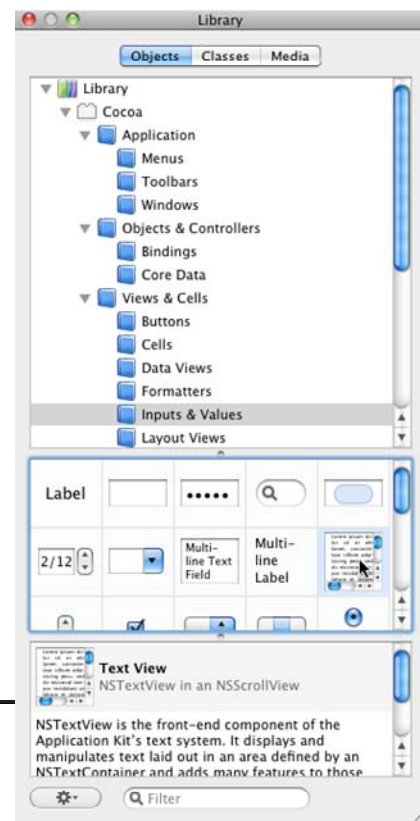


Note: the `NS` prefix on all the Cocoa classes is a legacy nomenclature aspect harking back to the days of NeXTSTEP.

The other main item of interest in the Document window is the main window controller. Without wanting to turn this into a full Cocoa tutorial, it is important to know that Cocoa enforces an MVC (Model-View-Controller) design pattern to one extent or another. The view is represented by what you design into the .nib file with Interface Builder. The model is represented in your code (as much as you choose to make your code act like a model). The controller connects up the two, is placed in the .nib file and is represented in your code as you will see over the coming sections.

You add additional controls to your window to build up the UI using the Library window (Tools, Library). When designing the UI you should take various aspects of Apple's *Human Interface Guidelines* into account. You can find these from Interface Builder by selecting Help, Human Interface Guidelines.

The controller object is typically defined as a class inherited from the default base Cocoa class `NSObject`.



Interface Builder allows you to set up descendant classes on the *Classes* tab on the Library window. Simply select the base class from the class list, right-click and choose *New Subclass...*; this will add a new class to the list for you to use. The *MainWindowController* class in the template project .nib is created like this as a class inherited from *NSObject*. You will find this custom class if you browse through the list of classes on the Library window's *Classes* tab.

With a controller in place you can add connections to it to allow the application code and UI to interact. You can add *outlet* connections, which ultimately surface as instance variables in the controller class source and which allow you to link the controller to a UI control. These allow your code to talk to important controls and views in the UI; it's almost like declaring variables that represent the controls you wish to talk to. You can also add *action* connections. Various UI controls offer a number of action messages that they will send when the user interacts with them (a message being sent can be considered similar to a method being called). You can define action methods in the controller that can be hooked up to control actions in order to enable code to respond to those actions - action methods are essentially event handlers. You'll see outlets and actions being set up in some examples later.

SIMPLE TEXT EDITOR

We'll start by editing the *Interface.nib* of the generated template project and make it do something useful. In this first instance we'll edit some of the menu items. If you use the menu editor and examine the text in the menu items you'll see that there are several instances of *NewApplication* as a place holder for your application name. There are three under the main application menu (also marked as *NewApplication*): *About NewApplication*, *Hide NewApplication* and *Quit NewApplication*. There is also one under the Help menu: *NewApplication Help*. You can edit each of these by double-clicking the menu item, or selecting it and editing the text on the *Attributes* pane of the Inspector.

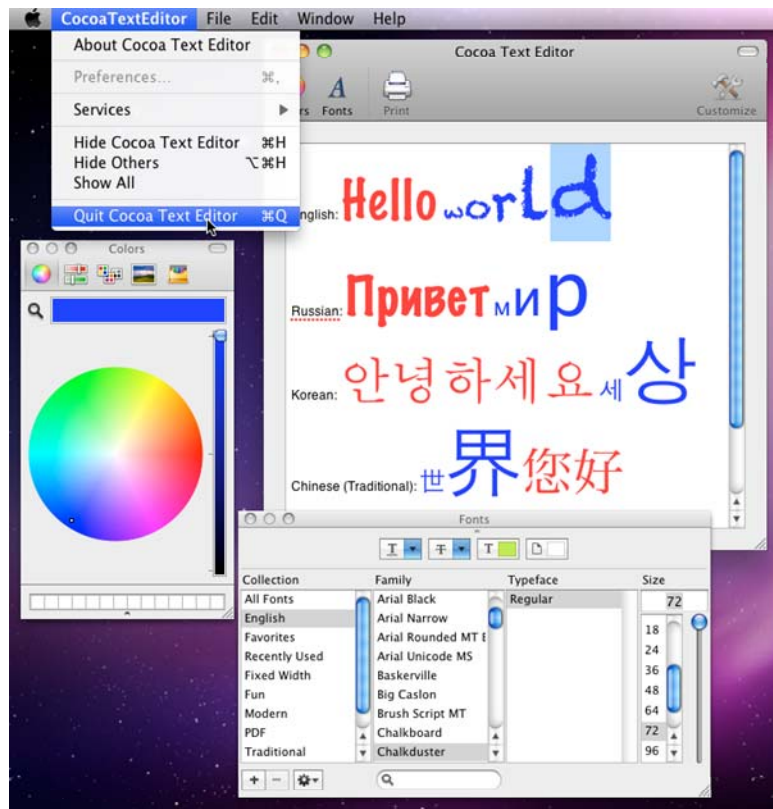
Note: editing the text of the bold menu item, *NewApplication*, will have little effect as the text for this is actually set at runtime from the application name.

Next add a Text View control onto the main part of the window. You can find this on the *Objects* tab under *Cocoa*, *Views & Cells*, *Inputs & Values*, but it is often quicker to type part of the name into the filter box at the bottom of the Library window. When you have added the control, you can size it to fit and you'll notice the sizing guides helping to get it suitably sized to fit in with the *Human Interface Guidelines*.

To further customize the control you should first ensure it is selected in the Document window. On the Inspector's *Size* tab you can set up automatic geometry management (i.e. have the text view resize in sync with the window being resized) by playing with the red springs and struts in the *Autosizing* section - an animation shows the effect of your chosen settings so you should be able to get the desired effect easily. The Inspector's *Attributes* page allows you to customer the border and scroll bars of the text view.

Don't forget that you can change the application icon by replacing *App.icns* in the project. If you choose a file with a different name then just update the *MacIconFile* entry in the project file as covered in the *Mac Application Icons* section on page - 24 -.

The application can now be run and you'll see that you have a capable rich text editor (albeit without the ability to load or save files):



It takes very little work to improve this editor. The sample menu added by the project template does not include a `Format` menu, but there is one in the Library. You can add it in by expanding the `MainMenu` in the Document window and dragging it to the appropriate point. This adds in menu items to launch the `Fonts` and `Colors` panels as well as other features including a very functional ruler that sits at the top of the Text View.

MONOBJC AND SNOW LEOPARD

Thanks to the introduction of thread-local garbage collection in the OS X Snow Leopard (see http://www.sealiesoftware.com/blog/archive/2009/08/28/objc_explain_Thread-local_garbage_collection.html for more information) certain assumptions made by low-level aspects of the `Monobjc` bridge library are now unsafe. This means that after a certain amount of time the application will hang or crash in the middle of the Mono garbage collection code (see https://bugzilla.novell.com/show_bug.cgi?id=537764). The `Monobjc` author, Laurent Etiemble, has analyzed the problem and a new build remedies the problem (at the time of writing this new build, 2.0.436.0, was being tested).

To avoid the issue the new version of `Monobjc` introduces two native shared libraries, (one for each version of the Objective-C runtime) that need to be copied into the app bundle in the subdirectory containing the execution script. However, since that directory will not be on the library search path, the script itself also needs updating to fix this. The forthcoming update to Delphi Prism will include these libraries and hide these required updates, but in the meantime

the following steps are required to avoid the thread-local garbage collection issues with Monobjc applications on Snow Leopard.

1. Download the latest version of Monobjc from the Downloads link on <http://monobjc.net> and place it next to the current version in your *C:\Program Files* directory (the downloads are gzipped tarballs, or .tar.gz files, and so need to be un-archived on the Mac using its Stuffit Expander support or on a PC with a suitable Windows tool, such as WinRar). My installation directory is *C:\Program Files\Monobjc-2.0.436.0*, which I'll refer to as \$(MONOBJC)
2. Verify The two shared libraries libmonobjc.1.dylib and libmonobjc.2.dylib are in \$(MONOBJC)\dist
3. Verify the generic execution script AppLoader is in \$(MONOBJC)\src\tools\NAnt.MonobjcTasks\Embedded
4. Add the following post-build events to your project (on the *Build Events* tab of the project properties window). Note the commands are wrapping here but what we have is a pair of copy commands:

```
copy "$(ProgramFiles)\Monobjc-2.0.436.0\dist\libmonobjc.*" ⚡
$(TargetDir)$(TargetName).app\Contents\MacOS
copy "$(ProgramFiles)\Monobjc-2.0.436.0\src\tools\NAnt.MonobjcTasks\Embedded ⚡
\AppLoader" $(TargetDir)$(TargetName).app\Contents\MacOS\$(TargetName)
```

5. Build your Monobjc application in Delphi Prism as usual

Again, this will be made transparent and automatic in the next update to Delphi Prism.

CORRECT CLOSURE

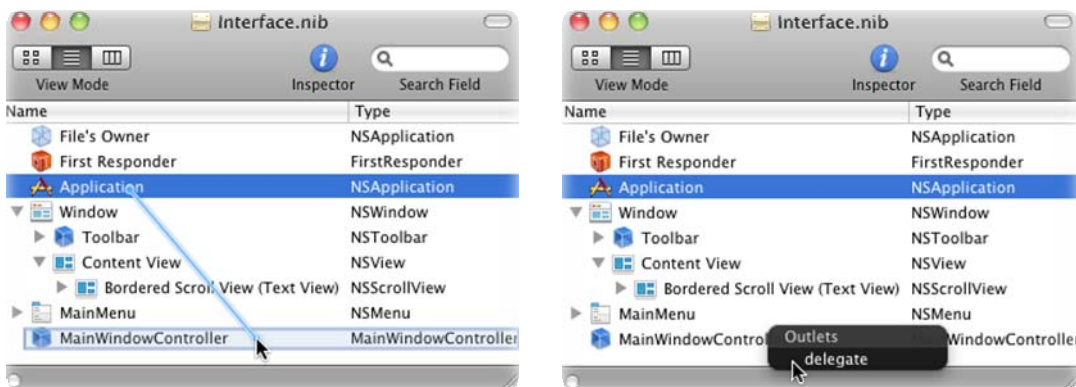
One aspect that seems to be overlooked, even in some of the OS X tools, is a recent change in Apple's *Human Interface Guidelines* specifically for single window applications. You can find the specific guideline by browsing through to *Part III: The Aqua Interface, Windows, Window Behavior, Closing Windows*. This states that when you close the window in a single-window application, the application should quit. Note that this differs from the normal behavior of Mac applications where they continue running with the menu on-screen after all windows are closed. This behavior allows new windows to be created in the application and is similar to that of a Windows MDI container when all child windows are closed, but it doesn't make sense in a single window application, such as this sample text editor.

You can follow the Apple guideline by setting up a delegate for your application object (the `NSApplication` object in the Interface Builder Document window). The delegate can respond to selected messages on behalf of the application object and control these aspects of shutdown. You can look up the `NSApplication` delegate messages either in the Monobjc API documentation (http://api.monobjc.net/html/T_Monobjc_Cocoa_NSApplication.htm), which also has links to the event handler type definitions, meaning you'll know exactly what parameters to define, or in the Cocoa documentation (http://developer.apple.com/mac/library/documentation/Cocoa/Reference/NSApplicationDelegate_Protocol/Reference/Reference.html).

You'll need to handle the `applicationShouldTerminateAfterLastWindowClosed:` message at the very least to indicate that the application should disappear after the window is closed. However, given this application is a text editor (notwithstanding the fact that it can't save or load) you should also handle `applicationShouldTerminate:` and get confirmation from the user that the application should close. You'll also need a delegate for the window object and handle its `windowShouldClose:` message for the same reason. If the user closes the application through the menu, then the application will send a `applicationShouldTerminate:` message. If the user closes the window, then the window will send a `windowShouldClose:` message (and if the response is that the application should close, the application will also send a `applicationShouldTerminate:` message).

Note: In Objective-C a message that has parameters includes a `:` suffix in its name.

The delegate for both the application and the window object in this case will be our main window controller. To set up this delegate relationship you use Interface Builder. Simply `ctrl+drag` from the object (in the Document window) that will be sending the messages (the application or window object) and drop on the delegate object (the main window controller). A blue line will form during the drag operation and when you drop a connection box will pop up listing all the available outlets in the controller. Selecting the currently sole item in the list, `delegate`, sets up the relationship.



With both delegate relationships set up we can look at the code of a Monobjc application. The `Program.pas` file contains the program entry point and executes standard code to kick-start a Monobjc application by loading the Cocoa framework, loading the main `.nib` file and starting off the main event loop.

The controller class, `MainWindowController`, is actually spread across several files thanks to the partial class feature. The one you'll be focusing your attention on mostly is `MainWindowController.pas` where you'll add your action methods as well as helper methods. Another part of it lives in `MainWindowController.Designer.pas`, where a couple of constructor overloads can be found as well as action methods for a couple of system action messages (these methods are just declarations without implementation thanks to them being declared as partial empty methods; you can choose to implement them in the controller class if you see the need). The other partial class definition is found in within the `.nib` bundle, in `Interface.nib\designable.nib\designable.pas`. This partial class will surface the action methods

and outlets you declare in your controller in Interface Builder, and you'll see this happening later.

Note: if you rename the project (or save as a new name) then the namespace in Designable.pas will be automatically updated, but the namespace in the other two files containing partial definitions of your controller class won't change. This means the compiler will now find two separate controller classes with the same name but in different namespaces. It is therefore very important to update your unit namespaces when you rename your project.

Depending on what you have done with the controller in Interface Builder dictates what you might find in the class definition here. You may also find representations of other classes set up (explicitly or implicitly) in Interface Builder (such as `NSFontManager`).

For the current requirement you need to implement three delegate message handling methods (along with any helpers required for the job). Here is some sample code:

```
type
  MainWindowController = public partial class(Monobjc.Cocoa.NSObject)
  private
    method OKtoTerminate: Boolean;
  public
    [ObjectiveCMessage('windowShouldClose:')]
    method WindowShouldClose(window: NSWindow): Boolean;
    [ObjectiveCMessage('applicationShouldTerminate:')]
    method ApplicationShouldTerminate(App: NSApplication): Boolean;
    [ObjectiveCMessage('applicationShouldTerminateAfterLastWindowClosed:')]
    method ApplicationShouldTerminateAfterLastWindowClosed(
      App: NSApplication): Boolean;
  end;

method MainWindowController.OKtoTerminate: Boolean;
begin
  var msgResult: Integer := AppKitFramework.NSRunAlertPanel(
    'Cocoa Text Editor', 'Really quit?', 'No', 'Yes', nil);
  Result := msgResult = NSPanel.NSAlertAlternateReturn;
end;

method MainWindowController.WindowShouldClose(window: NSWindow): Boolean;
begin
  exit OKtoTerminate();
end;

method MainWindowController.ApplicationShouldTerminate(App: NSApplication): Boolean;
begin
  if NSApplication.NSApp.MainWindow = nil then exit True;
  exit OKtoTerminate();
end;

method MainWindowController.ApplicationShouldTerminateAfterLastWindowClosed(App:
NSApplication): Boolean;
begin
  exit True;
end;
```

Note: in order for the world of Objective-C and Cocoa to connect up to Delphi Prism methods (actually any Mono methods) in a Monobjc application, you must mark them with the `ObjectiveCMessage` attribute. You may have noticed that the message handling methods in

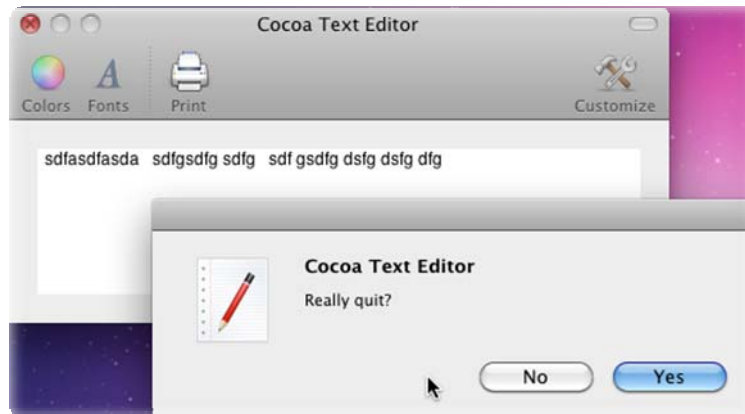
MainWindowController.Designer.pas also use this attribute. Also be very careful to include the : suffix if the original message has parameters otherwise it will be an incorrect message signature.

The logic follows the remit outlined earlier. If the window is closed, the `OKtoTerminate()` helper is called to present a confirmation dialog (alert panel) to the user. The first two parameters of `NSRunAlertPanel()` are a title and some text to draw on the panel. Then you have the default button text, the alternate button text and text for a third button. In this case `No` is the default button and there is no third button. You return `True` or `False` from this method (which is then returned from `WindowShouldClose()`) based on which button was pressed. If `Yes` was pressed, then the window will close.

Whether the window was closed or the user chooses the `quit` menu item on the application menu, `ApplicationShouldTerminate()` will be called. To decide whether you need the confirmation box you check if the application's main window still exists and proceed accordingly.

Note: `NSApplication.NSApp` is a convenient way to get a reference to the application object.

Finally, `ApplicationShouldTerminateAfterLastWindowClosed()` returns `True` to ensure that the application does indeed terminate on cue.



INTERACTING CONTROLS EXAMPLES

To try and get some familiarity with Monobjc and Cocoa we will again run through some simple tutorials on the web, looking at how you can achieve the same effects in Delphi Prism. It's worth noting from the off that you will find in Cocoa tutorials that actions and outlets are defined in code in Xcode, and then sucked into Interface Builder automatically. With Delphi Prism you have to take the opposite approach. Since Interface Builder knows nothing of Delphi Prism files you must manually add outlets and actions in Interface Builder and then have Delphi Prism generate the declarations in



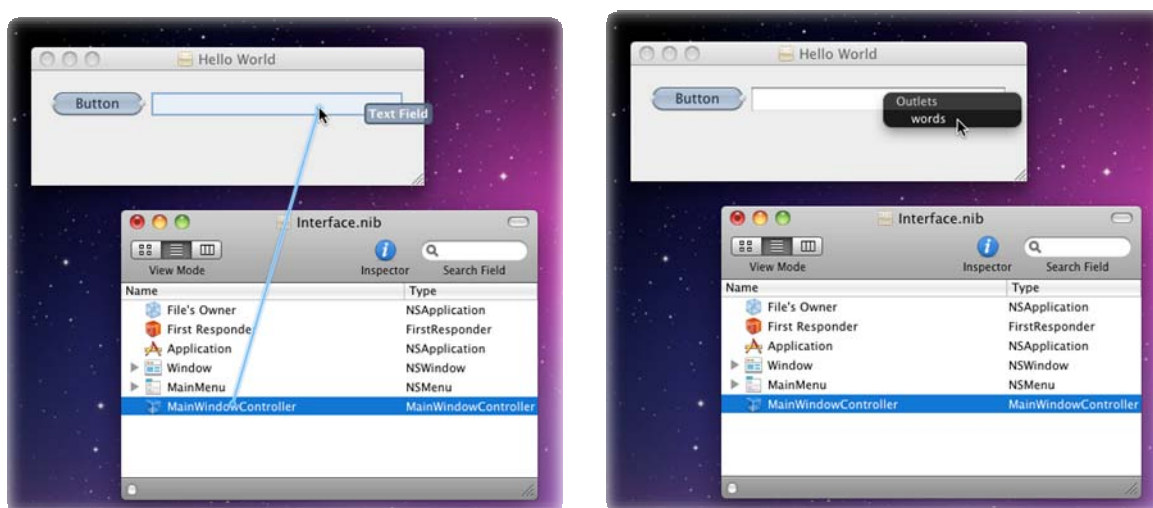
Designable.pas for us using its Visual Studio custom tool; this emits Designable.pas after looking at what's in the .nib files.

The first simple tutorial we'll try and emulate is from The Unix Geek at <http://theunixgeek.wordpress.com/2007/11/11/the-cocoa-tutorial-everyone-needs>. This tutorial just familiarizes us with setting up communication between the UI and the code via a controller; it involves a button writing to a text field.

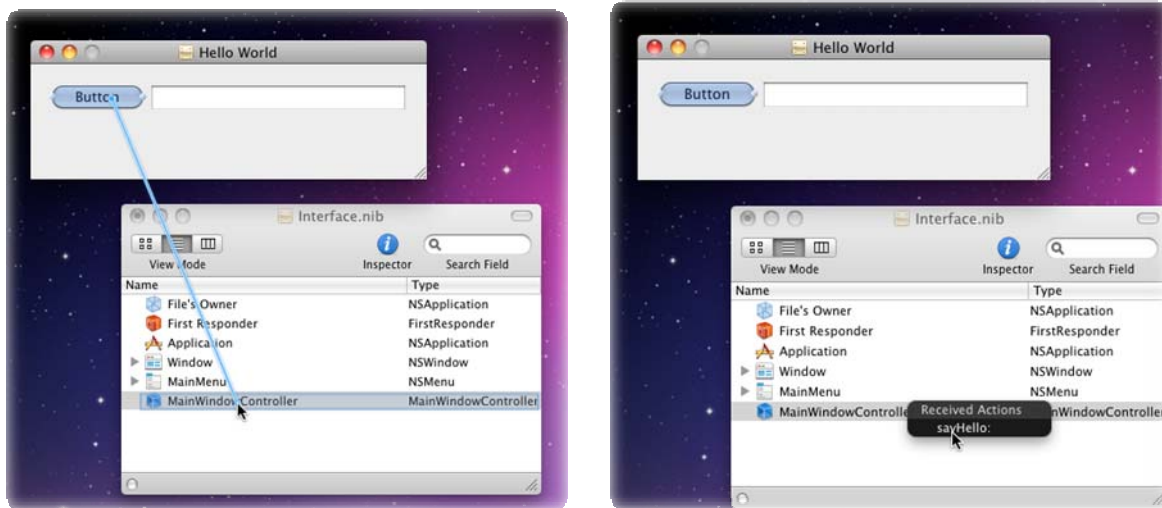
Start a new Monobjc project and in Interface Builder delete the toolbar and add a Button (any of the various types available) and a Text Field from the Library window. You should also update the text of the menu items that contain the application name and remove the Monobjc assembly references that aren't required. Now you need to add an outlet and an action to the controller. The outlet will be connected to the Text Field to allow it to be accessed from the code. The action will be a method that responds to the button being pressed.

To add an outlet to the custom controller go to the *Classes* tab on the Library window and locate the `MainWindowController` class, either by scrolling through the list of classes, or by reducing that list with the search box at the bottom of the window. When selected the lower half of the Library window shows information about the `MainWindowController` class, including its inheritance hierarchy (which is empty as the class simply inherits from `NSObject`), a summary of its definition, and a list of outlets and actions. In the *Outlets* tab use the + button to add an outlet called `words`. In the *Actions* tab add an action called `sayHello:` (don't forget the : suffix).

Now you connect the controller outlet to the Text Field by `Ctrl`+dragging from the controller in the Document window to the Text Field either in the Document window or on the representation of your application window. When you release the mouse a list of potential outlets will be shown and you can select `words`.



To connect the button to the controller's action just `Ctrl`+drag from the button (on the window or on the document) to the controller and release, then select `sayHello:`.



You can see all the connections of the controller summarized on the *Connections* tab of the Inspector if you select the controller in the Document window. In fact you can also set up the connections from here if you prefer. Given a new outlet or action you simply drag the circle on the right side of the window and drop it on the pertinent control on the Document or window.

In the Controller Connections screenshot you can see that the controller is also acting as a delegate for the application object to again enable correct closing behavior when the sole window is closed.



Having changed the UI definition in the .nib it is necessary to run the nib import custom tool to update Designable.pas using the context menu item shown earlier. This produces this updated controller partial class:

```
type
  [ObjectiveCClass]
  MainWindowController = public partial class
    public
      var [ObjectiveCField] words: NSTextField;
      [ObjectiveCMessage('sayHello:')]
      method sayHello(aSender: NSObject); partial; empty;
    end;
```

You can see the words outlet correctly declared as an NSTextField, and the partial declaration of the sayHello() method. Both have appropriate Monobjc attributes to ensure that Objective-C will be aware of them.

All that is left is to implement `sayHello()` in `MainWindowController.pas`. Remember that making changes to `Designable.pas` is quite futile as you will lose any edits next time the custom nib tool is run.

```

type
  MainWindowController = public partial class(Monobjc.Cocoa.NSObject)
  public
    method sayHello(aSender: NSObject); partial;
    [ObjectiveCMessage('applicationShouldTerminateAfterLastWindowClosed:')]
    method ApplicationShouldTerminateAfterLastWindowClosed(
      App: NSApplication): Boolean;
  end;

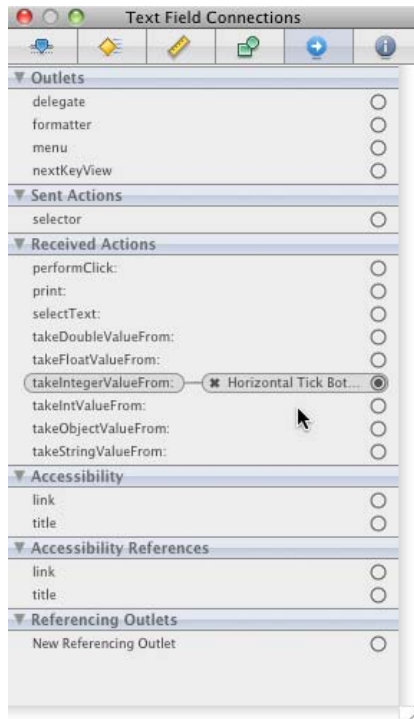
method MainWindowController.sayHello(aSender: NSObject);
begin
  //Give the text field a new value
  words.StringValue := 'Hello world';
  //Ensure it is updated ASAP
  words.NeedsDisplay := True;
  //Make the app speak as well, just for the sake of it
  var Speech: NSSpeechSynthesizer := new NSSpeechSynthesizer();
  Speech.StartSpeakingString(words.StringValue);
end;

method MainWindowController.ApplicationShouldTerminateAfterLastWindowClosed(
  App: NSApplication): Boolean;
begin
  Result := True;
end;

```

As you can see, getting the text field updated is quite straightforward after the outlet connection is made. In addition to writing a new value in the text field the code also speaks the new value, just for sheer extravagance.

This next example comes from a tutorial by Korrupted aka DaxTsurugi at <http://www.insanelymac.com/forum/index.php?showtopic=14778> and shows how the UI can get “additional” functionality without writing code. In this case a slider control will populate a text field with its position. Start a new Monobjc project, delete the toolbar, edit the menu items and remove the unnecessary Monobjc assembly references. Add a Horizontal Slider and a Text Field onto the window. You can set up the slider using the *Attributes* tab of the Inspector. In this case the most important setting is *Continuous*, which means that it will keep send messages to anyone listening as you change the position, rather than waiting until you stop.



To get the functionality, `ctrl+drag` from the slider to the text field and choose the slider's `takeIntegerValueFrom:` received action. What this is saying is when the slider is moved to a new position it tells the text field to set its value to an integer value taken from the slider position. That's it - the example is finished!

The text field has a number of these actions you can trigger, as well as some outlets as you can see to the left here.

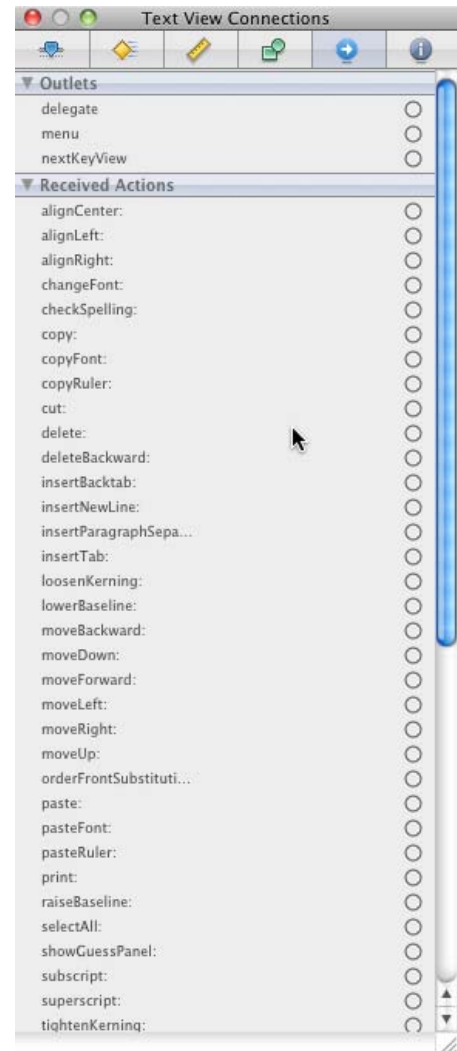
Taking advantage of the functionality offered by

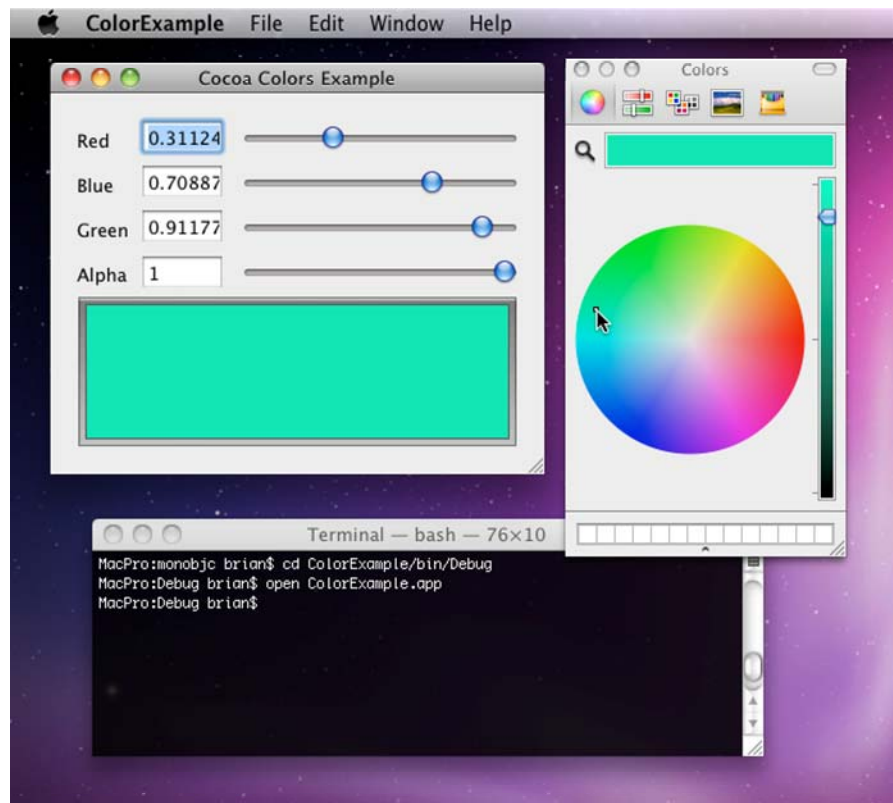
the controls to avoid writing unnecessary code is a useful part of the learning process with Cocoa. The Text View control has many actions you can invoke.

COLOR CHOOSER EXAMPLE

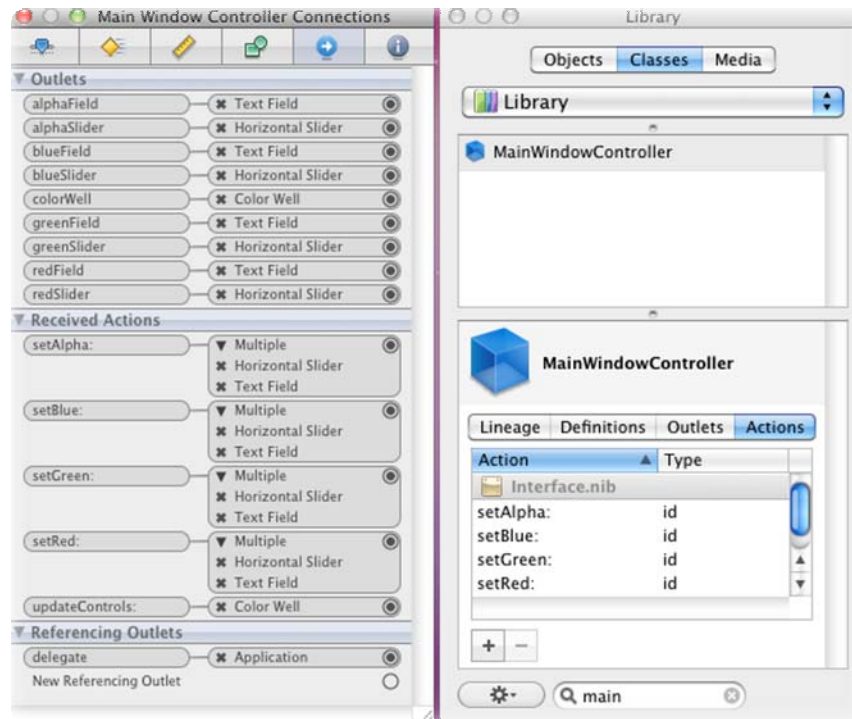
Let's now look at a slightly more ambitious example, using the same outlet and action connection principles just several times over. This example mirrors a tutorial by Michael Beam at

<http://macdevcenter.com/pub/a/mac/2001/06/15/cocoa.html> and builds up a color value editor as shown below. The sliders and the text fields can all update the color shown in the color well control. If a text field is edited, the corresponding slider is updated and vice versa. Clicking the color well control displays a color chooser panel (that's standard behavior for a color well control); choosing a color on the color panel also updates all the controls on the window.





As usual you start with a Monobjc project, remove the toolbar, edit the menus and remove the unnecessary Monobjc assembly references. Then the UI can be laid out: four Labels, four Text Fields, four Horizontal Sliders and a Color Well. Next you need nine outlets and 6 actions declared in the controller class (on the *Classes* tab of the Library window). You can see the names of all the outlets and actions, and also see how they are connected up to the relevant controls in this screenshot and they will all be emitted into *Designable.pas* when you run the custom tool on *Designable.nib*:



The code isn't very exciting, but does the trick. Note that this time you have an implementation of the `AwakeFromNib()` message handler, declared as partial empty in `MainWindowController.Designer.pas`. This executes as soon as the nib has been fully loaded and all the objects defined within have been instantiated.

```
MainWindowController = public partial class(Monobjc.Cocoa.NSObject)
private
    redValue: Single := 0.5;
    greenValue: Single := 0.5;
    blueValue: Single := 0.5;
    alphaValue: Single := 0.5;
    method updateRedUIControls;
    method updateBlueUIControls;
    method updateGreenUIControls;
    method updateAlphaUIControls;
    method UpdateColorWell;
protected
public
    method AwakeFromNib; partial;
    method setRed(aSender: NSObject); partial;
    method setBlue(aSender: NSObject); partial;
    method setGreen(aSender: NSObject); partial;
    method setAlpha(aSender: NSObject); partial;
    method updateControls(aSender: NSObject); partial;
end;
```

```
method MainWindowController.AwakeFromNib;
begin
    updateRedUIControls();
    updateGreenUIControls();
    updateBlueUIControls();
    updateAlphaUIControls();
    UpdateColorWell();
end;

method MainWindowController.setRed(aSender: NSObject);
begin
    redValue := NSControl(aSender).FloatValue;
    updateRedUIControls();
    UpdateColorWell();
end;

method MainWindowController.setBlue(aSender: NSObject);
begin
    blueValue := NSControl(aSender).FloatValue;
    updateBlueUIControls();
    UpdateColorWell();
end;

//Similar code for green & alpha

method MainWindowController.UpdateColorWell;
begin
    colorWell.Color :=
        NSColor.ColorWithCalibratedRedGreenBlueAlpha(
            redValue, greenValue, blueValue, alphaValue);
end;

method MainWindowController.updateControls(aSender: NSObject);
begin
    var color: NSColor :=
        NSColorWell(aSender).Color.ColorUsingColorSpaceName('NSDeviceRGBColorSpace');
    redValue := color.RedComponent;
    blueValue := color.BlueComponent;
    greenValue := color.GreenComponent;
    alphaValue := color.AlphaComponent;
    updateRedUIControls();
    updateGreenUIControls();
    updateBlueUIControls();
    updateAlphaUIControls();
end;

method MainWindowController.updateRedUIControls;
begin
    redField.FloatValue := redValue;
    redSlider.FloatValue := redValue;
end;

method MainWindowController.updateBlueUIControls;
begin
    blueField.FloatValue := blueValue;
    blueSlider.FloatValue := blueValue;
end;

//Similar code for green & alpha
```


COCOA UI TECHNIQUES - ERROR INDICATION BY WINDOW SHAKE

You may have encountered a neat feature in the OS X login screen, whereby if you enter incorrect details it indicates this not with an error box, but by shaking the login window horizontally a few times. If this UI mechanism is good enough for OS X then it's good enough to include in your own applications to indicate certain error states. We don't have any Monobjc examples that involve error states so let's shoe-horn the shaky window idea into the Cocoa text editor from earlier. When the user is asked if they wish to quit, if they say No we'll shake the window to emphasize the point.

Moving the window around in this animated way is achieved with help from the Core Animation framework. The code below is a translation from <http://www.cimgf.com/2008/02/27/core-animation-tutorial-window-shake-effect> and shows a useful comparison between Objective-C Cocoa calls and their Monobjc equivalents.

```
method MainWindowController.ShakeAnimation(frame: NSRect): CAKeyframeAnimation;
const
    numberOfShakes = 4;
    durationOfShake = 0.5;
    vigourOfShake = 0.05;
begin
    //Create an animation, and the define a path for it to follow
    var animation: CAKeyframeAnimation := CAKeyframeAnimation.Animation;
    var shakePath: IntPtr := CGPath.CreateMutable();
    var identityTransform: CGAffineTransform :=
        CGAffineTransform.CGAffineTransformIdentity;
    CGPath.MoveToPoint(shakePath, var identityTransform,
        NSRect.NSMinX(frame), NSRect.NSMinY(frame));
    for I: Integer := 1 to numberOfShakes do
    begin
        CGPath.AddLineToPoint(shakePath, var identityTransform,
            NSRect.NSMinX(frame) - frame.size.width * vigourOfShake, NSRect.NSMinY(frame));
        CGPath.AddLineToPoint(shakePath, var identityTransform,
            NSRect.NSMinX(frame) + frame.size.width * vigourOfShake, NSRect.NSMinY(frame));
    end;
    CGPath.CloseSubpath(shakePath);
    animation.path := shakePath;
    animation.duration := durationOfShake;
    exit animation;
end;

method MainWindowController.ShakeWindow;
begin
    //Add our animation to the animation dictionary, with key 'frameOrigin'
    mainWindow.Animations := NSDictionary.DictionaryWithObjectsAndKeys(
        ShakeAnimation(mainWindow.Frame), new NSString('frameOrigin'), nil);
    mainWindow.Animator.SetFrameOrigin(mainWindow.Frame.origin);
end;

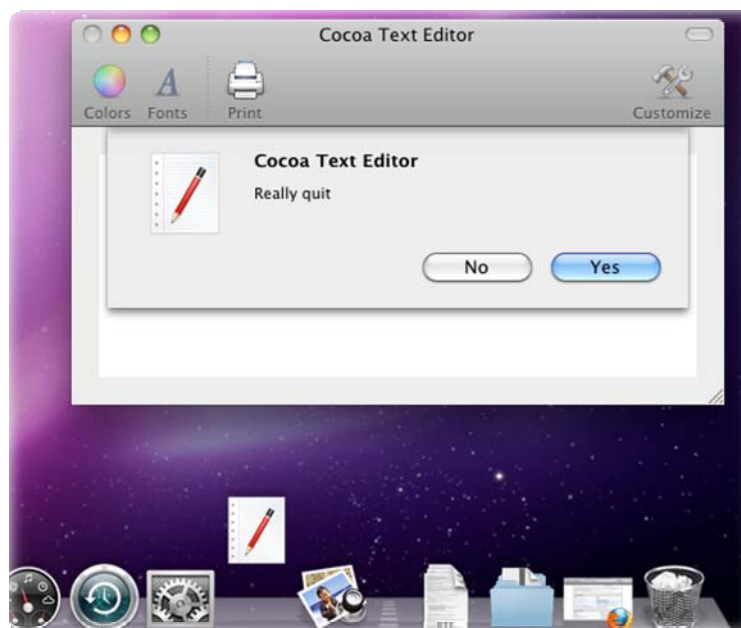
function MainWindowController.OKtoTerminate: Boolean;
begin
    var msgResult: Integer := AppKitFramework.NSRunAlertPanel(
        'Cocoa Text Editor', 'Really quit?', 'No', 'Yes', nil);
    Result := msgResult = NSPanel.NSAlertAlternateReturn;
    //If we are not closing, let's shake the window
    if not Result then
        ShakeWindow();
end;
```

You might note that various global functions in the original code are now static methods. Also, the second parameter to `MoveToPoint()` is declared as a `var` parameter and so the `null` of the original call has to be replaced with a reference to an identity transform in order to perform no actual transform on the move target.

The logic in the code is to create a path that describes the motion you want to impose on the window, which is then applied to an animation. When you want the window to shake you apply the animation to the origin of your window's frame, causing the window to be moved following the animation path.

COCOA UI TECHNIQUES - CONFIRMATION BY SLIDE-IN SHEET

In the *Correct Closure* section on page - 44 - we added in a confirmation process using a popup `NSPanel`. It is actually more common in Mac applications to see the use of sheets that slide in from the top of the window, but serving exactly the same purpose. In the screenshot below you can see an alert sheet in use - notice the Dock icon bouncing up and down to get our attention to this application.



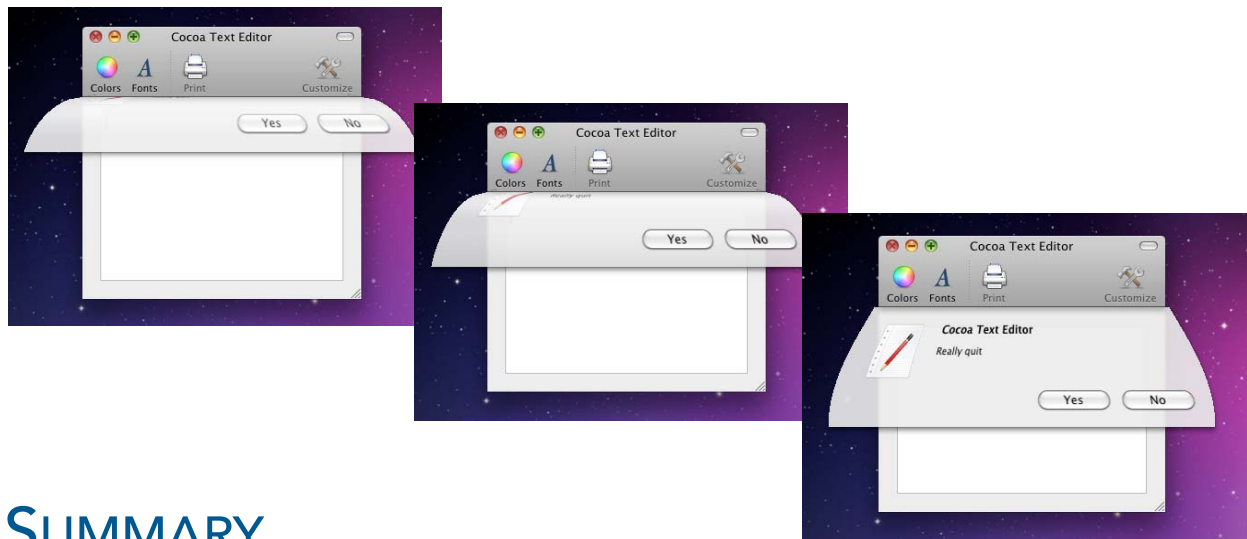
The primary difference between the alert panel and the alert sheet, programming-wise, is that the alert panel pops up, and the `NSRunAlertPanel()` function doesn't return until the user chooses an option and so dismisses the panel. However, with the alert sheet, the call to `NSBeginAlertSheet()` returns immediately. When the user chooses an option on the sheet another method is triggered. This method must be marked with the `ObjectiveCMessage` attribute and be passed to `NSBeginAlertSheet()` as a special type of parameter called a selector. In Objective-C selectors are obtained by using `@selector`. In Monobjc you can call `ObjectiveCRuntime.Selector()`. Here's the modified code:

```
type
  MainWindowController = public partial class(Monobjc.Cocoa.NSObject)
  private
    terminating: Boolean := False;
    method OKtoTerminate: Boolean;
  public
    [ObjectiveCMessage('windowCloseConfirmationDelegate:')]
    method WindowCloseConfirmationDelegate(sheet: NSWindow;
      returnCode: Integer; contextInfo: IntPtr);
  end;

method MainWindowController.OKtoTerminate: Boolean;
begin
  AppKitFramework.NSBeginAlertSheet(
    'Cocoa Text Editor', 'No', 'Yes', nil, mainWindow, Self,
    //Pass delegate selector here to have it called as soon as user presses a button
    nil,
    //Pass delegate selector here to have it react after sheet closes
    ObjectiveCRuntime.Selector('windowCloseConfirmationDelegate:'),
    nil, 'Really quit');
  //The sheet will hang about for a bit, so we'll say "No" for now,
  //and act accordingly when the user shuts the sheet with a button
  exit False;
end;

method MainWindowController.WindowCloseConfirmationDelegate(sheet: NSWindow;
  returnCode: Integer; contextInfo: IntPtr);
begin
  //Now the sheet has been used we can decide whether to terminate the app or not
  if returnCode = NSPanel.NSAlertAlternateReturn then
  begin
    terminating := True;
    NSApplication.NSApp.Terminate(Self);
  end
  else
    ShakeWindow()
  end;
end;
```

A sheet will slide down from the top of the window if the window is wide enough to accommodate it. If it's not, the sheet inflates out in an animated way, which is aesthetically pleasing.



SUMMARY

This white paper has shown how Delphi Prism can be utilized to take existing skills with both the .NET platform and the Delphi language and produce applications that run on Linux and/or Mac OS X. Cross platform development has many issues to take into consideration and there are various levels at which existing code can be re-used.

After reading this paper you should have a good picture of what your options are and the potential of your code base. Where native-looking applications are required, some learning of new GUI toolkits is necessary; however this can pay dividends with careful setup of your architecture, with an eye to maximizing the re-use of UI-less business logic.

ACKNOWLEDGEMENTS

Thanks are due to the following people for help in understanding various technical areas and overcoming various technical hurdles: marc hoffman, Richy King, Carlo Kok, Adrian Milliner and Steve Scott.

Brian Long has spent the last 1.5 decades as a trainer, trouble-shooter and mentor focusing on the Delphi, C# and C++ languages, and the Win32, .NET and Mono platforms. In his spare time Brian is re-discovering and re-enjoying the idiosyncrasies and peccadilloes of Unix-based operating systems. Besides writing a Pascal problem-solving book in the mid-90s, he has contributed chapters to books, written countless magazine articles and acted as occasional Technical Editor for Sybex. Brian has a number of online articles that can be found at <http://blong.com>.

© 2009 Brian Long Consulting and Training Services Ltd. All Rights Reserved.



Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance and accelerate innovation. The company's flagship tools include: Embarcadero® Change Manager™, Embarcadero™ RAD Studio, DBArtisan®, Delphi®, ER/Studio®, JBuilder® and Rapid SQL®. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com.