Best practices when coding EwE6

This page describes standard solutions for recurring oddities and nasties that we have ran into over the years when building EwE6.

THIS SECTION IS UNDER DEVELOPMENT

The following issues are discussed on this page:

Robustness
- Catching unexpected failures (Try / Catch)
- Logging events
- Checking for proper usage (Assertions)
- Creation and disposal of objects and data

Cross-platform compatibility
- Mono
- Target processor

User-interface design guidelines
- Re-use what's already there
- Pop-up messages
- Status feedback
- Styling

Robustness

Take it from us. Users can find very creative ways to use a software product in ways that a programmer could never, ever have predicted. As such, it is of vital importance that your code is robust to failures, even if you do not know in what ways your product may fail. It's rarely a good think to be caught with your pants down, and this section aims to provide a few simple guidelines.

Try / Catch

Make your code robust to failures by encapsulating big function calls in ?Try / Catch blocks. Public methods in public classes are often the entry point to deeper functionality, and placing a Try / Catch inside the public methods is a good way to capture any internal failures within your code. We've done the same in the plug-in framework, where every plug-in point call is wrapped in a Try / Catch block to make sure EwE is not affected by a failing plug-in point or button click.

Take for instance the code that runs in response to a 'Run' button click on the Run Ecosim form:

```
Private Sub OnRun(ByVal sender As System.Object, ByVal e As System.EventArgs) _
  Handles m_btnRun.Click

  Try
     If Not Me.IsRunning Then
        Me.m_iTimeSteps = Me.Core.nEcosimTimeSteps
        Me.m_graph.Refresh()
        Me.Core.RunEcoSim(AddressOf TimeStepFromEcoSim_handler, True)
     End If
  Catch ex As Exception
     cLog.Write(ex, "form RunEcosim.OnRun")
  End Try

End Sub
```

Note that in EwE we prefer not to rely on exceptions for regular error testing. Exception handling is very processor intensive, and a simple ?If / Then test is endlessly faster than bluntly executing code and waiting for the exceptions to happen. As such, we suggest to only use exceptions to capture unforeseen problems.

Logging events

The EwE Log file, created and managed by the core, and stored in the Windows application data folder, contains a track list of important actions taken by EwE in response to user requests and a track record of failures and successes. The purpose of the log is to provide postmortem diagnostics when an error has occurred. You can use the log as well in your plug-ins. Frankly, we think you should.

If you strategically encapsulate Try / Catch blocks, why not write the result of the exception to the log file, as shown here in the Catch part of the try/catch block:

```
Private Sub OnRun(ByVal sender As System.Object, ByVal e As System.EventArgs) _
  Handles m_btnRun.Click

  Try
     If Not Me.IsRunning Then
        Me.m_iTimeSteps = Me.Core.nEcosimTimeSteps
        Me.m_graph.Refresh()
        Me.Core.RunEcoSim(AddressOf TimeStepFromEcoSim_handler, True)
     End If
  Catch ex As Exception
     cLog.Write(ex, "form RunEcosim.OnRun")
  End Try

End Sub
```

You can write anything to the log. The log class, cLog, lives in the EwECore assembly and is worth exploring.

Assertions

Make it a habit to make your code as sound as it can get by testing whether ?your code is properly used. ?Assertions are a neat way to slap yourself on the wrist if built-in assumptions in your code are somehow failing. In EwE, we sometimes refer to these checks as *sanity checks* ;-) In released versions of EwE assertions will not show up, but they are a good tool to check whether your code is used as it is supposed to while developing.

To illustrate, does the following code make sense?

```
Public Function Div(ByVal dividend As Single, ByVal divider As Single) As Single

  'Sanity checks
  Debug.Assert(dividend <> 0, "Cannot divide by zero")

  Return dividend / divider

End Function
```

Compatibility

We want EwE to run on any operating system, and any processor memory model. This section provides some practical tips to keep EwE Operating System-neutral.

Mono

.NET is in theory a system-independent programming platform. .NET applications can be deployed under Unix, Linux, MacOS, etc using runtime environments such as ?Mono. However, not all .NET features work under Mono, most notably the Microsoft.VisualBasic assembly. If you wish to enable your code to run on non-Windows operating systems, do the following:

Remove the reference to Microsoft.VisualBasic from a new VB.NET project as follows:

- Right-click your project in the Solution Explorer and select 'Properties'
- In the References tab, clear the check beside Microsoft.VisualBasic in the 'Imported Namespaces'

The VisualBasic assembly provides features that are backward compatible with Visual Basic 6, the pre-cursor of .NET. We redefined only those the features and constants in the Microsoft.VisualBasic assembly that do NOT have .NET counterparts:

- ?TriState constants

- ?Functions such as IIF, Val

Here is a brief table how Visual Basic 6 constructs map to .NET:

VisualBasic construct	.NET alternative
vbTab, etc	Convert.ToChar?(Keys.Tab), etc
vbNewline	Environment.NewLine
InStr, Left, Mid	.NET ?String class
MsgBox?	See below, User Interface guidelines

Target processor

The EwE source code now fully utilizes 64-bit capabilities. In order to make sure that Windows finds the correct 32 or 64 bit Access drivers make sure you always compile your EwE6 main project against x86 or x64, never against AnyCPU in *Menu* > *Build* > *Configuration Manager*.

Note that 64-bit EwE will not be able to find 32-bit Access database drivers and vice-versa.

User Interface design guidelines

Please adhere to the User Interface Guidelines when building user interfaces.

Resources and localization

Theoretically, the EwE6 scientific interface can be localized to any language. In intention all language-specific elements in EwE are provided in either localizable forms or in localizable resource tables. We have tried to consistently implement this but exceptions may exist; please let us know if you find any.

The ScientificInterfaceShared assembly offers a whack of shared resources, such as strings and images, for plug-ins and the main Scientific Interface to share to reduce the amount of scattered resources that need localizing. In your assembly simply add a statement such as import ScientificInterfaceShared.My.Resources = SharedResources, and access all shared resources on the imported SharedResources thingy.

When you develop your own plug-ins with a user interface, please try to stick to the following resource guidelines:

- Use resources provided in ScientificInterfaceShared when possible,
- Set the 'localizable' property of any forms that you develop to True.

Notifications

Nasty experiences

- Always override *Dispose(bDisposing)* to clean up sub-classed UI elements, do not use *OnHandleDestroyed* because the .NET framework, which wraps Win32 controls, may call this method during the regular life span of a .NET control to do housekeeping. The call may be followed by *OnHandleCreated* - it's simply not a valid trigger to assume your control is dying.
- Note that the Visual Studio designer automagically places a Dispose method in its *.designer.vb files which is blocked from debugging. You may want to manually move this method to your main vb file and strip off *DebuggerNonUserCode* tags that prevent the debugger from stepping through the code.