Clarion Magazine

Clarion News

- » New StrategyOnline Video Tutorials
- » Super Invoice 6.71
- » iQ-XML 2.0 Released
- » J-HTML 1.10
- » SetupBuilder Microsoft Data Access Components (MDAC) 2.8 SP1 Redistributable Update Alert
- » Icetips Build Automator
- » CapeSoft at Aussie DevCon
- » CapeSoft FileExplorer 5
- » gFileFind 2.2 Clarion Source Update
- » gFileFind/QuickSearch 2.0.3
- » Clarion Folk Lore Podcast #3
- » New IDE Parser For Latest Clarion Beta
- » Clarion# Adds Generics
- » New Data Diagrammer Adds Report Views
- » AppGen In Internal Testing
- » J-Html 1.08
- » List and Label Beta 1 & April 18 Deadline
- » New Report Designer
- » gFileFind 2.0.2
- » FinalStep 2.17
- » NeatMessage 2.14
- » DeveloperPLUS Email Address Change
- » SetupBuilder 6.7 April 2008 Update
- » gFileFind 2.0 Search Utility And Clarion Source/Template
- » Clarion 6 Accounting Code
- » SimGlobalButtons
- » J-Html 1.07
- » 1st Logo Design April Deals
- » vuSendKeys 1.4

Latest Subscriber Content

Creating A Custom Rounded Panel Control In Clarion#

Early on in his explorations of Clarion# Randy Rogers decided he needed a panel control with rounded corners. The problem: no such panel is part of the standard control set. The solution: create a custom panel control.

Posted Wednesday, April 30, 2008

RTF Mail Merge

It's easy to do mail merge with MS Word and an appropriate third party utility. But what if your users don't all have MS Word? Not to worry; as John Griffiths shows you can always use the RTF Text Control.

Posted Wednesday, April 30, 2008

Passing Filter Expressions To Reports

Dynamic filter expressions are a great way to let your users query data, but they'll probably want to apply these filters to reporting as well. Paul Blais shows how it's done.

Posted Thursday, April 17, 2008

Loosely Coupled .NET Applications And Inversion of Control

Designing loosely-coupled applications is one thing; getting all those components to work together, based on runtime configuration settings, is another. The answer: use an Inversion of Control container to manage the creation and lifespan of your objects.

Posted Thursday, April 17, 2008

Loosely Coupled .NET Applications: The Baseline Example

In preparation for the grand finale on configurable, adaptable .NET applications David Harms introduces a sample application and shows how to use its classes in typical tightly-coupled fashion.

Posted Wednesday, April 16, 2008

Using Dynamic Filter Expressions for QBE

Paul Blais takes a look at filter expressions employed in the Clarion VIEW structure. These expressions are a close cousin to EVALUATE, and a great way to enhance your users' ability to query the database.

Posted Friday, April 11, 2008

Loosely Coupled .NET Applications: Understanding Reflection

At the midway point in his series on flexible application design David Harms explores the very cool world of .NET reflection.

Posted Thursday, April 10, 2008

Source Code Library 2008.03.31 Available

The Clarion Magazine Source Code Library has been updated to include the March source. Source code subscribers

- » OfficeInside Updated
- » Clarion Domains For Sale
- » Video Tutorials At StrategyOnline
- » J-Html 1.03
- » Huenuleufu's Support Pack Includes FileTuner
- » PrintWindow 1.20
- » FullRecord 2.09 and 1.87
- » Clarion Version Switcher 1.1.8
- » Clarion Folk Lore Podcasts
- » Clarion# Calculator
- » gCalc 5.1 Clarion Calculator
- » vuSendKeys 1.4 Pre-Release Available
- » vuLimiter 2.01 Minor Update

[More news]

- » Clarion.NET FAQ
- » Clarion# Language Comparison
- » Creating A Custom Rounded Panel Control In Clarion#

[More Clarion & .NET]

[More Clarion 101]

Latest Free Content

- » Source Code Library 2008.03.31 Available

[More free articles]

Clarion Sites

Clarion Blogs

can download the Mar 2008 update from the My ClarionMag page. If you're on Vista please run Lindersoft's Clarion detection patch first.

Posted Wednesday, April 02, 2008

[Last 10 articles] [Last 25 articles] [All content]

Source Code

The ClarionMag Source Code Library

Clarion Magazine is more than just a great place to learn about Clarion development techniques, it's also home to a massive collection of Clarion source code. Clarion subscribers already know this, but now we've made it easier for subscribers and non-subscribers alike to find the code they need.

The Clarion Magazine Source Library is a single point download of all article source code, complete with an article cross-reference.

More info • Subscribe now

Printed Books & E-Books

E-Books

E-books are another great way to get the information you want from Clarion Magazine. Your time is valuable; with our e-books, you spend less time hunting down the information you need. We're constantly collecting the best Clarion Magazine articles by top developers into themed PDFs, so you'll always have a ready reference for your favorite Clarion development topics.

Printed Books

As handy as the Clarion Magazine web site is, sometimes you just want to read articles in print. We've collected some of the best ClarionMag articles into the following print books:

- » Clarion Tips & Techniques Volume 4 - ISBN 978-0-9784034-09
- » Clarion Tips & Techniques Volume 3 - ISBN: 0-9689553-9-8
- » Clarion 6 Tips & Techniques Volume 1 - ISBN: 0-9689553-8-X
- » Clarion 5.x Tips and Techniques, Volume 1 - ISBN: 0-9689553-5-5
- » Clarion 5.x Tips and Techniques, Volume 2 - ISBN: 0-9689553-6-3
- » Clarion Databases & SQL - ISBN: 0-9689553-3-9

We also publish Russ Eggen's widely-acclaimed Programming Objects in Clarion, an introduction to OOP and ABC.

From The Publisher

About Clarion Magazine

Clarion Magazine is your premier source for news about, and in-depth articles on Clarion software development. We

publish articles by many of the leading developers in the Clarion community, covering subjects from everyday programming tasks to specialized techniques you won't learn anywhere else. Whether you're just getting started with Clarion, or are a seasoned veteran, Clarion Magazine has the information *you* need.

Subscriptions

While we do publish some free content, most Clarion Magazine articles are for subscribers only. Your subscription not only gets you premium content in the form of new articles, it also includes all the back issues. Our search engine lets you do simple or complex searches on both articles and news items. Subscribers can also post questions and comments directly to articles.

Satisfaction Guaranteed

For just pennies per day you can have this wealth of Clarion development information at your fingertips. Your Clarion magazine subscription will more than pay for itself - you have my personal guarantee.

Dave Harms

ISSN

Clarion Magazine's ISSN

Clarion Magazine's International Standard Serial Number (ISSN) is 1718-9942.

About ISSN

The ISSN is the standardized international code which allows the identification of any serial publication, including electronic serials, independently of its country of publication, of its language or alphabet, of its frequency, medium, etc.

Copyright © 1999-2008 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited.

Clarion Magazine

Clarion News

Search the news archive

Helderberg Clarion User Group Meeting

The next Helderberg Clarion User Group Meeting (HCUG) in the Western Cape, South Africa, will be on May 6, 2008, 17:30.) Cover charge is R30; venue is the Trustco Group Office at Unit 304 Oakmont Building,Somerset Links Office Park (off De Beers Ave), Somerset West. Topics include Clarion 7, Clarion 6 Tips, Questions and answers, productivity tools and tips.

Posted Monday, May 05, 2008

Wingnut Holiday Schedule

The Wingnut Solutions office will be closed from May 4th to May 27th. Support may be delayed for several days during this time. All products remain available for immediate purchase.

Posted Friday, May 02, 2008

SetupBuilder 6.7 May 2008 Update

Lindersoft has released the SetupBuilder 6.7 May 2008 Update. This release is available, free of charge, to all SetupBuilder customers who have an active SetupBuilder maintenance subscription plan. To get the latest product version, select "Check for Updates" from within the SetupBuilder 6 IDE. To get the latest documentation, select "Check for Documentation Updates" from within the SetupBuilder 6 IDE.

Posted Friday, May 02, 2008

CPCS Support Schedule

CPCS Support will be unavailable from May 5-12, 2008. Larry will handle any support issues that arise during that time (including providing install codes for new purchases) immediately upon his return.

Posted Friday, May 02, 2008

CHT 12B1.00

Clarion Handy Tools build 12B1.00 is now available for download.

Posted Friday, May 02, 2008

Polar Control Acquires gCal, gNotes, gReg

Polar Control Inc. has acquired gCal, gNotes and gReg from Wingnut Solutions Inc. The first release will be an updated version of gCal, followed by gNotes and then gReg.

Posted Friday, May 02, 2008

New StrategyOnline Video Tutorials

Several new video tutorials are available on the StrategyOnline site.

Posted Tuesday, April 29, 2008

Super Invoice 6.71

Super Invoice 6.71 is available for download. This is a major upgrade, featuring:; Support for FileDrops and FileDropCombos; Option to use a form for all edits, when EIP isn't desired, but you want the in-memory processing and transaction support; Ability to invoke EIP mode is even more intuitive (e.g.: toggle checkbox with single click, without entering EIP mode first); Significantly improved transaction processing and options; Popup menu support; Many more methods for hooking into the underlying classes; Support for local variables as EIP fields; Improved documentation; Compatible with Clarion 7-Beta. Next up is Super Browse (within a week), followed by Super Import-Export.
Posted Tuesday, April 29, 2008

iQ-XML 2.0 Released

New in iQ-XML 2.0: Many changes were made to the core Parser functions to make them faster; Added SoapHeaderOnly option to Load functions; Created and compiled for Clarion Versions 5.5, 6.1, 6.2 and 6.3; More text/information in online help. iQ-XML is a free tool for Clarion developers to add XML functionality to their applications with very little knowledge. It offers many features not found in Clarion's own XML functions. iQ-XML comes with both Parser and Writer functions.
Posted Tuesday, April 29, 2008

J-HTML 1.10

New features in J-HTML 1.10 include (but are not limited to): Replace the WebBrowser control's popup menus with our Clarion popups; Design custom print preview screens; Save the loaded HTML document, specifying a filename; Hide the control's scrollbars.
Posted Tuesday, April 29, 2008

SetupBuilder Microsoft Data Access Components (MDAC) 2.8 SP1 Redistributable Update Alert

Lindersoft has updated the Microsoft Data Access Components (MDAC) 2.8 SP1 Redistributable for SetupBuilder 6 Developer Edition. The redistributable can install Microsoft Data Access Components (MDAC) 2.8 SP1. This revised package replaces the previously released redistributable package and comes with an updated "rt_mdac28_x86.sbi" include script. To update your *existing* Microsoft Data Access Components (MDAC) 2.8 SP1 Redistributable: 1. Select Help -> Redistributable Manager...; 2. In the Installed Runtime Package(s) list select the "Microsoft Data Access Components (MDAC) 2.8 SP1 Redistributable" item and click the "Remove Runtime" button to remove the package; 3. In the Available Runtime Package(s) list select the "Microsoft Data Access Components (MDAC) 2.8 SP1 Redistributable" and click the "Install Runtime" button to install the new redistributable package.
Posted Tuesday, April 29, 2008

Icetips Build Automator

Icetips Creative, Inc. has released the Build Automator, a new utility program for developers that saves time and simplifies building new software installations. With strong support for Clarion developers and SetupBuilder users, the Build Automator makes it easy to create fully automated scripts to build a completely new install of your software. This includes compiling Clarion applications, copying files and compiling SetupBuilder scripts. Build Automator supports both the new Clarion7/Clarion.NET IDE, through the use of MS-Build, and the Clarion 6 (and older) IDE. The Build Automator also supports other popular developer suites such as Visual Studio and Delphi. It also natively supports Inno Setup, which is a popular freeware install software. Native support for InstallShield and Wise will be added later. There is no scripting language to learn and no programming skills required. The Build Automator has the ability to be extended

with plugin dlls for those who may wish to do so. (This option available only in the Developer Edition.) You can also use the Build Automator to automate any other repetitive tasks, such as copying files for backup purposes. The Build Automator can run any program you want, and you can specify parameters you may need to pass to the program. The Build Automator is currently in beta until June 1st, when it will be released as the 2008 Edition. Until then, the full Developer Edition with one year of maintenance is being offered for $99.00, which is 50% off the regular price of $199. The Build Automator is available for download as a 10 day unlimited demo.
Posted Tuesday, April 29, 2008

CapeSoft at Aussie DevCon

Bruce Johnson will be present at the Australian Devcon in May 2008 and will be presenting two sessions. The first is a technical session about using Clarion VIEW's (especially in embed code) and the second is a business session on Customer Support. This is the third part of a three part session on Good Business Practices. In addition to the main event (running from May 30 to June 1) Bruce will be hosting a workshop on Thursday (May 29) and Friday morning (May 30). The goal of this time will be to assist attendees with their own projects, demonstrating possible improvements, assisting with implementations, and helping with any problems. There is a nominal cost to cover the venue and lunch so please contact Geoff Spillane (initial plus last name, all lower case, at clarion dot net dot au) if you're interested in attending this workshop.
Posted Tuesday, April 29, 2008

CapeSoft FileExplorer 5

CapeSoft FileExplorer lets you embed a whole range of document types directly in your application. Embed a Web Browser on your window. Play all music, and video files. View PDF's. Edit Web pages. FileExplorer 5 features include: A brand new HTML Editor (the old editor is still supported, but deprecated); A brand new feMedia2 class (replaces the old feMedia class); A brand new feFile class that provides file loading and saving using the Windows API, as well as providing functions to manage file names and paths. The upgrade (from an old version of FileExplorer) will be on sale for $69 until 1 May 2008, when it will increase to $99 during the duration of the beta period. New users may purchase FileExplorer at the old FileExplorer price ($149) until 1 May 2008. On 1 May 2008, FileExplorer will cost $199. If you purchased FileExplorer after 1 April 2007, you get the upgrade at no cost. To apply for a free upgrade simply add in the comments field "Application for free upgrade".
Posted Tuesday, April 29, 2008

gFileFind 2.2 Clarion Source Update

New in gFileFind version 2.2: Issue with text searches over 255 byte boundary resolved; Added DISPLAY(gffPath) to File lookup accepted embed; Performance enhancements (thanks to Geoff Robinson); Fixed an issue with ?Image1 control and silent mode. Includes a copy of QuickSearch.
Posted Tuesday, April 29, 2008

gFileFind/QuickSearch 2.0.3

gFileFine, the standalone product, has been renamed QuickSearch to prevent confusion with gfileFind the Clarion add-on.
Posted Tuesday, April 29, 2008

Clarion Folk Lore Podcast #3

The third Clarion Folk Lore Podcast is now available for your listening pleasure.
Posted Thursday, April 17, 2008

New IDE Parser For Latest Clarion Beta

The latest Clarion beta (build 3276) contains a completely rewritten IDE parser for faster overall IDE performance, lower memory usage, and improved code completion, among other features.
Posted Thursday, April 17, 2008

Clarion# Adds Generics

As noted in the blog post, SV released limited support for generic types in the previous beta build. As of 3276 support for generics is fully established.
Posted Thursday, April 17, 2008

New Data Diagrammer Adds Report Views

The latest release of the new IDE (build 3276) contains a report views feature in the new data diagrammer. Reports let you view all files, keys, tables, relations and triggers in your dictionary and you can sort by the various attributes of each.
Posted Thursday, April 17, 2008

AppGen In Internal Testing

SV has provided a status update on the new Application Generator, now undergoing internal testing. A "good percentage" of applications are converting, but conversion bugs are still being squashed and the AppGen isn't ready for beta release yet. Other support components such as the window and report designers and the source editor are also undergoing testing.
Posted Thursday, April 17, 2008

J-Html 1.08

J-Html 1.08 includes 57 pages of documentation.
Posted Thursday, April 17, 2008

List and Label Beta 1 & April 18 Deadline

List and Label Beta 1 is now available. April 18th, 2008 is the last day to get the version 13 upgrade. You will need your valid serial number to get the upgrade (enter it in the comments section of the order form). This data is sent to Combit and verified. You will then get another email with download instructions. What you get from the RadFusion web site is the beta 1 templates, docs and examples. List and Label itself is not part of the install, you get that from Combit via RadFusion. Also the prices are the same as can found on Combit's site except you also get the templates.
Posted Thursday, April 17, 2008

New Report Designer

ClarionFolk has screen shots of the new report designer by Steve Ryan & Co.
Posted Thursday, April 17, 2008

gFileFind 2.0.2

gFileFind 2.0.2 has been released. Changes include: Changed some help file contents; Fixed an issue with moving files; Fixed an issue with the destination directory not being shown when selecting a directory; Fixed an issue with using Shift-Q; Fixed issues with CRC32 calculations.
Posted Thursday, April 17, 2008

FinalStep 2.17

FinalStep 2.17 has been released. This includes a fix for the legacy template where process procedures were not handled.
Posted Thursday, April 17, 2008

NeatMessage 2.14

NeatMessage 2.14 is available for download. New features include: Background color to highlight selected messages; Bug fix for enhanced compatibility when ok button is specified as "0"; Compatibility fix for FullRecord variable not recognized; Legacy template fix - some embeds were still generated even if template was disabled.
Posted Thursday, April 17, 2008

DeveloperPLUS Email Address Change

CustomerService@ is no longer a valid email address for DeveloperPLUS. It has been retired due to a sudden and immense deluge of rejection notices from forged spam. The replacement address is problems@.
Posted Friday, April 11, 2008

SetupBuilder 6.7 April 2008 Update

Lindersoft has released the April 2008 update of Setupbuilder 6.7. This release is available, free of charge, to all SetupBuilder customers who have an active SetupBuilder maintenance subscription plan. This is an update to the latest stable release, and contains some bug fixes and improvements. To get the latest product version, select "Check for Updates" from within the SetupBuilder 6 IDE. To get the latest documentation, select "Check for Documentation Updates" from within the SetupBuilder 6 IDE.
Posted Friday, April 11, 2008

gFileFind 2.0 Search Utility And Clarion Source/Template

The gFileFind 2.0 Search Utility is now offered as a standalone utility or as a utility with source code. Changes include: Rebranded; Moved API source from a separate DLL/LIB app to INC/CLW files; Adjusted template to work with INC/CLW files and added appropriate DOS/ASCII drivers; Moved global variables into a group; Changed function name to allow for passing of queue and settings; Queue and settings structures can now be set from the code template to allow for local result queue/settings; Changed Icon names for better uniqueness; Added reference variables for using custom progress windows; Modified application to use local variables instead of global for easier importing of procedures; Moved options from separate window to main window; Added several confirmation dialogs to the application; Added MaxDisplay parameter to setup max display of filenames during search; Changed rename dialog to show current filename. Demo available. Utility plus source code is US$39.
Posted Friday, April 11, 2008

Clarion Magazine

Creating A Custom Rounded Panel Control In Clarion#

by Randy Rogers

Published 2008-04-30

I have spent a bit of time over the last few months familiarizing myself with Clarion# and the whole .NET environment. Armed with my new found knowledge I decided to take a small Clarion 6 invoicing application and port it to Clarion#. My goal was to create a "standard" Clarion menu, browse, form, report type of application in Clarion#. This is the first in a series of articles in which I will share much of what I discovered along the way.

Early in the project I decided I wanted to have panels with rounded corners on my forms. There is no RoundCornerPanel control in the .NET library, so to get what I wanted it became necessary to derive a new control from the Panel control. In Clarion 6 it's difficult to enhance existing controls; .NET makes this task much simpler. In this article I will present the information necessary to derive your own control from an existing .NET control.

Namespaces

There were several false starts to the project as I tried different things and made more discoveries further down the path. One of my early enlightenments was the need to organize my assemblies (assembly is the .NET term for DLLs and EXEs) so that they could be reusable. Ultimately I looked at how the .NET and Clarion# libraries were organized and decided to follow a similar pattern when creating my assemblies. I use Keystone as my base namespace and for User Interface controls I use a UIControls subordinate namespace. So, the name for the assembly will be Keystone.UIControls.

I begin by creating a new Clarion.Net/Windows Applications/Class Library Solution. You might be tempted to select the Windows User Control Library Quick Start as the solution type. That Quick Start has a different purpose which I will cover in my next article.

My starting point looks like Figure 1.

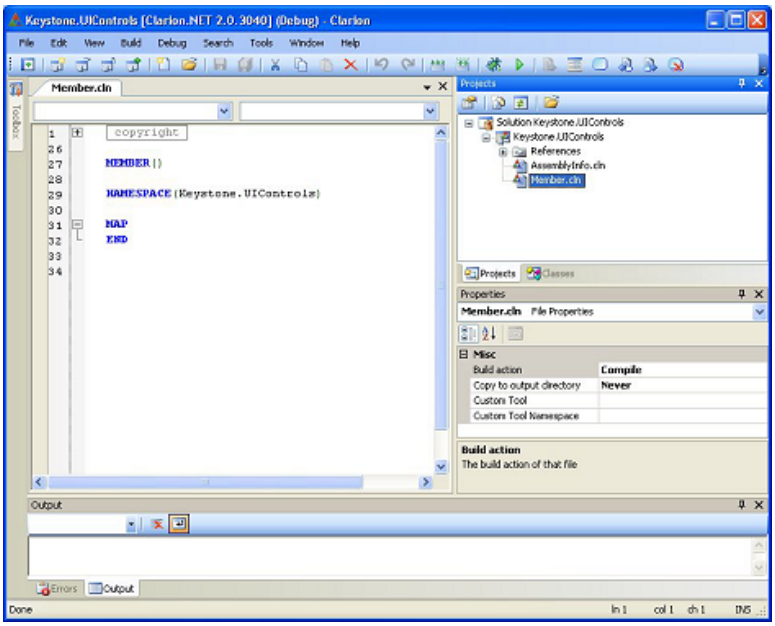

Figure 1. The initial UIControls project (view full size image**)**

The Quick Start template creates a default Member.cln file for the project. First, I want to rename Member.cln in the Projects pad to RoundedPanel.cln. In build 3040, the release I used for this project, renaming caused an IDE exception; I had to do a File|Save As and then remove the old file and add the new one. As of build 3276 this bug is fiixed. In any case after renaming the file my project looks like Figure 2.

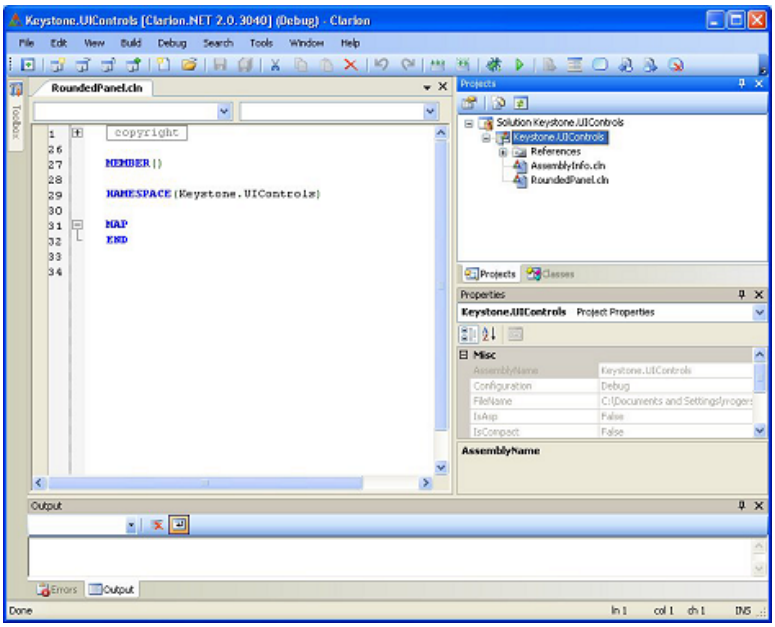

Figure 2. The renamed RoundedPanel file (view full size image**)**

To create the RoundedPanel control I need to derive from the Panel control class, and I will also need to use the OnPaint method to draw the border of the panel with rounded corners. I declare my class like this:

```
RoundedPanel   CLASS(System.Windows.Forms.Panel),|
               TYPE,PUBLIC,NETCLASS,PARTIAL
Construct       PROCEDURE(),PUBLIC
OnPaint         PROCEDURE(System.Windows.Forms.PaintEventArgs e)|
               ,PROTECTED,DERIVED
           END
```

Next I create stubs for the methods:

```
RoundedPanel.Construct  PROCEDURE()
  CODE
    RETURN
RoundedPanel.OnPaint PROCEDURE(System.Windows.Forms.PaintEventArgs e)
  CODE
    RETURN
```

I press the Build button just to make sure I haven't made any mistakes yet... success!

At this point I add some USING statements after the MAP to declare the additional namespaces I want to reference.

```
USING(System)
```

```
  USING(System.ComponentModel)
  USING(System.Drawing)
  USING(System.Runtime.InteropServices)
  USING(System.Windows.Forms)
```

I also want to add a few custom properties to my class so that I can control the color and width of the border as well as the size of the corners themselves:

```
m_CornerSize      System.Int32
[Category('Appearance')]
[Description('The size of the Corner')]
CornerSize        PROPERTY,System.Int32,PUBLIC
m_BorderColor     System.Drawing.Color
[Category('Appearance')]
[Description('The color of the Border')]
BorderColor       PROPERTY,System.Drawing.Color,PUBLIC
m_BorderWidth     System.Int32
[Category('Appearance')]
[Description('The width of the Border')]
BorderWidth       PROPERTY,System.Int32,PUBLIC
```

And I create the associated methods:

```
RoundedPanel.get_CornerSize   PROCEDURE()
  CODE
    RETURN SELF.m_CornerSize

RoundedPanel.set_CornerSize   PROCEDURE(System.Int32 value)
  CODE
    SELF.m_CornerSize = value
    RETURN

RoundedPanel.get_BorderColor   PROCEDURE()
  CODE
    RETURN SELF.m_BorderColor

RoundedPanel.set_BorderColor   PROCEDURE(System.Drawing.Color value)
  CODE
    SELF.m_BorderColor = value
    RETURN

RoundedPanel.get_BorderWidth   PROCEDURE()
  CODE
    RETURN SELF.m_BorderWidth

RoundedPanel.set_BorderWidth   PROCEDURE(System.Int32 value)
  CODE
```

```
    SELF.m_BorderWidth = value
```

Note that I'm using a naming convention of set_*varname* and get_*varname* which is one of the two standard Clarion# syntaxes for property getters and setters. The [Category] and [Description] attributes tell the IDE how to display these properties in the property pad.

Since I don't want the standard Panel border I set the Panel BorderStyle property to BorderStyle.None in the Construct method. I also set the default values for my custom properties:

```
RoundedPanel.Construct      PROCEDURE()
  CODE
    SELF.BorderStyle = BorderStyle.None
    SELF.CornerSize = 10
    SELF.FrameColor = System.Drawing.Color.Black
    SELF.FrameWidth = 1
    RETURN
```

To actually draw the border on the panel I will be using methods from the System.Drawing namespace. First I will use MSDN library descriptions to briefly explain the methods used and then I'll present the code to draw the border. From MSDN:

> A **GraphicsPath** represents a series of connected lines and curves.
>
> The **GraphicsPath.StartFigure** method starts a new figure without closing the current figure. All subsequent points added to the path are added to this new figure.
>
> The **GraphicsPath.AddArc** method appends an elliptical arc to the current figure. If there are previous lines or curves in the figure a line is added to connect the endpoint of the previous segment to the beginning of the arc. The arc is traced along the perimeter of the ellipse bounded by the specified rectangle. The starting point of the arc is determined by measuring clockwise from the x-axis of the ellipse (at the 0-degree angle) by the number of degrees in the start angle. The endpoint is similarly located by measuring clockwise from the starting point by the number of degrees in the sweep angle. If the sweep angle is greater than 360 degrees or less than -360 degrees, the arc is swept by exactly 360 degrees or -360 degrees, respectively.
>
> The **GraphicsPath.CloseFigure** method closes the current figure and starts a new figure. If the current figure contains a sequence of connected lines and curves, the method closes the loop by connecting a line from the endpoint to the starting point.

Here is the first part of the OnPaint method for the RoundedPanel class. This part creates the path for the panel border; in the next part I actually paint the border:

```
RoundedPanel.OnPaint    PROCEDURE(System.Windows.Forms.PaintEventArgs e)
rect        Rectangle
p           GraphicsPath
BorderPen    Pen
widenPen    Pen
  CODE
  p = NEW Drawing2D.GraphicsPath()
  p.StartFigure()
  !top left corner
  rect.X = 0
  rect.Y = 0
  rect.Width = SELF.cornerSize
  rect.Height = SELF.cornerSize
  p.AddArc(rect, 180, 90)
```

```
!top of panel
p.AddLine(SELF.cornerSize, 0, SELF.Width - SELF.cornerSize, 0)
!top right corner
rect.X = SELF.Width - SELF.cornerSize
rect.Y = 0
rect.Width = SELF.cornerSize
rect.Height = SELF.cornerSize
p.AddArc(rect, 270, 90)
!right side of panel
p.AddLine(SELF.Width, SELF.cornerSize, SELF.Width, SELF.Height - SELF.cornerSize)
!bottom right corner
rect.X = SELF.Width - SELF.cornerSize
rect.Y = SELF.Height - SELF.cornerSize
rect.Width = SELF.cornerSize
rect.Height = SELF.cornerSize
p.AddArc(rect, 0, 90)
!bottom of panel
p.AddLine(SELF.Width - SELF.cornerSize, SELF.Height, SELF.cornerSize, SELF.Height)
!bottom left corner
rect.X = 0
rect.Y = SELF.Height - SELF.cornerSize
rect.Width = SELF.cornerSize
rect.Height = SELF.cornerSize
p.AddArc(rect, 90, 90)
!left side of panel
p.CloseFigure()
```

The Region property is a collection of pixels within the window where the operating system permits drawing. The operating system does not display any portion of a window that lies outside of the window region. The coordinates of a control's region are relative to the upper-left corner of the control, not the client area of the control.

The GraphicsPath.Widen method creates an outline around the original lines in a GraphicsPath object, with a distance between the existing lines and the new outline equal to that of the width of the Pen object used in the call to Widen. If you want to fill the space between the lines you must use the FillPath method rather then the DrawPath method.

To complete the OnPaint method I create a region from the graphics path, then set that as the control region and call the Parent.OnPaint method to let the parent Panel class paint the control. Next I create a pen (I used green but the colour does not matter since I am going to use the DrawPath method) and widen the path by two pixels so I don't overlay anything already painted. Finally, I create a pen of the appropriate color and use the DrawPath method to draw the rounded border.

```
SELF.Region = NEW Region(p)
PARENT.OnPaint(e)
widenPen = NEW Pen(color.Green,2)
p.Widen(widenPen)
!Create a pen to draw the frame with the specified colour and width
framePen = NEW Pen(SELF.BorderColor, SELF.BorderWidth)
!Set the Graphics smothingMode to antialias
e.graphics.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias
!Use the framePen to draw the panel frame
```

```
e.graphics.DrawPath(framePen, p)
RETURN
```

Crossing my fingers I press the Build button... success! Now it's time to test my RoundedPanel control to see if it really works; I close this solution and create a new Windows Form project called RoundPanelTest. With the MainForm.cln file open in design mode I right click on the Tools Sidebar and select Configure Sidebar... from the context menu. I click New and create a new category called Clarion Magazine. I select the Clarion Magazine category in the list box and then press the Add Components button. I select the Custom tab then press the ellipsis button to search for the Keystone.UIControls.dll I just created. I find it in the Keystone.UIControls /bin/debug folder, select it, and then press the Show Components button. There is my RoundedPanel control already selected. I press OK and I'm back to the window designer.

Selecting the Clarion Magazine category on the Tool Sidebar allows me to select the RoundedPanel control and drag it onto my form. I can set properties, including my custom ones, and use the RoundedPanel control anywhere I might otherwise use the standard Panel control. As I change the control's properties the IDE executes my code to redisplay the control, so it always looks the same in the IDE as it will in my running application. Figure 3 shows the control in the IDE; Figure 4 shows the control in the application.

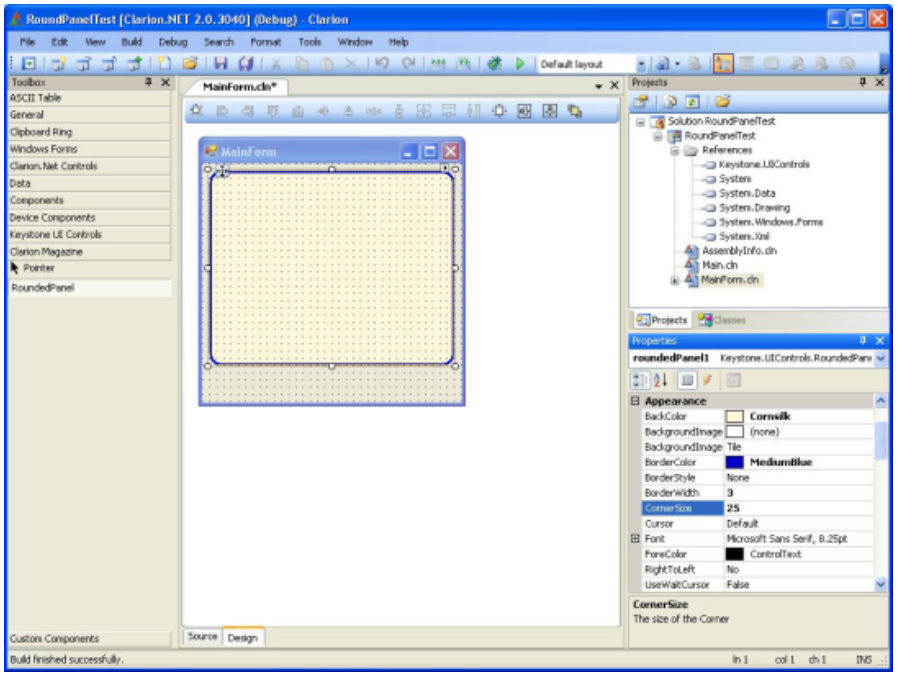

Figure 3. Using the rounded panel in the IDE (view full size image**)**

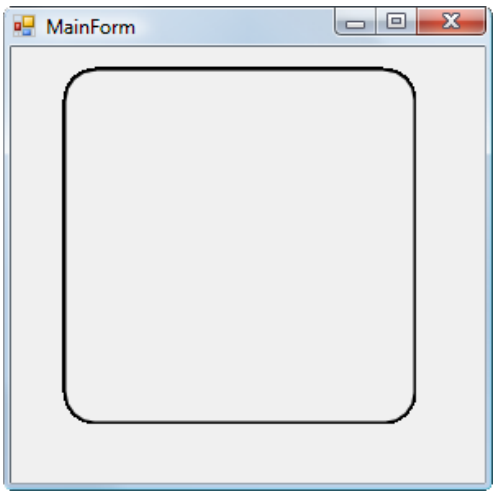

Figure 4. The rounded panel in action

Summary

That about wraps it up for this article. As you can see it is not too difficult to derive a control from a standard windows forms control and add new properties and behavior.

The downloadable source zip contains two Clarion# projects, one for the control library and the other for the test application. Unzip using the existing paths to keep the projects separate.

Resources

- Followup: Enhancing The RoundedPanel Control

Download the source

Randy Rogers is a data processing professional with over 35 years of experience in a wide variety of industries including accounting, municipal government, insurance, printing, and pharmacoeconomics. He has a degree in Mathematics from Florida State University and is the president of Keystone Computer Resources. Randy is the author of ClassViewer, a utility for browsing the Clarion class hierarchies. He is also the creator of NetTools, Queue Edit-in-Place, and Screen Capture Tools for Clarion application developers.

Reader Comments

Posted on Wednesday, April 30, 2008 by Stephen Ryan

thanks randy, great demo on the use of graphics and the onpaint events of controls. Of course if you started out in clarion for dos or even clarion for windows onpaint is something you dont think about let alone actually update the window when it repaints control by control.

just shows how much stuff clarion has always taken care of and now if you want to use clarion and dot net its all pretty simple for us all to learn!

Add a comment

Clarion Magazine

RTF Mail Merge

by John L Griffiths

Published 2008-04-30

Recently I built a document management system that operates across multiple physical sites connected via the internet. The system involved data collection at remote sites and transmission of that data to a central location where some documents were generated using Capesoft's Office Inside and MS Word . The documents were stored in an SQL database at the central location and could be requested by the remote sites for printing and signing.

As with any project that has been in place for a while, the users began to want more features.

I now needed to allow the users to generate some of the documents at the remote sites, independent of the central server. I could not be sure that these users would all have MS Word so the tool I would normally use (Office Inside) was not an option.

I opted to implement the feature using SoftVelocity's built-in RTF control.

The process plan

My process plan was as follows:

- Collect the Data
- Validate the Data
- Make a copy of the Source Document
- Run a "Replace Routine"
- Save the Document
- Print The Document

The source document

I soon learned that the RTF control had a few limitations. I was able to work around these by carefully changing the source documents I was supplied by the client.

The limitations I found were:

- The document should have no header/footer sections
- There should be no MS Word sections with newspaper style columns
- Tables were okay but I had to be sure than none would stretch across page breaks

Aside from tables it is okay to include text boxes and images within the document.

I create the source document using MS Word and save it as an RTF file.

Within that source RTF document I place the tokens that I will later use as targets for my replacement routine. My tokens

start with qq$ and are kept reasonably short to speed up the RTF control's standard FindAndReplace processing.

The RTF text control

You will want to use the RTF Text Control, not the older RTFControl.

I place this control on a window with a timer and use the timer to increment a staging variable I name Stager:

```
Stager += 1    !increment the Stager
0{prop:timer} = 0
CASE Stager
OF 1
  do CopySrcToDest
OF 2
  do LoadDoc
OF 3
  do Replacers
OF 4
  do PrintDoc
OF 5
  do AllDone
OF 6 to 999
  post(event:CloseWindow)
end
```

By using Stager this way I can step through a series of tasks without going into a tight loop.

My original plan was to have the window with the RTF control located at co-ordinates such as -3000,-3000 so it would not show on screen. I later found it best to let the user see the window as my automated process was doing its thing.

I set up a TYPEd group to contain the information needed to generate the document:

```
PGT    GROUP,TYPE   ! Group Type for passing to ProcessRTF
SrcDoc   cstring(201)
DestDoc  cstring(201)
Salute   string(12)
lname    string(30)
fnames   string(40)
addr1    string(30)
addr2    string(30)
City     string(30)
State    string(5)
Zip      string(10)

Balance  decimal(11,2)
PastDue  decimal(11,2)
```

```
    iRate    decimal(7,4)

    DueDate  long
    PayBy    long
    MinPay   decimal(11,2)

        end
```

I have another group which contains information about the printing process itself:

```
    ParamGRP          GROUP,PRE(PG) !parameters Group
    AlsoSave          BYTE
    PrintCopies       BYTE
    NoShow            BYTE
    AllowEdit         BYTE
              END
```

Now I'm ready to set up the data I'll need to do the mail merge:

```
    pg:AlsoSave = 1          ! 1 will retain saved output to MyDocuments
    PG:PrintCopies = 1       ! 1 to 5 copies allowed
    PG:NoShow = 0            ! 0 will allow user to see operation
    PG:AllowEdit = 0         ! not used

    GLO:ProgPath = path()
    MyGrp.SrcDoc = GLO:ProgPath & '\PayBacks.RTF'
    ! Get the "My Documents" Folder
    ShGetSpecialFolderPath(0,   GLO:MyDocsPath  , CSIDL_PERSONAL, 0)
    GLO:MyDocsPath = LONGPATH(GLO:MyDocsPath)
    MyGrp.Salute = 'Mr'
    MyGrp.lName = 'Green'
    MyGrp.fnames = 'Billy John'
    MyGrp.addr1 = '2000 Green Rd South'
    MyGrp.City = 'Greensville'
    MyGrp.State = 'WA'
    MyGrp.Zip = '12345'
    MyGrp.Balance = -11123.55
    MyGrp.MinPay = 1000
    MyGrp.PayBy  =   TODAY() + 7
    MyGrp.iRate  =   33.3
```

Having gathered and validated the data into a group I pass this group and the ParamGrp described above to the procedure, which is basically a window with a timer and an RTFTextControl. The document generation is then driven by the timer and several routines.

To find the user's My Documents folder I use the Windows ShGetSpecialFolderPath function. To find out more about this function check out Randy Rogers' article on Vista and INI files. See also the MSDN entry for ShGetSpecialFolderpath.

The LoadDoc routine

In the LoadDoc routine I load the file into the RTF control and set the sizes of the paper and the margins. I do this using a reference to the window control.

The RTF control seems to be quite dumb and cannot read its page size information from the source document so I must tell it how big the document should be:

```
RTFControl1.Load(MyGrp.DestDoc)
! Set the measurements UNITs I will be using
rtfUnit = UNIT:Twips
! Make a Reference to the RTF Control Object
pgsetup &=RTFControl1.Props.PageSettings()
! Set the Width and height Here I convert Centimeters to TWIPS
w = 21 * 567          !! 21 Cm
h = 29.7 * 567        !! 29.7 Cm
! TELL the control what its Size is
pgsetup.SetSize( w , h , UNIT:Twips )
! Set the Margins
lm = 720
rm = 720
tm = 720
bm = 720
pgsetup.SetMargins( lm , tm , rm , bm , rtfUnit)
```

I am now ready to replace tokens with the data I passed in. To do this I make several calls to the RTF Control's FindAndReplace method. Here is the call to replace the qq$Salute token with the contents of the MyGrp.Salute field:

```
RTFControl1.FindAndReplace( 'qq$salute', clip(mygrp.salute ) )
```

Once I have done all the replacements I can save the document:

```
! now SAVE the Changed document to the file
RTFControl1.TakeAction( RTFToolbar:CtlButtonSave)
```

And here's the code to print it (using the margins I forcibly set earlier):

```
RTFControl1.TakeAction( RTFToolbar:CtlButtonPrint )
```

Now I just close the window and I'm done.

Figure 1 shows the demo window with sample data. Figure 2 shows the generated RTF document before printing.

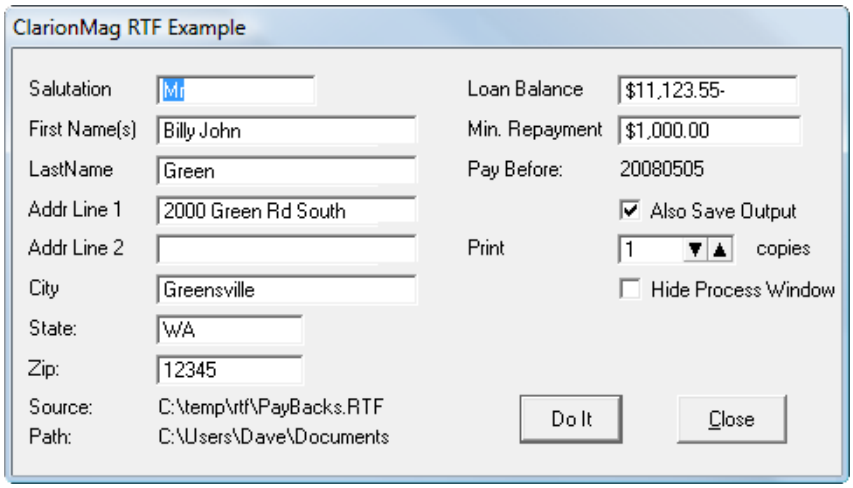

Figure 1. The demo data

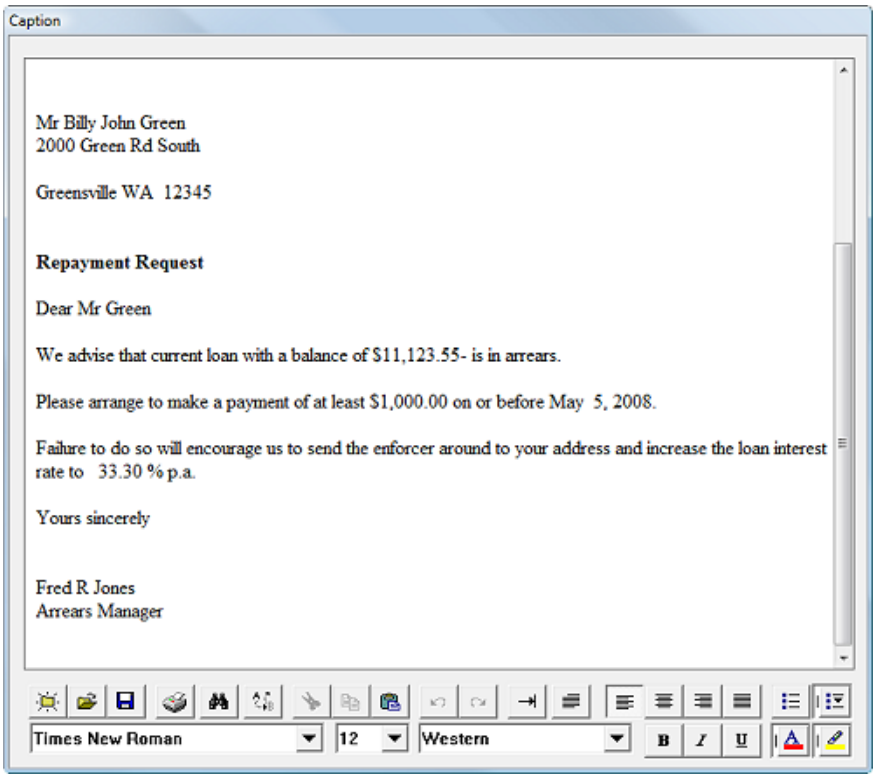

Figure 2. The generated RTF document (view full size image**)**

I use the window with a timer and the Stager variable in many situations, especially when processing documents with third party tools. The timer-based processing seems to give the operating system time to complete tasks that need to be done in an orderly and controlled sequence. At the beginning of each timer event where I increment the Stager variable I set the window's timer back to zero. Then I call the appropriate routine, and before exiting the routine I turn the timer back on.

Some timer settings can be critical. Notice in the example I set the timer to 100 after the initial copy of the source document is made. But after other routine calls I am setting it to a value of 60. I have found that on a busy server I sometimes need to allow a pause of about a second so the copied file is available for the next action. Perhaps this is due to the operating system drive caching settings. Theoretically once the copy statement completes the file should be available, but my experience has shown otherwise.

Where I am setting the timer back to 60 in the example you can safely use a smaller value. I usually use a value of 20 or 30.

This approach to automated RTF processing has proven very useful in several applications I have written. I hope it helps you.

Download the source

John Griffiths, an Australian, has been developing with Clarion since the DOS days. John has two summers each year, spending six months in Australia for the Southern summer and six months in Texas for the Northern summer. He works as a contractor for a Texas company and has developed several business/financial programs which he sells worldwide. John has a B.Bus degree with a major in Informations Systems.

Reader Comments

Posted on Thursday, May 01, 2008 by S Jayashankar

Hi John,

A more flexible approach to RTF Mail Merging would be to embed tokens with actual columns names like <CUS:FirstName>, <CUS:Address1>, etc and evaluate them before replacing. This gives you the added advantage of using formulas lile <SUB(CUS:FirstName,1,1)>. Processing is as simple as idetifying all the tokens using INSTRING(), EVALUATE(), FindandReplace and Print.

Regards

.....

Posted on Thursday, May 01, 2008 by Riebens

The RTF component of the Locus templates makes use of the token replacement strategy to do mail merging, as suggested.

.....

Posted on Thursday, May 01, 2008 by Riebens

John

Very nicely laid out and easy to follow article...

Thanks for the easier learning curve.....

Add a comment

Clarion Magazine

Passing Filter Expressions To Reports

by Paul Blais

Published 2008-04-17

Last time I introduced a modified version of the Invoice application which uses a few new controls and a dynamic filter to add query-by-example (QBE) to a browse. I also described a fictious flower shop bookkeeper who uses that application and is pleased with the changes.

Now the bookkeeper needs to print the resulting list. A quick trip back to the procedure list shows an existing report, PrintPRO:KeyproductSKU, which has the right format. All it needs is the right filter.

I mentioned earlier that SetFilter is a ViewManager method, and ViewManager is the base class for both BrowseClass and ProcessClass. When I create a report on a file I'm using ProcessClass, so if the report contains the same fields as my browse I should be able to simply pass my filter to the report.

Here is where the act of not using BIND is going to be a huge asset. If I had used BIND I would need to declare new local variables in the report, add the variables to the BIND list, pass all the local variables to the report, and finally assign them all to the report local variables. If I made them global variables, then I would only have to BIND them again and I could cut and paste the BIND code, but global variables raise other problems.

I don't need global variables and I don't need to cut and paste BIND code. Both of these concepts always tell me I am going to the dark side of writing too much useless code.

Passing the filter

Since I don't want to use a global variable I need to modify the report procedure so I can pass the filter string. Set both the Prototype and the Parameters to (String FilterString) as shown in Figure 1.

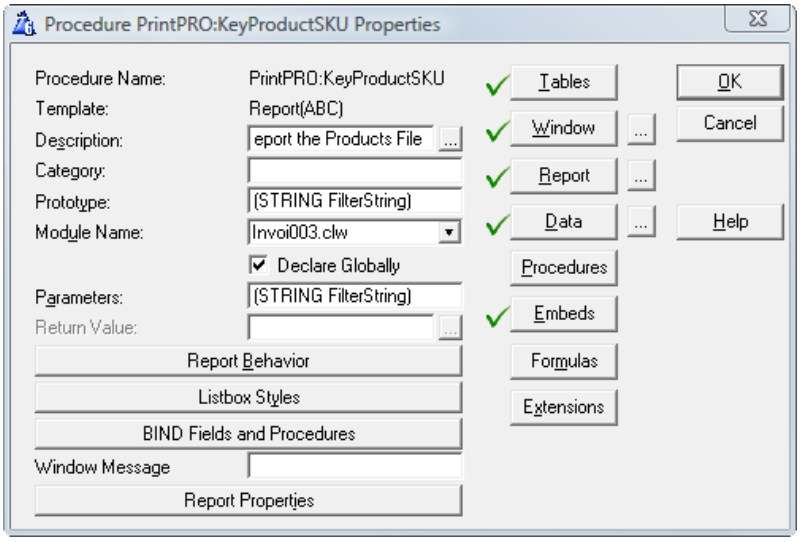

Figure 1. Setting the prototype and parameter list

I now can pass the filter by value as a string parameter. I really should be almost done, but the Clarion IDE is funny about passed parameters. Since parameter lists are just strings the IDE knows nothing about their contents, and you can't

pick the individual parameters in any template prompt. Fortunately, I can assign the FilterString to the report's filter without having to pick any variables. I go to the Report procedure definition and click on the Report Properties button (Figure 2).

In the Record filter field type

```
!FilterString
```

The IDE will remove the ! and not attempt to validate the text that follows.

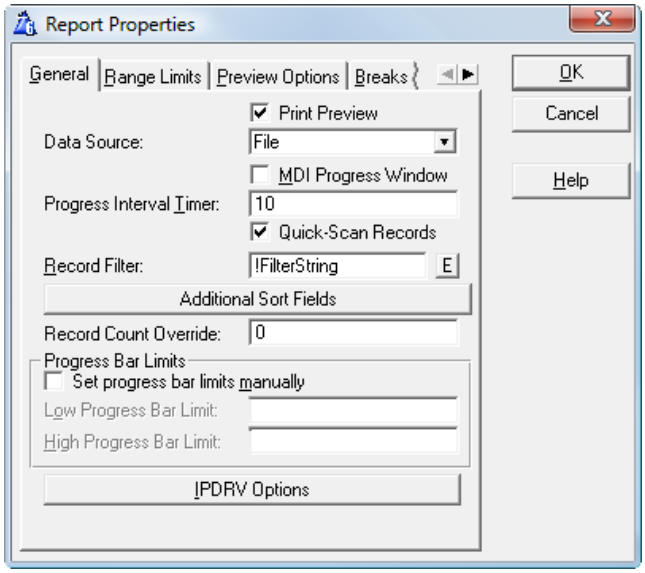

Figure 2. Setting the record filter

If I try to compile at this point I'll get a compile error because the report is also called from the main menu, but without a string parameter. The easiest way to fix that is to go to the main menu's window, edit the menu item, and on the Actions tab enter an empty string of '' in the parameters list.

I still need a print button. I quickly populate a new button on the browse screen and I add the following code to the Print button accepted event:

```
PrintPRO:KeyProductSKU(BRW1::View:Browse{Prop:Filter})
```

I reconsider and add an IF statement. The bookkeeper hates wasting paper for blank reports.

```
IF RECORDS(Queue:Browse:1)
  PrintPRO:KeyProductSKU(BRW1::View:Browse{Prop:Filter})
END
```

I can now run the full report from the main menu, or I can run a filtered report from the Products browse.

Displaying the filter

If these reports are just subsets of the whole data, that should really be indicated on the report. Figure 3 shows what I did.

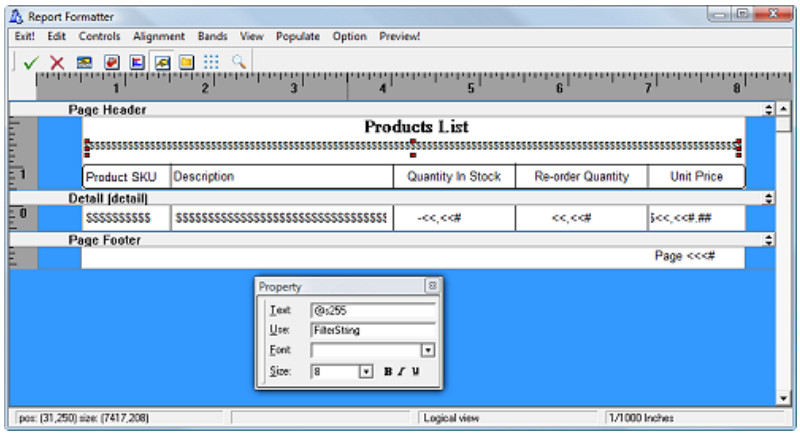

Figure 3. The modified report layout (view full size image**)**

I moved the title up and added a string control under the title in small print, center justified. I also went to the page view and dragged the upper boundary of the body so it won't overlap the new larger header.

If I simply set FilterString as the variable for the string control and I attempt to run the report from the main menu or from the Products browse with no filter set, the application will GPF. The answer is to create a procedure variable, which I've called ReportFilterString. It's a 500 character string, which should be large enough. Then in ThisWindow.Init I assign FilterString to ReportFilterString.

For consistency, I should change the report's filter from FilterString to ReportFilterString as well.

At last I have victory. I can see that my change to support a special report is quite complete. Or perhaps not.

One moment please

A little bit of user testing reveals a problem with the old QBE template and my new QBE addition that was not well received by the flower shop bookkeeper.

I mentioned above that you can still run the QBE template extension (leaving aside for now the question of whether that's a good idea). Perhaps not immediately noticed is how the clear button at the bottom of the browse does not clear the QBE set in the template, and if I go back to the QBE extension and clear the QBE it does not clear my QBE variables. "Why?" was the question from the bookkeeper. "It's a feature" I say. The reply is "No, it's not". Users always know what they hate when they see it.

The problem is the way the ViewManager.SetFilter method works. You'll recall that the second parameter is the filter ID. The QBE extension does just what I did above, but it uses the ID '9 - QBE'. When you press Clear, the QBE template only clears that specific filter. A direct assignment of the View Filter using Prop:Filter uses the ID '5 - Standard'. ABC does this to allow multiple sources of filtering to coexist without stepping on each other and to give you a way of inserting your filters before or after the built-in filters. Whatever the reason, the flower shop bookkeeper is not impressed.

The QBE extension template generates a bunch of code into the browse procedure, including a couple of classes that mange the QBE behavior. The first task is to make my Clear button clear both the local group and the filter and the '9 - QBE' filter ID. The next task is to clear the group variables when I enter the QBE by pressing the Query button.

I still need to plug one more nasty hole. If the QBE is active and I decide to add values to the browse fields I need to clear the QBE filter even if my user forgets to hit the clear button. That makes three places I need to change my code.

I'm going to make my code easier to read by adding another local field called ForceReset, a BYTE data type. I'll add it to the set of reset fields above. I want a way to force my filtering code to clean up this mess and I want all that code in one place so I don't run around hunting to find it nine months down the road.

I make the following changes to the Clear button's Event:Accepted embed point:

```
CLEAR(QueryGrp)
```

```
ForceReset = TRUE  ! add this new line the browse reset will deal with it
```

The Query button accepted event needs the following embed code to clear my own filters when the user wants a standard QBE:

```
CLEAR(QueryGrp) ! this is new code turn off my filters
```

Changes to the ApplyFilter section

I'm going to wrap all the code that actually makes filters inside an IF block. The block will only execute when there are values in any of the local variable reset fields but the Init of my filters will always be initialized to blank. The IF block further isolates the new code from the QBE extension. I will also include the a test for when ForceReset is true. I only need to control the QBE extension template when my filters are being used. This logic will now do all I need and only when I need it.

I can now refactor the code as follows:

```
IF QGrp:HighPrice OR |
  QGrp:LowPrice  OR |
  QGrp:HighQuantityInStock OR |
  QGrp:LowQuantityInStock  OR |
  QGrp:HighReorderQuantity OR |
  QGrp:LowReorderQuantity  OR |
  QGrp:HighCost            OR |
  QGrp:LowCost             OR |
  QGrp:MissingPicture      OR |
  LEN(CLIP(QGrp:Description)) OR |
  ForceFilter = TRUE

  Self.SetFilter('','9 - QBE')  ! clear QBE filter - it conflicts with my own
  ForceFilter = FALSE           ! reset the force set by the Clear button

  ! Price filters                ! all the remaining code is unchanged
  IF QGrp:HighPrice
    Self.SetFilter('PRO:Price <= ' & QGrp:HighPrice,e_FilHighPrice)
  END
  IF QGrp:LowPrice
    Self.SetFilter('PRO:Price >= ' & QGrp:LowPrice,e_FilLowPrice)
  END
  ! qauntity in stock filters
  IF QGrp:HighQuantityInStock
    Self.SetFilter('PRO:QuantityInStock <= ' & |
      QGrp:HighQuantityInStock,e_FilHighQuantity)
  END
  IF QGrp:LowQuantityInStock
    Self.SetFilter('PRO:QuantityInStock >= ' & |
      QGrp:LowQuantityInStock,e_FilLowQuantity)
  END
  ! Reorder filters
  IF QGrp:HighReorderQuantity
    Self.SetFilter('PRO:ReorderQuantity <= ' & |
```

```
      QGrp:HighReorderQuantity,e_FilHighReorder)
    END
    IF QGrp:LowReorderQuantity
      Self.SetFilter('PRO:ReorderQuantity >= ' & |
       QGrp:LowReorderQuantity,e_FilLowReorder)
    END
    ! cost filters
    IF QGrp:HighCost
      Self.SetFilter('PRO:Cost <= ' & QGrp:HighCost,e_FilHighCost)
    END
    IF QGrp:LowCost
      Self.SetFilter('PRO:Cost >= ' & QGrp:LowCost,e_FilLowCost)
    END
    IF LEN(CLIP(QGrp:Description))
     Self.SetFilter('INSTRING(' & '<39>' & >
       UPPER(CLIP(QGrp:Description)) & '<39>' & |
        ', UPPER(PRO:Description) ) <> 0',e_FilDesc)
    END
    IF QGrp:MissingPicture
     Self.SetFilter('LEN(CLIP(PRO:PictureFile)) = 0',e_FilPicture)
    END

  END ! don't forget this end for the IF
```

The issue of resetting the QBE extension filter to blank is important. Using this approach I never need to know if a QBE is active so long as I shut it off when I need to make my own active, or when the user presses my new Clear button indicating they want no filters at all. I don't know a method I can use to detect the activity of a QBE filter expression, but I do know my own filter activity from my variables.

> **NOTE:** The QBE Extension template has an option to load the last query when the procedure is next launched from the main menu. This procedure does not use that feature but you can enable it and not be disappointed.

Save, compile, and run

This feature is complete. I have coded a very solid QBE and have used very few resources. I've broken no other features and can add more filter options or logic, although the ABC templates limit the total filter length to 5000 characters. I am not clear if this limit is artificial.

This code will migrate from Clarion 5 to Clarion 7 unchanged. With the exception of the INSTRING command, this code will also migrate from TPS to SQL. The ability to pass around filter strings allows far more custom options than copy and paste tactics. The techniques here also apply to the ABC Process template as used outside of reports.

Using these tools you will find even more places where expressions are great tools. Thinking about how easy code will be to change later on is a critical ingredient to modern programming.

Summary

These last requirements were not simple. To find the ID for the QBE filter string I had to dig through the BrowseClass LIBSRC code, as documentation does not exist to this level of detail. I use the Keystone Class Viewer and Carl Barnes' Clarion Source Search for that type of work.

In the real world I would not have two QBE methods in the same procedure, as under most circumstances users don't need

two ways to filter. I find the technique presented here to be a better way to deliver QBE features because the options are less limited. Also, the design is my own and requires no pop up window with hidden features. Making my code enhancements coexist with other code is an important part of using the embed points.

Understanding how View filters work is the key to using them in more places than you may have before. The survey of Clarion embed points recently done by Clarion Magazine revealed ValidateRecord to be the third most popular embed. If most of those embeds were placed in the ApplyFilter embed Clarion developers would not be asking for more data than they want and writing more code to throw away the unwanted data. ApplyFilter lets you ask for exactly what you want and you keep them all.

Study your ABC's to make more expressive browses (and reports) using dynamic expressions.

Download the source

Paul Blais has been a Clarion developer since CPD version 2003. He became a full time independent Clarion Developer in 1998. In 2000 he merged his life and business with his wife Barabra. The merger yielded 3 dogs, one horse, 3 vehicles, and Organizational Development Strategies, Inc.. When not writing code he and his wife can often be seen aboard Bright Eyes sailing the Chesapeake Bay.

Reader Comments

Add a comment

Clarion Magazine

Loosely Coupled .NET Applications And Inversion of Control

by Dave Harms

Published 2008-04-17

In the previous article in this series I laid out my example scenario of an invoice class which uses one of two tax calculation classes to determine tax amounts. As you'll recall these two tax classes both implement the ITaxManager interface; my argument for using interfaces rather than inheritance is that interfaces provide a higher degree of flexibility; any class at all can implement the specified interface.

I also gave some examples of how I would assemble an invoice object using one or the other of my ITaxManger implementations. But all of this was done in source code, and the point of this rather lengthy series of articles is to show that you can configure an application at runtime rather than have to rely on compile-time decisions.

Finally, it's time to work that runtime magic.

Dependency injection and inversion of control

The technique I'll show here is a particular form of *inversion of control*, or IoC. Inversion implies the opposite of something, and in programmer terms that something that needs reversing is the top-down kind of control that was most evident back in the DOS days. If you were a CPD programmer you may recall writing form code that assumed the user would proceed through form fields in the specified sequence. The program controlled the user's actions. When Clarion for Windows came along you had to get used to event-driven programming: the user did something, and your program had to respond. It wasn't possible for the code to know ahead of time the sequence of events; that was up to the user.

The kind of inversion of control I'm going to be talking about in this article doesn't depend on user interaction; rather, it depends on configuration information (usually in an XML file) that determines which objects the application will use to perform which tasks.

In order to do this kind of inversion of control you something called an *IoC container*.

Windsor IoC container

An IoC container isn't something you pee into at the Olympics; it's a library that uses reflection to discover and create classes based on the information you give it (again, usually via an XML file).

The IoC container I'm going to use is Windsor, and it's an open source product by the good folks at the Castle Project. Windsor is covered by the Apache license which means you can use it in closed-source commercial software if you wish.

Step one is to go to the Castle download page and install the entire Castle project, which includes Windsor.

Step two is to add the following references to the project. If you've installed to the default location you can find these DLLs in the Program Files\CastleProject\Bin\net-2.0 directory.

- Castle.Core
- Castle.DynamicProxy
- Castle.MicroKernel
- Castle.Windsor

Okay, now it's time to write some code. First, here are a few using statements that might come in handy:

```
using System
using System.Reflection
using Castle.Windsor
using Castle.Core.Resource;
using Castle.Windsor.Configuration.Interpreters
```

These aren't strictly necessary but without them I'll be doing a lot more typing.

Windsor without a config file

Ultimately I'll show you how to configure your application with an XML file, but first here are a few source examples that demonstrate what Windsor is and what it does.

Windsor is an IoC container, and like other containers you put stuff into it and you take stuff out of it. I'll begin with a simple example that works with the Invoice class. First I need two reference variables, one for the container and one for the invoice object I want to retrieve.

```
container       IWindsorContainer
inv             Invoice
```

Next, in the code I create a new instance of the container and I add a new Invoice component:

```
code
container = new WindsorContainer()
container.AddComponent('invoice', |
  typeof(ClarionMag.Reflection.Example.Invoice))
```

This form of AddComponent takes two parameters: the first is an arbitrary string which serves as a key for the component; the second is the Invoice class's type, via the typeof operator. You can use either of these to get back an instance of the object using the container's indexer:

```
inv = container[typeof(Invoice)] tryas Invoice
```

or

```
inv = container['invoice'] tryas Invoice
```

Be careful when getting objects via the string key as it *is* case sensitive.

Now you have an instance of the object and you can use it normally. When you're done it won't hurt to release the object back to the container, although the effect of this will vary, as I'll explain toward the end of the article.

```
! Do something with the object here
container.Release(inv)
```

Wiring up objects automatically

In the previous example I added only the Invoice class to the IoC container. That worked because I have a default

constructor that doesn't require any parameters. In fact, I'll only want to use this Invoice class in conjunction with an implementation of the ITaxManager interface, in this case TaxManager or TaxManagerGST. Watch what happens when I add both Invoice and TaxManager to the IoC container:

```
WindsorWiringDemo   procedure
container             IWindsorContainer
inv                   Invoice


  code
  ShowTaskName('WindsorWiringDemo')
  System.Console.WriteLine('Creating invoice with TaxManager')
  container = new WindsorContainer()
  container.AddComponent('invoice', |
    typeof(ClarionMag.Reflection.Example.Invoice))
  container.AddComponent('taxmanager',|
    typeof(ClarionMag.Reflection.Example.ITaxManager), |
    typeof(ClarionMag.Reflection.Example.TaxManager))
  inv = container[typeof(Invoice)] tryas Invoice
  ! Do something with the object here
  container.Release(inv)
  dispose(container)
```

The output from this code, thanks to the System.Console.Write statements I've put in my constructors, is:

```
Created TaxManager object
Created Invoice with ITaxManager object
```

All I did was add the two objects to the container and Windsor correctly wired them up. First it created the TaxManager object, which implements ITaxManager, and then it created the Invoice object using the constructor that takes an ITaxManager instance. Windsor automatically resolved the ITaxManager dependency and *injected* the TaxManager instance. Specifically this is called *constructor injection*, and is a specific kind of dependency injection.

Windsor with a config file

Although you can add components to Windsor in code, you're more likely to use an external configuration source in the form of an XML document. That way you can change the behavior of your application without having to recompile.

The standard location for .NET application-specific information is an XML file called app.config (and for web apps, web.config). When you compile your application, the app.config file is automatically copied to the same location as your executable and is renamed to *applicationname*.exe.config.

You can use app.config as a place to store your Windsor configuration information. Here's an example (line breaks added):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <configSections>
   <section
     name="castle"
     type="Castle.Windsor.Configuration.AppDomain.↵
```

```
        CastleSectionHandler, Castle.Windsor" />
  </configSections>
  <castle>
   <components>
    <component
     id="invoice"
     type="ClarionMag.Reflection.Example.Invoice, ↵
       ClarionMag.Reflection.Example"/>
    <component
     id="taxmanager"
     service="ClarionMag.Reflection.Example.ITaxManager, ↵
       ClarionMag.Reflection.Example"
     type="ClarionMag.Reflection.Example.TaxManager, ↵
       ClarionMag.Reflection.Example">
    </component>
   </components>
  </castle>
</configuration>
```

The <components> section lists each of the components you want Windsor to register. The invoice component has two name/value pairs: id is the string key identifying the component, and type is the fully qualified type (namespace + class name) and the assembly name, separated by a comma.

The taxmanager component adds a third element called service. The service in this case is the interface (ITaxManager) presented by the TaxManager object. When Windsor is asked for an instance of the Invoice class it sees the constructor requiring an ITaxManager instance and it goes looking in its list of services for a match.

Here's a version of the example code that loads up the data from app.config and creates an Invoice instance:

```
WindsorConfigDemo   procedure
container             IWindsorContainer
inv                 Invoice
   code
   container = new WindsorContainer(|
     new XmlInterpreter(new ConfigResource('castle')))
   inv = container[typeof(Invoice)] tryas Invoice
   ! Do something with the object here
   container.Release(inv)
```

That's short and sweet. The output will be the same as in the previous example:

```
Created TaxManager object
Created Invoice with ITaxManager object
```

If I change the taxmanager's type value as follows I'll get a TaxManagerGST instance:

```
 type="ClarionMag.Reflection.Example.TaxManagerGST, ↵
       ClarionMag.Reflection.Example">
```

Here's the output:

```
Created TaxManagerGST object
Created Invoice with ITaxManager object
```

Simply by changing the XML file I've altered the program's running code. But it's not simply because of Windsor that I can do this; it's also because I've designed my invoice and tax manager objects in such a way that I can plug in different implementations as needed.

Adding parameters

Combining objects often means more than just handling constructor dependencies; you may also need to set parameters on classes. So far I haven't run any actual tax calculation code, in part because I want to be able to specify the tax rates in the XML file.

Here again is the TaxManager implementation:

```
                    MEMBER('')

                    NAMESPACE(ClarionMag.Reflection.Example)
                    MAP
            END

TaxManager      class,implements(ITaxManager),public
Construct       procedure
_rate           real(0),private
Rate            property,Real,public
  inline
    getter; code; return self._Rate
    setter; code; self._Rate = value
  end
            end



TaxManager.Construct    procedure
  code
  System.Console.WriteLine('TaxManager.Construct called')

TaxManager.ITaxManager.CalculateTax procedure(real subtotal)
  code
  System.Console.WriteLine('TaxManager calculating taxes')
  return subtotal * self.rate
```

Note the Rate property. I want to set this value in app.config; here's the component section from app.config:

```
<component
 id="taxmanager"
```

```
  service="ClarionMag.Reflection.Example.ITaxManager, ↵
        ClarionMag.Reflection.Example"
  type="ClarionMag.Reflection.Example.TaxManager, ↵
        ClarionMag.Reflection.Example">
  <parameters>
    <Rate>0.12</Rate>
  </parameters>
</component>
```

The <parameters> element contains one <Rate> element which corresponds to the Rate property setter in TaxManager (although any method name is acceptable - it doesn't have to specifically be a SETTER).

If I instead want to implement TaxManagerGST I'll use a slightly different set of elements:

```
<component
  id="taxmanager"
  service="ClarionMag.Reflection.Example.ITaxManager, ↵
        ClarionMag.Reflection.Example"
  type="ClarionMag.Reflection.Example.TaxManagerGST, ↵
        ClarionMag.Reflection.Example">
  <parameters>
    <GstRate>0.05</GstRate>
    <PstRate>0.07</PstRate>
  </parameters>
</component>
```

TaxManagerGst has a completely different set of properties than TaxManager, and that's fine. All that really matters is that each class implements the appropriate interface, in this case ITaxManager.

Here's some code that creates an invoice, adds a couple of items, and calculates the tax:

```
WindsorConfigParameterDemo   procedure
container            IWindsorContainer
inv                        Invoice
   code
   container = new WindsorContainer(|
     new XmlInterpreter(new ConfigResource('castle')))
   inv = container[typeof(Invoice)] tryas Invoice
   inv.AddItem('Widget',20,3)
   inv.AddItem('Fiddlestick',10,4)
   System.Console.WriteLine('Invoice total: ' & inv.GetTotal())
   container.Release(inv)
   dispose(inv)
```

The output for this code is as follows:

```
Created TaxManagerGST object
Crated Invoice with ITaxManager object
TaxManagerGST calculating taxes: pst 0.07, gst 0.05
```

```
Invoice total: 112.35
```

The line item subtotal is $100; TaxManagerGST computes the PST as $7.00 and the GST as 5% of $107.00 or $5.35. If I were to use the TaxManager object with a value of 12% for the equivalent simple tax calculation the invoice total would be $112.00 rather than $112.35; the extra 35 cents is equal to 5% of $7.00.

Which object did I just get?

Let's say I want two invoice objects at the same time:

```
WindsorLifeCycleDemo      procedure
container              IWindsorContainer
invA                   Invoice
invB                   Invoice
  code
  container = new WindsorContainer(|
    new XmlInterpreter(new ConfigResource('castle')))
  invA = container[typeof(Invoice)] tryas Invoice
  invB = container[typeof(Invoice)] tryas Invoice
  container.Release(invA)
  dispose(invB)
```

When I look at the console output I see something odd: the TaxManager and Invoice constructors are each called just once:

```
Created TaxManager object
Created Invoice with ITaxManager object
```

I don't, in fact, have two different TaxManager objects, or two different Invoice objects: I have two references to an Invoice object, and that Invoice object has a reference to the only TaxManager object.

Any time you request an object from Windsor, that object is subject to a lifestyle specification. The available lifestyles are:

- singleton - only one instance is ever created
- thread - one instance will be returned for each thread
- transient - a new instance will be returned for each request
- pooled - a limited number of instances will be made available, and must be released after use (you also specify the initial and maximum pool size)
- custom - a custom implementation (you must also supply a class that implements the ILifestyleManager interface)

Presumably pooled instances are the only ones that actually need to be released back to the container, although I don't have any official docs on this.

The singleton lifestyle is the default, which is why I only got one object despite requesting two. I probably want a transient lifestyle, so I add the lifestyle attribute to my component in app.config:

```
<component
  id="invoice"
  type="ClarionMag.Reflection.Example.Invoice, ↵
        ClarionMag.Reflection.Example"
  lifestyle="transient"/>
```

I also add it to my TaxManager component:

```
<component
 id="taxmanager"
 service="ClarionMag.Reflection.Example.ITaxManager, ↵
       ClarionMag.Reflection.Example"
 type="ClarionMag.Reflection.Example.TaxManager, ↵
       ClarionMag.Reflection.Example"
 lifestyle="transient">
 <parameters>
  <rate>0.12</rate>
 </parameters>
</component>
```

If you're not accustomed to working with XML there's one little thing that might trip you up between those two component elements. The invoice component is a single element with no children, so it ends with />. The taxmanager component, however, has child elements, so the first part of the tag, right after "transient", ends with just a > character; the closing </component> tag comes at the end.

Now both components are marked as transient, and when I run my code again I get the following output:

```
Created TaxManager object
Created Invoice with ITaxManager object
Created TaxManager object
Created Invoice with ITaxManager object
```

It's clear that I want a different invoice object each time I ask for one, but what about the TaxManager? In fact, I may need only one, shared by all Invoice instances. But I also have to be careful of threading issues.

The trouble with singletons

In any given application you will only ever have one instance of a singleton class: that's the very definition of a singleton. And singletons are basically a kind of global data, and you should always be cautious about using global data. In a multi-threaded environment, which describes just about any web or desktop application these days, you have to be sure that singletons are thread-safe. The TaxManager.CalculateTax method should be fine, I think, because it works only with local copies of the received data, and with a property (the tax rate) which never changes. But in a real world situation you'd probably need CalculateTax to examine the queue of line items for individual tax applicability, and in that case (or any other where the code references shared data) you'd want to use a critical section or other synchronization approach to avoid mucking up the data.

An alternative to a singleton TaxManager is a per-thread TaxManager. That keeps the instance count down and as you only have one instance per thread you don't have to make the code thread safe. But don't get too hung up on reducing the instance count unless you have a bazillion class instances, or classes that really eat up memory, or a specification that limits the number of class instances. Creating and destroying instances is pretty inexpensive in the grand scheme of things.

Checking for exceptions

One final point: if Windsor can't find the class you ask for it'll return a Castle.MicroKernelComponentNotFound exception, so you'd be wise to wrap your object retreival code in a try/catch block and have a suitable exit strategy.

Summary

I began this series of five articles by hinting at the power of .NET reflection. In this last article in the series I've demonstrated how the Windsor IoC container, through the power of reflection, provides a powerful way of instantiating and wiring up objects so they're ready to use whenever and however you need them. I've also discussed the importance of interfaces in the design of loosely-coupled applications; with a little forethought you can create applications that are readily adaptable to many different situations.

There are many benefits to using an IoC container like Windsor, including:

- Classes are easily instantiated with the appropriate constructors and parameters
- Dependencies are automatically resolved
- Objects can have specific lifestyles - patterns such as pooling and singletons are easily achieved
- The way objects are wired up can be determined at runtime by configuration data (XML)

I've often said that one of the great advantages of Clarion# over Clarion is the incredible variety of programming tools which suddenly become available. Castle Windsor is a terrific example of a tool that was simply out of reach before Clarion#. I hope you'll find it a useful way to implement your loosely-coupled designs.

Resources

- Castle Project home page
- Download Windsor
- Windsor configuration information
- Wikipedia on Inversion of Control

Notes on the example app

The sample zip includes the compiled app and the necessary DLLs, so you don't actually need Clarion# to try it out. Just navigate to the bin\debug directory and run ClarionMag.Reflection.Example.exe from the command line (so you can see the console output). Use Program.cln as a reference to see what's happening, and experiment with different settings in the config file. Just remember that it's the ClarionMag.Reflection.Example.exe.config file that's used by the EXE, not the original app.config which is in the same directory as the source code.

Download the source

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of *Developing Clarion for Windows Applications*, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Reader Comments

Add a comment

Clarion Magazine

Loosely Coupled .NET Applications: The Baseline Example

by Dave Harms

Published 2008-04-16

In the previous article in this series I provided a bare-bones description of metadata and reflection. Now it's time to create an application that I can use to illustrate these principles more fully. Note that all of the code shown here is written in Clarion#.

The concept behind my invoice example app is simple. I have an invoice class which models an invoice and contains a list of line items as well as a reference to a class that will calculate the tax for those items. Now, a completely separate class might be overkill for something as simple as calculating taxes or might be barely sufficient depending on the tax regime. But the point is not that this is an example of how to approach invoicing so much as it is an example of how to approach loosely-coupled, component-based design.

I want to be able to substitute tax classes on the fly so I can accommodate any kind of tax calculation scheme without having to rewrite my invoicing functionality. Furthermore I want to be able to ship all the functionality in one package and determine the final behavior at runtime. But first I had better describe the classes involved. First I'll describe an inheritance-based approach, which I'll then discard in favor of an interface-based approach.

The inheritance scenario

Here's how I might declare the invoice and tax calculation classes for a simple flat rate tax, using an inheritance-based approach:

```
ItemQueue          queue,private
Description          string
Price              real
Quantity             long
                end

Invoice      class,public
ItemQ          ItemQueue,private
Construct       procedure
Construct       procedure(TaxManager taxman)
AddItem         procedure(String description,|
                  real price,long quantity)
GetTotal        procedure,real
taxman          TaxManager
            end

TaxManager     class,public
```

```
Rate                real(0),private
Construct           procedure
CalculateTax        procedure(real subtotal),real
SetRate             procedure(real rate)
                end
```

I'll get to the methods a little later when I've fleshed out the example, but as you can see from the Invoice class declaration I have a queue for my line items, a method to add new line items, and a method to get the invoice total. Most importantly the class also has a reference to a TaxManager object and a constructor that lets me assign that reference.

The code to create an invoice, add some items, and get an invoice total would look something like this:

```
inv             Invoice
taxman          TaxManager


  code
  taxman = new TaxManager()
  taxman.SetRate(0.07)
  inv = new Invoice(taxman)
  inv.AddItem('Widget',20,3)
  inv.AddItem('Fiddlestick',10,4)
  System.Console.WriteLine('Invoice total: ' & inv.GetTotal())
```

First I create a new instance of my TaxManager class and set the tax rate. Then I create a new Invoice instance, passing the taxman object to the constructor. I add two line items and I get back a total including the tax amount. This is a console application which is why I'm using System.Console.WriteLine instead of Message().

And yes, this is an overly simple example of an invoicing system. I did warn you.

Adding a new tax calculation with inheritance

Using the above design the only option I have for adding in a different tax calculation is deriving a class from TaxManager, and I've created one called TaxManagerGST in honor of the least popular tax in my own country. In 1991 the Canadian government introduced a Goods and Services Tax, abbreviated GST (and widely derided at the time as the "Gouge and Screw Tax"). There was some confusion at the time as to whether the GST would be applied in addition to provincial sales taxes or on top of provincial sales tax. Happily (does that world ever apply to taxes?) the GST is not in fact applied to provincial sales tax (PST).

But what if GST were charged on PST as well as the goods and services themselves? I might need a class something like this:

```
TaxManagerGST       class(TaxManager),public
GstRate             real(0),private
Construct           procedure
CalculateTax        procedure(real subtotal),real
SetGstRate          procedure(real GstRate)
                end
```

I could then use similar code to create and use the invoice:

```
inv             Invoice
taxman            TaxManagerGST

  code
  taxman = new TaxManagerGST()
  taxman.SetPstRate(0.07)
  taxman.SetGstRate(0.05)
  inv = new Invoice(taxman)
  inv.AddItem('Widget',20,3)
  inv.AddItem('Fiddlestick',10,4)
  System.Console.WriteLine('Invoice total: ' & inv.GetTotal())
```

Of course, I'd still need to implement my class methods and apply the appropriate taxes but all the data is there and the Invoice object has access to the tax class's CalculateTax method.

So far there's nothing here that you can't readily duplicate in C6. And the same goes for the next bit. But stick with me, because this is going somewhere important.

Redesigning with interfaces

Although the inheritance-based approach I've shown here works, I'm going to redesign it to use interfaces. As you'll recall from Using Interfaces And Composition To Create Flexible Applications an interface is a class-like structure without any data or method code, and which defines the methods that must be part of any class that implements that interface. Although inheritance will work in this situation, I don't necessarily want to restrict the implementation of the tax calculation class to just child classes of TaxManager. Changing to an interface-based approach allows me to use any class at all for the tax calculation, and if I find I really need the base class I can always create an instance of it within my interface implementing class.

Here's an interface that defines just one method:

```
  MEMBER('')

  NAMESPACE(ClarionMag.Reflection.Example)
  MAP
  END

ITaxManager     interface,public
CalculateTax      procedure(real subtotal),real
            end
```

There's nothing else to this interface, meaning there is no source code anywhere for this CalculateTax method. It's simply a pattern, if you like, that any class implementing ITaxManager must follow.

Here's the TaxManager class implementation:

```
            MEMBER('')

            NAMESPACE(ClarionMag.Reflection.Example)
            MAP
            END

TaxManager      class,implements(ITaxManager),public
Construct       procedure
_rate           real(0),private
Rate            property,Real,public
  inline
    getter; code; return self._Rate
    setter; code; self._Rate = value
  end
            end



TaxManager.Construct    procedure
  code
  System.Console.WriteLine('Created TaxManager object')

TaxManager.ITaxManager.CalculateTax procedure(real subtotal)
  code
  System.Console.WriteLine('TaxManager calculating taxes')
  return subtotal * self.rate
```

You can see that TaxManager now implements(ITaxManager). Invisibly, the ITaxManager.CalculateTax method is included in the TaxManager class declaration and that means that the method implementation must follow the same syntax (TaxManager.ITaxManager.CalculateTax). The code for CalculateTax is simple: it returns the received subtotal plus the subtotal times the tax rate. This is demonstration code, remember? In a real world situation you could easily have some line items that were tax exempt or subject to different rates, or whatever.

I've also included a default constructor with a console message function so I can see some debugging output when the constructor is called.

The only other item to note is the getter/setter syntax, which is new in Clarion#. Getters and setters simply provide a way of using functions to assign values to and get values from variables using property-like syntax. If I have an instance of this class called taxman, the following code gets and sets the content of the private _rate variable:

```
taxman.Rate = .06
newRate = taxman.Rate
```

The advantage of using getters/setters over simple class properties is you can exercise some control over what data gets put in and read out, and you can also declare properties as GETONLY or SETONLY. I'll have more on getters/setters another time; they're fairly well documented in the latest Clarion.NET help file. And they're not a requirement here; I could also

have implemented the getting/setting functionality using normal methods.

Here's the code for the TaxMangerGST class:

```
                MEMBER('')

                NAMESPACE(ClarionMag.Reflection.Example)
                MAP
                END

TaxManagerGST       class,implements(ITaxManager),public
construct           procedure
_PstRate             real(0),private
PstRate              property,Real,public
  inline
    getter; code; return self._PstRate
    setter; code; self._PstRate = value
  end
_GstRate             real(0),private
GstRate              property,Real,public
  inline
    getter; code; return self._GstRate
    setter; code; self._GstRate = value
  end
            end

TaxManagerGST.Construct       procedure
  code
  System.Console.WriteLine('Created TaxManagerGST object')


TaxManagerGST.ITaxManager.CalculateTax      procedure(real subtotal)
PstTotal    real
  code
  System.Console.WriteLine('TaxManagerGST calculating taxes: pst ' |
    & self.PstRate & ', gst ' & self.GstRate)
  ! Thankfully the feds don't really compound the GST, although
  ! there was talk of it when the tax was first proposed.
  PstTotal = subtotal * self.PstRate
  return ((subtotal + PstTotal) * self.GstRate) + PstTotal
```

TaxManagerGST as two properties, PstRate and GstRate, and a slightly more complicated tax calculation since it's compounding the taxes.

Now, here's the Invoice class:

```
                MEMBER('')

                NAMESPACE(ClarionMag.Reflection.Example)
                MAP
                END

ItemQueue           queue,private
Description           string
Price               real
Quantity              long
              end

Invoice       class,public
ItemQ           ItemQueue,private
Construct       procedure
Construct       procedure(ITaxManager taxman)
AddItem         procedure(String description,real price,long quantity)
GetTotal        procedure,real
taxman          ITaxManager
            end

Invoice.AddItem   procedure(String description,real price,long quantity)
  code
  ItemQ.Description = Description
  ItemQ.Price = Price
  ItemQ.Quantity = Quantity
  add(ItemQ)



Invoice.GetTotal  procedure
x               long
total           real


  code
  total = 0
  loop x = 1 to records(self.ItemQ)
    get(self.ItemQ,x)
    total += (self.ItemQ.Price * self.ItemQ.Quantity)
  end
```

```
    return total + self.taxman.CalculateTax(total)


Invoice.Construct   procedure(ITaxManager taxman)
    code
    self.taxman = taxman
    System.Console.WriteLine('Created Invoice with ITaxManager object')

Invoice.Construct   procedure
    code
    System.Console.WriteLine('Created Invoice without ITaxManager object')
    self.ItemQ = new ItemQueue
```

Take note of the taxman reference and the parameterized constructor. Both now use ITaxManager instead of TaxManager. That means that whatever object is passed to the constructor and assigned to taxman has just one requirement: it must implement ITaxManager. And taxman, as an instance of an object implementing ITaxManager, will only "know" about the methods declared in that interface.

The code to implement and use the combination of Invoice and TaxManager is identical to the earlier example except that I'm using a property and a setter instead of the SetRate function. Had I stuck with SetRate the code would be identical:

```
SimpleDemo        procedure
inv               Invoice
taxman              TaxManager

    code
    taxman = new TaxManager()
    taxman.Rate = .12
    inv = new Invoice(taxman)
    inv.AddItem('Widget',20,3)
    inv.AddItem('Fiddlestick',10,4)
    System.Console.WriteLine('Invoice total: ' & inv.GetTotal())
    System.Console.ReadKey()
```

And again, the GST version is almost identical to the previous example, except for the property setters.

But what good is it?

You may be able to see the value in approaching a problem like invoicing using discreet objects, some of which are implemented with interfaces. But that's just the beginning. All I've done so far is lay more groundwork; the really good stuff comes in the next and final article in this series in which I'll show how to assemble objects, at runtime, into a configurable application by means of a simple XML configuration file.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author

with Ross Santos of *Developing Clarion for Windows Applications*, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Reader Comments

Add a comment

Clarion Magazine

Using Dynamic Filter Expressions for QBE

by Paul Blais

Published 2008-04-11

In a previous article I explored Clarion's EVALUATE statement and other runtime expressions. In this article I want to continue the discussion of expressions with a look at filter expressions employed in the Clarion VIEW structure. These expressions are a close cousin to EVALUATE and a great way to enhance your users' ability to query the database.

I will cover a technique that will put all the code in one tidy embed point, and I won't require any local variable to be BINDed. Using this technique I can build query by example (QBE) browses quickly and with a high level of user interaction. No additional templates are required, and I can write complex filtering logic that is easy to maintain.

Some ABCs

There are three main embed points you can use to filter ABC browses: ThisWindow.Init, BrowseClass.ValidateRecord, and BrowseClass.ApplyFilter.

ThisWindow.Init does provide a way to set up filtering (be sure to do so after the browse is initialized), but the code only executes once, at startup. You can't use ThisWindow.Init to apply filter options set by the user.

The BrowseClass.ValidateRecord virtual method embed point is a common solution to runtime filtering. You test for a filter condition in your code and then set the return value to either Record:Filtered or Record:OK. It works well enough, but what happens is all the records get retrieved and you throw out what you don't want. If you have a large data set filtering can be very slow.

The best solution is the BrowseClass.ApplyFilter embed point. This method executes whenever the browse refreshes; the trick is to specify whatever filters are needed before that refresh happens.

Figure 1 shows the embeditor before any code has been added to ApplyFilter. As you can see, there's a call to Parent.ApplyFilter. All you have to do is set the filters you want before this call and your filters will be applied.

```
BRW1.ApplyFilter PROCEDURE

! Start of "Browser Method Data Section"
! [Priority 5000]

! End of "Browser Method Data Section"
  CODE
  ! Start of "Browser Method Code Section"
  ! [Priority 2500]

  ! Parent Call
  PARENT.ApplyFilter
  ! [Priority 7500]

  ! End of "Browser Method Code Section"
```

Figure 1. The ApplyFilter embed points

Using SetFilter

You specify a browse's filters with the SetFilter method. BrowseClass is actually derived from the ViewManager class, which is where SetFilter is declared; ProcessClass is also derived from ViewManager, which has important benefits for reporting as I'll explain a little later.

The sole job of the ApplyFilter method is to stuff a logical expression into the ViewManager.

Figure 2 shows the normal browse template action prompt where you can manually type a Filter Expression. If my logic is simple enough to fit into one expression it goes here, but it's not always possible to reduce the logic to just one expression. This template also creates a static filter built by the programmer which is not evaluated until run time. If you're using EVALUATE this can be a time bomb waiting to crash your application with a BIND error. You almost never get a compiler error with this code.

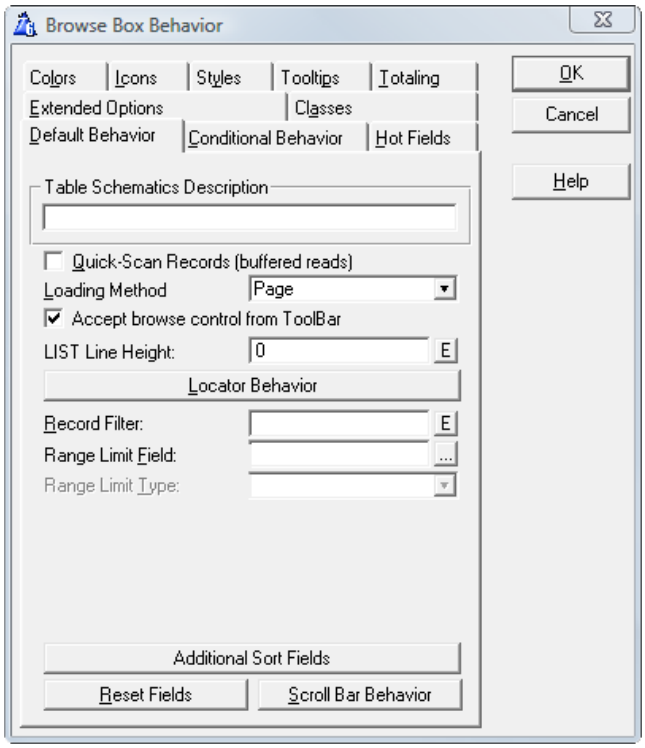

Figure 2. Setting a browse filter on the Default Behavior tab

The example

All versions of Clarion optionally install a set of example applications to demonstrate all the features of the ABC template chain. The example I'll use is the Invoice app which you can find in both the C6 and C5.5 examples subdirectory.

Figure 3 shows the procedure tree of the Invoice.app

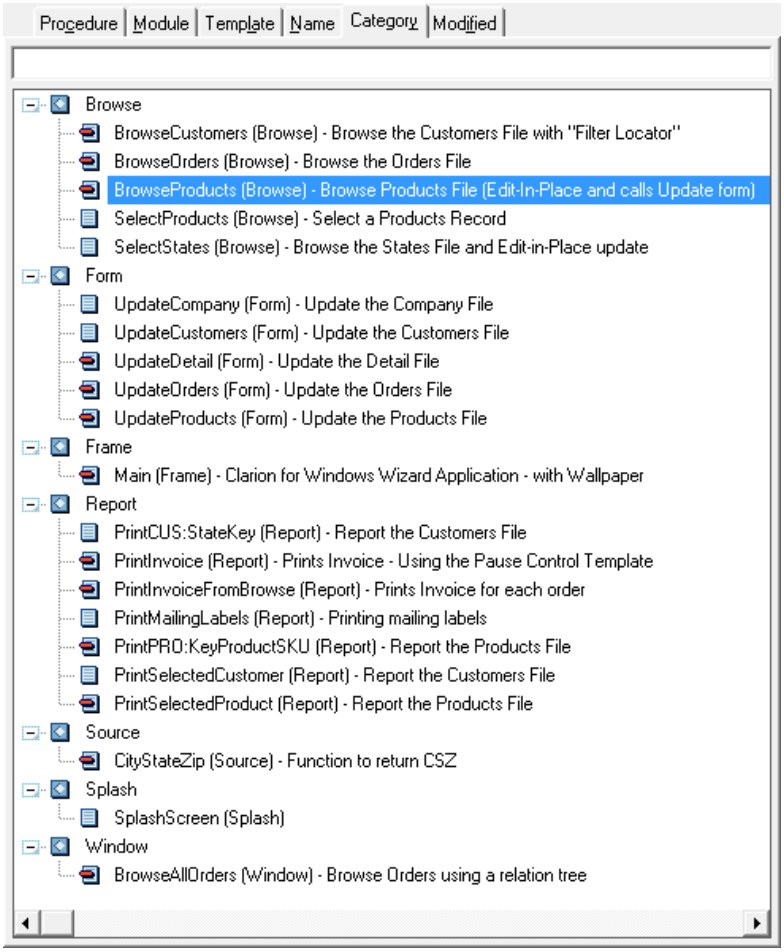

Figure 3. The procedure tree

From the category view open the BrowseProducts procedure. Next, open the Window in your IDE Screen formatter as shown in Figure 3.

The BrowseProducts procedure uses the query by example extension template (see the Query button). I am going to duplicate this feature using one embed point and some local variables. I won't break the edit in place or the QBE add-on with my work. You can actually run them both together.

Before you make any changes compile and run the Invoice.exe. Note how the BrowseProducts screen functions in the current version. I chose this example because it already is a richly featured browse. Hot fields and images only enhance the things users can do with a browse; more importantly, features like these eliminate a lot of wasted paper printouts. Screens that show the information are far more efficient than printing on paper. I will place the new QBE fields on the same screen so the user can access them quickly.

Getting started

To add runtime filtering to this browse I will need some local variables to hold the values entered by the user.

```
QueryGrp            GROUP,PRE(QGrp)!
Description         STRING(35)          !Product's Description
HighPrice           DECIMAL(7,2)        !Product's Price
LowPrice            DECIMAL(7,2)        !Product's Price
HighQuantityInStock DECIMAL(7,2)        !Quantity of product in stock
LowQuantityInStock  DECIMAL(7,2)        !Quantity of product in stock
```

```
HighReorderQuantity    DECIMAL(7,2)        !Product's quantity for re-order
LowReorderQuantity     DECIMAL(7,2)        !Product's quantity for re-order
HighCost               DECIMAL(7,2)        !Product's cost
LowCost                DECIMAL(7,2)        !Product's cost
MissingPicture         BYTE                ! is there a picture file
                   END! End group
```

I begin with a group structure called QueryGrp with a QGrp: prefix. I add two fields for each numeric Products field, with both a low and a high range variable. I'll use INSTRING to match the product description with the entered description. And the last field is a BYTE field check box that will filter out those records that are missing a picture.

As I add each field in the IDE data form I use the Derive button to pick the corresponding product field. This is a great lazy way to work - assuming I did all my DCT definitions properly - since the fields will populate as perfect clones including all prompts, control types, picture, etc. I do however change all the number entry fields to spin button fields with a step value of one.

Making the screen

Figure 4 adds some space below the browse for the new fields.

I have added a group box and included all the fields in the new local group. They require no embedded code.

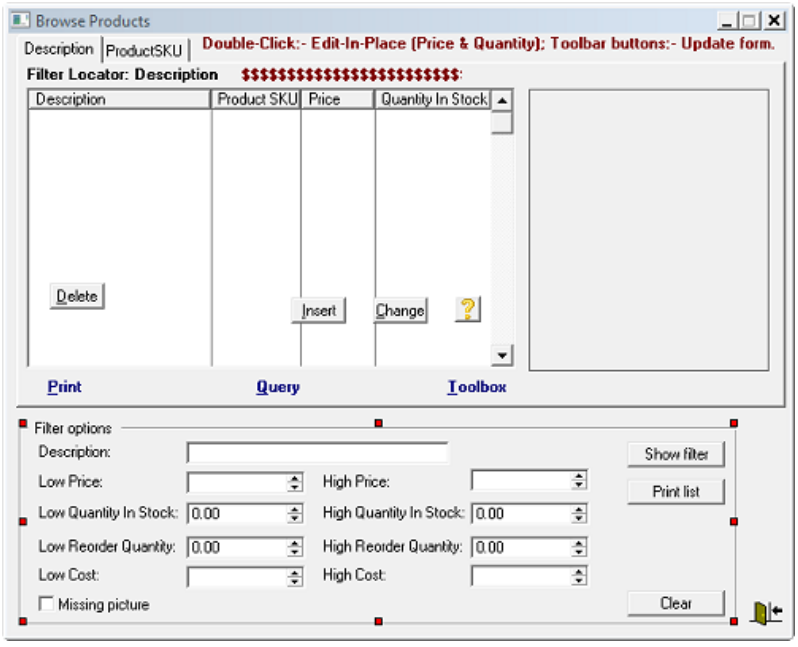

Figure 4. Adding the controls to the browse (view full size image**)**

One more button

I add a Clear button to reset all the filter fields and one line of embed code for the accepted event on the button:

```
CLEAR(QueryGrp) ! clear all the group fields for the user
```

Enhancing the browse actions

Since I'm not embedding any code in the filter field controls I need another way to force the browse to reset whenever a filter value changes. This mechanism already exists in the templates and is called Reset fields. I right click on the browse

and select the Actions tab. I then click on the Browse Box Behavior button and from the resulting window (Figure 5) I click on Reset Fields. I add each of my filter controls to the Reset Fields list.

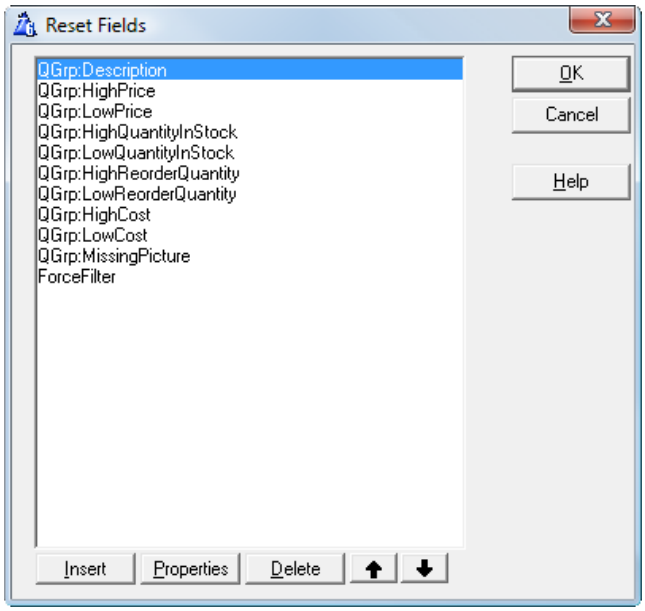

Figure 5. The reset fields

That takes care of the default tab. Next I select the conditional tab and add the same reset fields to the second tab's conditions. The BrowseClass requires each browse tab have its own reset fields.

That takes care of all the screen formatter tasks.

The Source code embed

Reset fields are fields watched by the BrowseClass and will reset the browse any time the value of any reset field changes. I just need to know where this reset action takes place so I can trap the reset and change the filters just before the ViewManager sends the Prop:Filter to the View structure. And that brings me back to the ApplyFilter method.

Brw1.ApplyFilter

Open your source editor and find the virtual procedure called Brw1.ApplyFiler (using alt F3 to search). See Figure 1 (above).

I will add local data to this procedure to make my code easier to read. I will also focus on making the code easier to maintain by using EQUATEs.

The SetFilter method

The ViewManager has a method called SetFilter. Here's the Clarion documentation:

> SetFilter(*expression [, id]*), VIRTUAL
>
> *expression*
>
> A string constant, variable, EQUATE, or expression that contains a FILTER expression. See FILTER in the Language Reference for more information. If expression is null (''), SetFilter deletes any existing filter with same id.
>
> *id*
>
> A string constant, variable, EQUATE, or expression that uniquely identifies (and prioritizes) the filter so you can apply multiple filter conditions, and so you can replace or remove filter conditions with subsequent calls to SetFilter.

If omitted, the filter gets a default id so that subsequent calls to SetFilter with no id replace the filter expression set by prior calls to SetFilter with no id.

By using the second parameter of SetFilter I can assign each individual filter condition separately and the BrowseClass (actually the ViewManager code) will combine these into a single filter using the AND operator.

The following equates are used for the second parameter. I like equates because they make the code more readable even though I type a little more the first time. I won't use these equates ever again so they belong here in this method where I can see them.

Cut and paste this code in the data section, before the CODE statement:

```
 ITEMIZE(1)
e_FilDesc        EQUATE  ! description
e_FilLowPrice    EQUATE  ! price
e_FilLowQuantity EQUATE  ! quantity on hand
e_FilLowReorder  EQUATE  ! reorder quantity
e_FilLowCost     EQUATE  ! cost
e_FilHighPrice   EQUATE  ! price
e_FilHighQuantity EQUATE  ! quantity on hand
e_FilHighReorder EQUATE  ! reorder quantity
e_FilHighCost    EQUATE  ! cost
e_FilPicture     EQUATE  ! is the picture missing
 END
```

Init the filters

At the start of the code section I init all the filters; later I'll set any specified filters. I add this after the CODE statement:

```
! Init all filters to blank
Self.SetFilter('','e_FilDesc')
Self.SetFilter('','e_FilLowPrice')
Self.SetFilter('','e_FilLowQuantity')
Self.SetFilter('','e_FilLowReorder')
Self.SetFilter('','e_FilLowCost')
Self.SetFilter('','e_FilHighPrice')
Self.SetFilter('','e_FilHighQuantity')
Self.SetFilter('','e_FilHighReorder')
Self.SetFilter('','e_FilHighCost')
Self.SetFilter('','e_FilPicture')
```

Self in this case means the view for the Brw1 instance; SetFilter is one of BRW1's own methods.

High low ranges

The high/low range filters work alone or in combination. Using only a low filter I can set a floor value and using only a high filter value I can set a ceiling value, or I can use both at the same time for a range filter. Only non-blank, non-zero values result in a SetFilter call. Cut and paste this code after the above:

```
! Price filters
IF QGrp:HighPrice
```

```
  Self.SetFilter('PRO:Price <= ' & QGrp:HighPrice,e_FilHighPrice)
END
IF QGrp:LowPrice
  Self.SetFilter('PRO:Price >= ' & QGrp:LowPrice,e_FilLowPrice)
END
```

I construct a string expression using the dictionary field names and concatenate the actual runtime value of my local fields. By using concatenation I don't need to BIND the variables. The Product field names are already BINDed because of the option turned on in the dictionary table properties. I choose not to BIND local variables because the code runs a bit faster and I'd rather use the compiler than the runtime wherever possible. This also reduces potential BIND errors; if I make a coding error in a filter expression it crashes the application.

I repeat the exact same approach for the Quantity on Hand, Quantity for Reorder and the Cost ranges. I don't display error messages because the reset variables will execute on every spin button event. I might, however, want to display a message for conflicting high and low values.

Cut and paste this code after the above:

```
! quantity in stock filters
IF QGrp:HighQuantityInStock
  Self.SetFilter('PRO:QuantityInStock <= ' & |
    Grp:HighQuantityInStock,e_FilHighQuantity)
END
IF QGrp:LowQuantityInStock
  Self.SetFilter('PRO:QuantityInStock >= ' & |
    QGrp:LowQuantityInStock,e_FilLowQuantity)
END
! Reorder filters
IF QGrp:HighReorderQuantity
  Self.SetFilter('PRO:ReorderQuantity <= ' & |
    QGrp:HighReorderQuantity,e_FilHighReorder)
END
IF QGrp:LowReorderQuantity
  Self.SetFilter('PRO:ReorderQuantity >= ' & |
    QGrp:LowReorderQuantity,e_FilLowReorder)
END
! cost filters
IF QGrp:HighCost
  Self.SetFilter('PRO:Cost <= ' & QGrp:HighCost,e_FilHighCost)
END
IF QGrp:LowCost
  Self.SetFilter('PRO:Cost >= ' & QGrp:LowCost,e_FilLowCost)
END
```

The missing picture check box

I added a local checkbox with a validity check of zero / one. The user can check for inventory items added that don't have pictures. Cut and paste this code after the above:

```
IF QGrp:MissingPicture
  Self.SetFilter('LEN(CLIP(PRO:PictureFile)) = 0',e_FilPicture)
END
```

The name of the image is a STRING field. I clip it to compute the length. My expression will be true if there is no picture name in the field. This is a simple logical expression.

Matching description text

The filter string for matching descriptions is a good example of something that generally drives me nuts when writing expressions. The problem happens when I need a literal string inside the string. The following code will use the supplied user description string and perform an INSTRING match of the actual description:

```
IF LEN(CLIP(QGrp:Description))
  Self.SetFilter('INSTRING(' & '<39>' & UPPER(CLIP(QGrp:Description)) & |
'<39>' & ', UPPER(PRO:Description) ) <> 0',e_FilDesc)
END
```

The rat hole I can fall into is not keeping the concept of a literal string and the actual string expression straight in my mind. If you've ever had to declare a string literal containing just one single quote, you know that you have to type four single quotes (""). The first and last quotes begin and end the string, but you need a second interior quote to escape the actual quote that will makes up the string. This is hard code to read. Instead, I have surrounded the user supplied string with '<39>' (a literal quote).

When I'm in doubt I pretend I am not writing an expression but am writing regular compiler code. If I need a quote with compiler code I need it here too. I will agree adding a BIND of QGrp:Description makes for easier coding, but concatenation makes the code solid due to the compiler qualifying the code. String building on the fly means no redundant long logical expressions. I never need to bind values I already know.

Just one more thing

I can prove that all this works on the actual view with a simple bit of code. I add a button to my option group called Show filter, with the following code in Event :Accepted embed:

```
MESSAGE('Filter = ' & BRW1::View:Browse{Prop:Filter}, 'Show Filter')
```

BRW1::View:Browse is the view structure for this browse, as generated by the browse template. When I select the button a message box will now display the actual filter string as applied to the view, whether by my Brw1. ApplyFilter method or by the standard QBE extension.

Here's the code in ThisWindow.Init that assigns the view to the browse:

```
BRW1.Init(?Browse:1,Queue:Browse:1.ViewPosition,BRW1::View:Browse,Queue:Browse:1,Relate:Products,SELF)
```

A print button

Let's say the flower shop bookkeeper loves the new QBE but needs to print the resulting list too. A quick trip back to the procedure list shows PrintPRO:KeyproductSKU is the only product report. "It will do" says the bookkeeper. Provided, of course, that the report uses the same filter as the browse. Next time I'll show how easy it is to get a report to use the browse's filter.

Download the source

Paul Blais has been a Clarion developer since CPD version 2003. He became a full time independent Clarion Developer in 1998. In 2000 he merged his life and business with his wife Barabra. The merger yielded 3 dogs, one horse, 3 vehicles, and

Organizational Development Strategies, Inc.. When not writing code he and his wife can often be seen aboard Bright Eyes sailing the Chesapeake Bay.

Reader Comments

Posted on Saturday, April 12, 2008 by S Jayashankar

Hi Paul,

Since SetFilter processes the different filters in DESCENDING order of ID (which is alphanumeric) and the default ID for the SetFilter method is 5, it is better that the equates are NOT Itemized from 1 but from either 11 if it is to have lower priority than the Browse Filter set up on the template or 61 if it is to have a higher priority. Also, the itemized list should be set up in DESCENDING order of priority (lowest priority on top and highest priority at the bottom).

Regards

Posted on Saturday, April 12, 2008 by Paul Blais

Yes, the order can make a difference but in this case it can't. I wanted a simple example and getting into that detail just was not possible.

When using filters to a back end you really do want to consider the order. Engines like SQL will optimize and you can benefit greatly. The Clarion View engine also does a great job at this.

Posted on Sunday, April 13, 2008 by S Jayashankar

But your Itemize(1) would still cause some of your filters to appear BEFORE the template-set Browse Filter (if any). Maybe it does not apply to your example as it has no template-set Browse Filter but it is better users are aware of the pitfalls. Also, with ISAM databases, using UPPER(<Column>) in the filter ensures that the Case-Insensitive key is used by the VIEW Engine.

Posted on Sunday, April 13, 2008 by Paul Blais

For SQL you can use a Prop:SQL and probably do a better job for complex filters. For the most part the priority is really an after the fact enhancement. The worst problem I've ever found was when I was using a BIND on a function in TPS mode. The app supported both SQL and TPS so the difference was to say the least dramatic. It's not the common situation however. For the day in and day out filters it's not a problem if you already have a proper key defined with range limits. This example uses none of that as the key defined in the app was the SKU key (not my choice). This approach drops into existing code without side effects so it's a great upgrade choice that codes fast (as in cheap).

The main idea is that it works better than a TakeRecord embed in any situation. I know of no case where the TakeRecord is better for anything. I never use it myself.

Stay tuned for part 2 of the article. It gets into some other issues.

Add a comment

Clarion Magazine

Loosely Coupled .NET Applications: Understanding Reflection

by Dave Harms

Published 2008-04-10

I started off this article series by noting that the .NET framework included something called *reflection*, and by suggesting that reflection was pretty dang cool stuff that could completely change the way you create your applications. And then I rambled on about inheritance and interfaces, and didn't say a word about reflection except to explain that it was important to understand inheritance and interfaces first before going on to the role of reflection in creating loosely coupled applications. And so it is.

In this article and the two that follow I *will*, at least briefly, explain what reflection is; more importantly I'll provide a series of examples and a toolset you can use to leverage the power of reflection to create flexible, adaptable applications.

By way of illustration I'll introduce a small Clarion# application that contains little more than a class to represent an invoice, a couple of classes to represent various ways of applying taxes to invoice items and some test code to excercise those classes. This is a console application, so there's no user interface code to interfere with the basic concepts I want to communicate. And this isn't something you'd want to use in the real world but I hope it will inspire you to think about how you can apply these concepts to your Clarion# applications.

No article series on new technology would be complete without a few buzzphrases, so I'll along the way I'll introduce the terms *inversion of control* and *dependency injection*. Both of these terms describe, in part, how you can create applications out of components that you assemble, at runtime, into a working whole. But before I get there I need to explain why this kind of flexibility matters.

Why flexible matters

This article, and the two that follow, may not be for you. If you create simple applications that perform straightforward tasks in unvarying ways, then you probably don't need to read on. But in the world most of us inhabit, applications often have to serve multiple aims and varying customer requirements.

Let's say you have a C6 application that serves 90% of your customers well, but the other 10% need somewhat different functionality. Now you have an application that has to present two or more behaviors. The problem with that C6 application is simply this: the behavior of the application is determined at *compile* time. That is, in order for any block of code to execute the compiler has to know that the code exists, and where it exists. (That may seem obvious, but as I'll show a little later .NET removes the requirement that your app know *ahead of time* where to find the code it will run.)

Customizing applications

There are several ways to customize an application's behavior for different needs/customers. One is to rely on custom templates and configuration options. You set your custom template options the way you want, you generate the application code, you compile. Of course you need a different compiled EXE for each behavior, and therefore a different install, and managing all of that can be a headache. If you don't want to go the template route you might consider embodying all functionality in a single application and using some configuration options (say in an INI file or the registry) to determine which code gets executed.

Perhaps the least desirable way to manage varying behaviors is to fork the source code; now you have multiple code bases

to manage, although version control software can make this task easier. You are, of course, back to creating multiple EXEs and/or DLLs again.

A variation on this last approach is to isolate the unique code to a specific DLL which uses standardized functions/classes, so that all you have to do to is include the appropriate version of the DLL with your install.

This last approach comes closest to the flexibility offered by reflection in that it allows you to plug in specialized behavior without having to recompile the entire application. But you have to be very careful about the structure of each version of the DLL to maintain compatibility. Even though the DLL isn't linked in to your EXE, the LIB that matches the DLL is; if a DLL gets out of sync with the LIB that's used when the EXE is built then the best you can hope for is a GPF and the worst is a subtle and damaging bug.

All of these approaches offer some level of flexibility, but the beauty of .NET reflection is that it removes the requirement to hardwire your classes. Instead, much of what the C6 compiler does at compile and link time, Clarion# can do at runtime. And all of this is made possible by something called *metadata*.

Reflection and metadata

One of the main features of the .NET platform is that all .NET languages compile to Common Intermediate Language, or CIL. This CIL format is human-readable, but I'm not aware of any programmers who actually write CIL. Here's what a disassembled C# Hello World program looks like as CIL code (and yes, I am letting the lines get cut off on the right side - if you want to see everything use the printer-friendly view):

```
// Metadata version: v2.0.50727
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                    // .z\V.4..
  .ver 2:0:0:0
}
.assembly ClarionMag.Hello
{
  .custom instance void [mscorlib]System.Runtime.InteropServices.ComVisibleAttribute::.ctor(bool) = ( 01 00 00 00 00 )
  .custom instance void [mscorlib]System.Reflection.AssemblyTrademarkAttribute::.ctor(string) = ( 01 00 00 00 00 )
  .custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string) = ( 01 00 00 00 00 )
  .custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::.ctor(string) = ( 01 00 10 43 6C 61 72 69
6F 6E 4D 61 67 2E 48 65   // ...ClarionMag.He
                                                                  6C 6C 6F 00 00 )                        // llo..
  .custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(string) = ( 01 00 00 00 00 )
  .custom instance void [mscorlib]System.Reflection.AssemblyConfigurationAttribute::.ctor(string) = ( 01 00 00 00 00 )
  .custom instance void [mscorlib]System.Reflection.AssemblyDescriptionAttribute::.ctor(string) = ( 01 00 00 00 00 )
  .custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string) = ( 01 00 10 43 6C 61 72 69 6F
6E 4D 61 67 2E 48 65   // ...ClarionMag.He
                                                                  6C 6C 6F 00 00 )                        // llo..

  // --- The following custom attribute is added automatically, do not uncomment -------
  //  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute::.ctor(valuetype [mscorlib]System.
Diagnostics.DebuggableAttribute/DebuggingModes) = ( 01 00 07 01 00 00 00 00 )
```

```
  .custom instance void [mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01
00 08 00 00 00 00 00 )
  .custom instance void [mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() = ( 01 00 01
00 54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx
                                                                63 65 70 74 69 6F 6E 54 68 72 6F 77 73
01 )       // ceptionThrows.
  .hash algorithm 0x00008004
  .ver 1:0:3016:28100
}
.module ClarionMag.Hello.exe
// MVID: {FB603737-153F-4A42-AD76-A17200F5F120}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x00370000


// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit ClarionMag.Hello.MainClass
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       13 (0xd)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello World!"
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  } // end of method MainClass::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method MainClass::.ctor
```

```
} // end of class ClarionMag.Hello.MainClass


// =============================================================

// ********** DISASSEMBLY COMPLETE ***********************
// WARNING: Created Win32 resource file C:\temp\hello.res
```

I chose a C# example (created in the Clarion.NET IDE) rather than Clarion# only because Clarion's Hello World is quite a bit longer, at least in part because of the OO wrapper code needed to support Clarion#'s procedural capabilities.

If you attempt to decompile, say, a Clarion (not Clarion#) EXE or DLL you'll get something decidedly less readable. And it's not just the format of the dump that makes the difference, it's the actual content. A Clarion EXE or DLL contains relatively little information to help you figure out what's going on; by contrast, you can actually learn something about a .NET program by examining its CIL code.

All .NET applications are made up of assemblies (EXEs and DLLs), but unlike native code EXEs and DLLs assemblies contain not just code and data but also information about the code data. This is called *metadata*, or data about data.

> NOTE: Metadata brings many benefits but it also has a downside, and that's a certain loss of privacy. Usually when developers realize this information can be extracted from their DLLs and EXEs they begin to get nervous about protecting their intellectual property; if that's a concern for you then you'll want to look into .NET obfuscators which scramble your code without affecting its functionality.

Reflection is the ability to read and make use of this metadata at runtime.

For instance, reflection makes it possible for the Clarion# screen designer to make use of all sorts of custom controls, even those you create. By contrast, support for any new controls in the Clarion window formatter requires a recompile of at least part of the IDE. Simiarly, since you can discover information about classes on the fly, you can also create an interface to display class information. Again, this is an important feature of the IDE. Another important IDE feature is Intellisense - reflection makes it possible to display information about available class methods and properties.

Just as you can discover information about classes and methods, you can also create instances of classes and call those methods. It's this final aspect of reflection that's the focus of this article, but first I need to demonstrate one particularly important feature of reflection.

Creating a class from a string

Consider a class called Invoice; it doesn't do anything yet, but it will:

```
Invoice          class,public
                 end
```

Even though this class does nothing, let's assume I want an Invoice instance. That's easily enough done. Here's the declaration:

```
inv     Invoice
```

And then somewhere in my code I'll create the instance:

```
inv = new Invoice
```

But here's a different way to create the invoice instance:

```
inv = System.Activator.CreateInstance(|
  'ClarionMag.Reflection.Example',|
  'ClarionMag.Reflection.Example.Invoice').Unwrap() |
  tryas Invoice
```

The call to System.Activator.CreateInstance takes two *string* parameters: the first is the assembly namespace, and the second is the type of the requested class instance. Think about what's happening here. I'm not passing *type* information to CreateInstance, which is what a compiler (like the C6 compiler) would require if I wanted to create an instance of a class. Instead I'm simply passing a string, and CreateInstance looks for a class that matches that string. When it finds the class it discovers the type information from the metadata and creates the object. The tryas operator attempts to cast the object to an Invoice data type.

> **NOTE:** The string passed to CreateInstance does not have to specify the Invoice class, only a class that can be legally mapped to the inv reference; the string could as easily describe a class derived from Invoice, or, if Invoice were an interface instead of a class, any class implementing that interface.

It's actually not very likely that you'll use CreateInstance directly in your applications, but the ability to extract type information, demonstrated by CreateInstance, is key to creating applications which are highly flexible and configurable. There are many other reflection classes besides Activator; for more information see the System. Reflection Namespace on MSDN. Mark Sarson has a nice little example of using reflection with Clarion# to examine the contents of an assembly; be sure to check that out as well.

Summary

I've described the problem of creating applications that can be all things to all customers; in Clarion, there really is no one good solution to this kind of problem. I'm not sure that .NET offers the perfect alternative, but certainly .NET's reflection capabilities open up whole new worlds. I'll be more specific about how to use reflection to build adaptable, configurable applications in the final article in this series; meanwhile there's one more step to take, and that's to set up a suitable demo application.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of *Developing Clarion for Windows Applications*, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Reader Comments

Add a comment

Clarion Magazine

The Clarion.NET FAQ

by Dave Harms

Published 2007-11-17

In this Frequently Asked Questions (FAQ) page I'll attempt to answer at least some of the questions that have been raised about Clarion.NET and Clarion#. If you log in you can post your own comments and questions below.

Recent additions are marked in red; recent deletions are

~~marked with a strikethrough~~.

Terminology!

One of the problems in discussing Clarion.NET is finding a meaningful term for traditional (non-.NET) Clarion applications. Some developers use the term *Win32*, but that isn't always helpful since .NET applications can also be Win32 (or Win64).

For a while I described Clarion applications as WinAPI apps, but nobody really seemed to like that. In any case the convention now seems to be to describe traditional Clarion apps as simply "Clarion", and Clarion.NET apps as Clarion#, after their respective languages.

Isn't it too easy to confuse Clarion and Clarion#?

Probably. Oh well.

What's the difference between Clarion# and Clarion.NET?

Clarion# is the language; Clarion.NET is the new IDE. But that's potentially confusing too because the new IDE, when complete, will be used for both Clarion and Clarion# programming. Jan van Dalen suggested the new IDE be called "Clarion Studio" - I like that, and I hope it catches on.

Is Clarion.NET available now?

Yes, the first beta was released to subscription program participants on Saturday, Nov 17, 2007, and more betas have followed.

What is .NET, anyway?

From Wikipedia:

> The Microsoft .NET Framework is a software component that can be added to or included in the Microsoft Windows operating system. It provides a large body of pre-coded solutions to common program requirements, and manages the execution of programs written specifically for the framework. The .NET Framework is a key Microsoft offering, and is intended to be used by most new applications created for the Windows platform.

I'll just add (for now) that .NET is an object-oriented framework which accommodates a great variety of programming languages.

Isn't .NET all about web development?

You might think so, given the name of the platform. While .NET contains many web/Internet-related classes, it also has extensive support for desktop development. As well, there's a version of the framework for mobile devices.

.NET is a comprehensive development and runtime environment suitable for most kinds of programming, but not usually for low level hardware-oriented stuff (although you *can* get an 80386 assembler that generates .NET code).

What's the difference between Clarion# applications and the Clarion applications I write now?

There are a number of differences. One of the biggest is that the Clarion language (for Windows) is highly dependent on the Windows API. For instance, when you open a window, that window is created by the Clarion runtime library via calls to the Windows API. When you create a new variable or class instance, memory is allocated via the Windows API. This is procedure-oriented coding, and procedure-oriented applications (and operating systems) tend to be hardwired - they have only set ways of doing things. For instance, the Clarion window formatter only has a specific set of controls available, and you can't add your own custom widgets. You can use add-on components like ActiveX controls, but as anyone who's tried it in Clarion can tell you, it isn't all that easy either.

In a Clarion application you often have to work hard to make different functional units of code work together because the Windows API isn't really geared to the concept of components.

Applications written for .NET, however, use the extensive .NET Framework class library rather than the Windows API. In .NET it's all about objects. You can assemble an application out of different classes and components much more easily because .NET provides a congenial framework for all these things to work together.

Is the .NET Framework that much better than the Windows API?

Yes, in almost every way. That's not toadying to Microsoft - it's just the evolution of programming. The Windows API is disorganized - you need to know what you're looking for or you'll get lost very quickly. The .NET Framework library is organized into namespaces. For instance, security classes are grouped together, as are diagnostic classes, data access classes, etc.

The .NET framework library is much more massive than the API and is growing larger. It provides a whole lot of code that you'd otherwise have to write yourself. There are many other benefits to the library but many of these are better described in the context of the Common Language Runtime (CLR).

Okay, I'll bite. What's the Common Language Runtime?

The CLR is an essential part of .NET. It's the layer that sits between you and the hardware, and which provides important services to your applications including, but not limited to: memory management; thread management; exception handling; garbage collection; security; introspection and reflection. The CLR means many tasks are easier in .NET. For instance, if you NEW something you don't (usually) have to DISPOSE it - the CLR's garbage collector will detect when the object is no longer in use and will clean it up automatically.

Are there any other weird acronyms I should know?

Well, there's CIL, the Common Intermediate Language (often just called IL). The CLR doesn't actually run the code you write; instead, all .NET language compilers translate their code into this CIL (IL) bytecode, and that's what the CLR executes.

So .NET is a giant interpreter? This seems like a step back from true compiled languages - didn't CPD generate pseudocode **which had to be interpreted?**

You're right - CPD did in fact generate pseudocode, and it's a fair analogy. The advantage of IL is that since all .NET languages compile to an intermediate language, they all share a common platform. You can take some classes compiled in, say, Fortran.NET (no, I'm not kidding) and easily use them in C# or Boo or Clarion#.

What is Clarion#?

Clarion# is the .NET version of the Clarion programming *language*. The Clarion IDE for .NET is, at least at present, called Clarion.NET. But a lot of developers already use the terms Clarion# and Clarion.NET interchangeably.

Clarion is both a procedural and an object-oriented language, but .NET is an OO framework - can I still write procedure code in Clarion#?

Yes, you can still write procedural code in Clarion#. Your procedures are converted to object-oriented code when they're compiled to IL code, but you don't need to know or care about that if procedural code is what makes you happy. You will, however, find it much more difficult to make the most of the .NET platform and all the available classes unless you are willing to learn at least a little object-oriented programming.

Are Clarion 7 and Clarion# the same product?

No, they are separate products, but they share the same integrated development environment (IDE).

Does that mean if I buy Clarion 7 and Clarion# I'll end up with just one installed IDE able to work with both platforms?

Most likely, although there may be some versioning issues that make that more difficult, at least during the beta process. So for a time you will have two IDEs installed.

What can I do with Clarion.NET/Clarion# beta in the first release?

The beta includes a template wizard you can use to generate a .NET application, but since those templates run in C6, not the new IDE (AppGen isn't ready yet), you can't yet create and maintain a .NET APP file in the same way you do in C6.

As of build 3040, the new IDE ships with a fully functionaly dictionary editor. The AppGen is scheduled to be demo'd at the Aussie DevCon at the end of May, and Bob Z has indicated AppGen will not be held if it's ready for beta release before then; meanwhile you can hand code not just desktop applications, but also ASP.NET and mobile applications.

There are a number of example solutions shipped with the beta. These include demonstrations of connecting to a database with ADO.NET, displaying a BLOB image onscreen, a mobile app, data binding, drag and drop, a FOREACH with QUEUE example, glass buttons, new listbox features, the SCHOOL application, mixing Clarion# and C#, a simple web service, a "Hello world" ASP.NET web application, a .NET remoting example, the PEOPLE app with data grid and browse procedures, and a screen capture utility.

Since Clarion# is a full .NET producer/consumer language you have full access to all of the .NET framework library as well as the rich supply of third party .NET tools.

What will I be able to do with Clarion.NET/Clarion# in the long run?

When complete Clarion.NET/Clarion# will include templates to generate desktop (WinForms) applications, web (ASP.NET) applications, and mobile (Compact Framework) applications.

What is the Compact Framework?

The Compact Framework is .NET for mobile devices, and includes about 30% of the full framework plus classes specific to mobile devices, and takes up about 1/10 of the space, mostly due to file compression.

What is ASP.NET?

ASP.NET is Microsoft's Active Server Pages web application framework for .NET. You create ASP.NET pages using a development approach similar to desktop development: you place controls on a page, and write code for those controls; ASP.NET then renders the pages accordingly and executes the code on the server side when data comes back from the browser.

How hard is it to write .NET applications?

Writing Clarion# applications can be both easier and harder than writing Clarion apps, and the comparison between the two brings to mind the differences between DOS and Windows API development. You could argue that Windows development was a lot harder because you had to do so much more to create even a simple Hello World application (at least in C, if not in Clarion). On the other hand, Windows provided standard capabilities like a windowing library; in DOS you had to write or otherwise obtain a windowing library to achieve a consistent, accessible user interface. In DOS you had to know how to talk to every printer you wanted to use; in Windows you talked to the printer driver, and the printer driver sorted out the back end. Similarly .NET does a lot of the heavy lifting you now have to code in Clarion apps, leaving newer and more complex tasks to your wily programming brain.

One big difference between the Windows API and .NET is that the latter is exclusively object-oriented. To get the most out of Clarion# apps you will definitely want some basic OO programming knowledge. With that knowledge in hand, I think you'll find .NET easier and safer to use than the Windows API. If you've had to deal with ActiveX or (heaven help you) directly call COM functions you'll truly find .NET an easier place to code.

.NET introduces some new language concepts that may take some getting used to, such as delegates, which are a sort of type-safe object-oriented callback mechanism.

Is .NET open source?

.NET is not open source, but Microsoft is in the process of making the source code for some parts of the framework public.

Will my third party tools work in .NET?

The majority of third party tools will need to be ported to Clarion#. Templates that don't generate any source code and do not depend on a particular template chain are likely to work without modification, but there aren't many that fall into that category. You're probably best off assuming you'll need new versions until you hear otherwise from the vendor.

I've often said that .NET is a two-edged sword for third party vendors. If what you provide is readily available as part of the .NET framework, then zing!! off goes your head as you step into .NET land. On the other hand, if you have a great product and you can port it to .NET, you're ready to carve a swath through a programming market that numbers in the millions of coders.

Will my customers want .NET and if so why?

Some customers will want .NET just because it's a buzzword. That's an easy sale.

Other customers may or may not know whether they want .NET. Clearly what they want is software that does what they need it to. If their needs run to eye candy or very unique user interfaces, then .NET presents some distinct advantages. There are a bazillion custom controls out there for .NET, all of which you can use with Clarion#. And there are many class libraries that handle important behind-the-scenes tasks as well.

I find it difficult to overstate the importance of ready access to all of that existing code. With Clarion apps you have to be concerned about prototyping functions, register passing conventions, the arcana of COM, etc. etc. With .NET you just drop in the library and start to use it, no matter what language it's written in.

.NET offers programming benefits as well. For instance, your code compiles to IL code which is run under the watchful eye of the CLR. That means the CLR can detect problems with your code and present far more detailed information to you than you get from, say, a GPF in a Clarion application. This makes debugging easier and faster.

Can I run .NET apps on Linux or the Mac? What is Mono?

Mono is a Novell-sponsored project to port the .NET platform's functionality to multiple platforms, including Linux, Mac OS X, Solaris, BSD, and Windows. Mono necessarily lags behind Microsoft's efforts, and currently has completed support for .NET 1.1 and mostly-complete support for .NET 2.0.

Versions? What are all these .NET versions?

Microsoft released .NET 1.0 in 2002, and 1.1 in 2003. You may see some computers with 1.1 as the latest version, (and some computers without .NET installed at all) but the standard at present is .NET 2.0, released in 2005. Clarion# targets 2.0 apps - there was talk in the early days of 1.1 being supported as well but the only reason I can see for supporting 1.1 is for Mono compatibility, given that Windows Forms 2.0 support is now scheduled for Mono 2.5, which does not have a release target.

For most of us, .NET 2.0 will be the minimum.

What about .NET 3.0? Or 3.5?

The first thing to keep in mind about .NET 3.0 is that it is not a replacement for .NET 2.0. It's a bunch of new stuff added to 2.0, including Windows Presentation Foundation, Windows Workflow Foundation, Windows Communication Foundation, and Windows Card Space. The base class library is unchanged from 2.0.

.NET 3.5 uses the same CLR as 2.0 but it adds some new stuff to the base class library, in particular support for the LINQ query language. 2.0 apps will still run fine on 3.5.

SoftVelocity is "adding support for 3.0/3.5" but no timeline has been indicated.

Will I need to learn C# or VB.NET?

You will not need to learn C# or VB.NET or any other .NET language, but you may want to. In particular there's a lot of C# source code out there, and you may want to adapt some of it to your own uses. If you can read object-oriented Clarion code you won't have much trouble reading C#.

What is ADO.NET?

ADO.NET is Microsoft's data access layer for .NET, and consists of data providers (i.e. drivers) and DataSets. A DataSet is a set of objects that model the database elements (tables, views, columns, rows, relations, etc.).

Will my Clarion (.clw) programs compile in Clarion#?

While much of the Clarion language is unchanged in Clarion#, it's unlikely that any single Clarion application big enough to do useful work will compile as Clarion# code without modification.

What will I have to do to port my apps to .NET?

Porting applications to .NET is a bit of a gray area at the moment. Theoretically it can be done; the question is, is it worth the work? Clarion apps are built on a traditional, client-server model. Is this a good approach to take into the .NET world? Do we really want ABC.NET? Perhaps a multi-tier design would be more appropriate, particularly one where you could easily reuse your business logic in desktop, web, and mobile versions of your application. SV has alluded to this kind of design but it isn't clear yet what kind of desktop application templates will be included with Clarion#.

Can I mix and match .NET objects with Win32 API objects easily?

You can include WinAPI code in a .NET app and vice versa. Easy is a relative term. See Wade Hatler's series of articles.

Can I get my Clarion 7 and earlier programs to use .Net components I create using Clarion# or other .Net languages?

Most likely you could (see the article series above) but I think this would be a stopgap measure at best.

How secure is .NET code? Can it be easily decompiled?

Code security is a legitimate concern in .NET and yes, IL code can be decompiled much more easily than native Windows executable code. .NET obfuscators alter label names and use various code scrambling approaches to make it very difficult for anyone to make sense of the decompiled result. Or you could just hire someone who's a natural at writing unreadable code.

I've heard .NET programs run slower than native code. Will I notice the difference?

~~.NET uses just-in-time (JIT) compiler technology, meaning that IL code is compiled to native code the first time that IL code is needed by the application. That means there is a small startup penalty but once the code is compiled it runs just like any other native code. Theoretically code produced by a JIT compiler can~~ outperform ~~code issued by a standard compiler because the JIT compiler can tailor the code to the hardware. That said, .NET applications often do seem to run slower and take more memory.~~

.NET uses just-in-time (JIT) compiler technology, meaning that IL code is compiled to native code the first time that IL code is needed by the application. If you take any two blocks of code that do the same thing in Clarion 6 and Clarion# you'll get approximately the same performance. On the one hand the JIT compiler can tailor its output to your computer's hardware, so you may see some performance improvements; on the other managed code may exact a penalty, usually no more than 10-15%.

Why then do some .NET programs actually seem to run slower and take more memory?

Without two functionally identical versions of the same program available for profiling, one .NET, the other not, it's almost impossible to say. Given that code execution speed is a wash the differences most likely come down to sloppy programming and/or feature bloat. You can write big, slow apps in just about any language.

Can I include a .NET runtime with my apps so I don't have to require my customers to install .NET?

There is a tool called the Salamander .NET Linker, Native Compiler and Mini-Deployment Tool that will do just this. It's a bit expensive, and I don't know how well it works. Apparently Thinstall will also create self-contained .NET installs.

If both new Clarions deliver desktop applications why should I buy both? Would Clarion# not be enough?

I'll assume here you mean *after* AppGen is released for C7 and Clarion#. While you can create real desktop apps already with Clarion#, most developers will want to use AppGen for larger apps.

So yes, you can create desktop apps with both. Why would you still want C7? Here are a few reasons:

- Better productivity with the new IDE, as compared to C6
- Ability to work with different versions of Clarion within the same IDE
- C7's runtime improvements including eye candy and Unicode/ClearType support.
- Stability - templates are well established and the runtime is solid
- Potentially smaller installs - no need for the .NET runtime

Can a Clarion# class inherit from a C# class?

Definitely. And vice versa. The same goes for all .NET languages.

Does Clarion# support generic types?

Generics are supported, although there are still a few compiler bugs being ironed out as of March 2008.

What are .NET's minimum requirements?

According to Microsoft, the minimum requirements for the .NET 2.0 redistributable are:

- 400 Mhz processor (800 Mhz recommended)
- 96-128 Mb memory (256 or better recommended)
- 280 Mb hard disk space (610 for 64 bit), 1 gig recommended
- 800x600 256 color, 1024x768 high color recommended

As with most Microsoft platform requirements, you're probably not going to be very happy at the low end of the spectrum. I suggest you take the "recommended" values as the minimum values.

Supported x86-based operating systems:

- Microsoft Windows 98
- Microsoft Windows 98 Second Edition
- Microsoft Windows 2000 Professional with SP4
- Microsoft Windows 2000 Server with SP4
- Microsoft Windows 2000 Advanced Server with SP4
- Microsoft Windows 2000 Datacenter Server with SP4
- Microsoft Windows XP Professional with SP2
- Microsoft Windows XP Home Edition with SP2
- Microsoft Windows XP Media Center Edition 2002 with SP2
- Microsoft Windows XP Media Center Edition 2004 with SP2
- Microsoft Windows XP Media Center Edition 2005
- Microsoft Windows XP Tablet PC Edition with SP2
- Microsoft Windows XP Starter Edition
- Microsoft Windows Millennium Edition
- Microsoft Windows Server 2003 Standard Edition
- Microsoft Windows Server 2003 Enterprise Edition
- Microsoft Windows Server 2003 Datacenter Edition Microsoft Windows Server 2003 Web Edition

x64-bit based systems

- Microsoft Windows XP Professional x64 Edition
- Microsoft Windows Server 2003, Standard x64 Edition

- Microsoft Windows Server 2003, Enterprise x64 Edition
- Microsoft Windows Server 2003, Datacenter x64 Edition

Itanium-based systems

- Microsoft Windows Server 2003 with SP1, Enterprise Edition for Itanium-based Systems
- Microsoft Windows Server 2003 with SP1, Datacenter Edition for Itanium-based Systems

Should I be learning C# or VB.NET, or some other .NET language?

Although Clarion is a full-fledged .NET language, it probably will be to your advantage to learn at least one other .NET language. There's a wealth of .NET programming information out there, and the vast majority of books and articles deal with either C# or VB.NET. So which language should you learn?

VB.NET in general has more Clarion-like syntax; there are certainly differences, but VB.NET is more of a "plain English" programming language. C# on the other hand has C-like syntax which isn't to everyone's liking. It's also easier to make non-obvious mistakes with C#. On the other hand, C# is the .NET reference language, so you can expect it to support all the latest .NET features, and it's generally a better source of programming examples.

If you have C, C++, or Java experience, choose C#. If you've never worked in a language with C-like syntax then you'll probably be better off with VB.NET. Carl Barnes recommends Programming VB .NET: A Guide For Experienced Programmers, which is also available as a free download - look for the Free eBook Download link on that page.

Why should I choose Clarion# over Visual Studio and VB.NET or C#?

Clarion developers have enjoyed the benefits of code generation since the days of CPD 2.0. And when the Clarion# AppGen and templates are ready, I think it'll be easy to see the productivity advantage in Clarion#. But the Clarion# AppGen isn't ready yet, and there are no shipping Clarion# templates. Until that happens, why should you choose Clarion# over VB.NET or C#?

First, let me deal with the reasons to use VB.NET or C# instead of, or in addition to, the Clarion# beta. Obviously both those languages are available in gold release, and have been for some years, while Clarion# is, well, in beta. So you can expect fewer bugs in VB.NET and C#, and more complete support for many .NET features. And Visual Studio is a more evolved hand-coder's environment, at least at the moment, with extensive add-in support.

There are, however, some important reasons for choosing the Clarion.NET beta over (or at the very least in addition to) VB.NET and/or C#:

- Language familiarity - you can start getting up to speed on .NET using a language with which you are familiar
- QUEUEs - although .NET has extensive support for collections, queues are still a dead-easy way to manage lists in memory, and a terrific feature of the language.
- Reports - The report designer isn't yet feature complete, but the report structure in Clarion# is basically the same as it is in Clarion. Reporting is one of Clarion's great strengths. Creating reports in other .NET languages typically means buying add-on products.
- File access - you have access to all the file drivers, including TopSpeed and Clarion files.
- File processing - you can still use Clarion's file access grammar (SET/NEXT etc.) if you want to.
- String handling - Strings in .NET are massively different from Strings in Clarion. The ClaString class provides compatibility with Clarion string handling code.
- Preparation - although SV has indicated a C# and/or VB.NET template chain is a future possibility, you can be sure that the first template sets will be for Clarion#. As with Clarion, the better you know the Clarion# language, the better you'll be able to take advantage of the templates and the corresponding class libraries.

Are there bugs in the beta? Sure, it's a beta. Don't buy in if you're not ready to deal with that fact. But you can already do

some pretty cool stuff with the first beta, and the compiler is pretty solid.

Additional reading

- Clarion Magazine articles on Clarion.NET
- All Clarion Magazine articles related to .NET
- Clarion Magazine articles on mixing Clarion 6 and .NET
- How to subscribe to Clarion Magazine
- CapeSoft's Clarion.NET FAQ
- Randy Rogers' Clarion# Examples

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of *Developing Clarion for Windows Applications*, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Reader Comments

Posted on Saturday, November 17, 2007 by Wolfgang Orth

If both new Clarions deliver desktop applications - why should I buy both? Would Clarion# not be enough? It seems to be the most evolved as it covers Desktop, somehow Web-applictions and mobile devices. So the traditional Clarion 7 with its WinAPI-programming is oldfashioned history from now on, even before it got gold? Or am I wrong on this?

.....

Posted on Sunday, November 18, 2007 by Dave Harms

Thanks Wolfgang - see answer above.

Dave

.....

Posted on Wednesday, November 21, 2007 by Carl Barnes

A concern I don't think I see addressed in the FAQ are the system requirements for the End User computer for a Clarion# application.

A "classic" Clarion Win32 app will typically run on Windows 98 or newer. Probably even Windows 95. No patches are needed. Nothing to download. 3rd party tools can change this.

A Clarion# .Net app requires the 2.0 .Net Framework. That is more picky about systems. Here's a list I found:
Windows 2000 SP3; Windows 98; Windows 98 SE; Windows ME; Windows Server 2003; Windows Vista; Windows XP SP2

Note that XP users must have SP2. I don't see NT 4 or 95.

Some (or many) users will have to download and install 2.0. That's a 23MB file that takes a 280MB of diskspace. I wonder if the 23MB is just the installer and it downloads more.

In summary if you are trying to reach the maximum users, that may be running older hardware, the .Net way might cost you a few.

.....

Posted on Friday, November 23, 2007 by Dave Harms

Thanks Carl - I've updated the FAQ.

.....

Posted on Thursday, December 13, 2007 by Robert Wright

If one would have to more or less re-write ones apps (and learn clarion#) and it looks like C# and VB.Net are high as recommemded languages. What advantage or disadvantage does clarion.net have over visual studio 2005/8? Does any one have a feature comparisson.

Posted on Thursday, December 13, 2007 by Dave Harms

Robert - I've added a response to the FAQ.

Dave

Posted on Wednesday, December 19, 2007 by Robert Wright

How does the tree control in clarion# compare with C6 Clarion, C# and VB.net?

Posted on Wednesday, December 19, 2007 by Dave Harms

Robert,

I've had a look at the declarations in the Clarion.Windows.Forms namespace and don't see anything indicating there's a Clarion-specific tree control. I might have missed it, or something may be in development, or it may be that you'll just use the standard WinForms control or any of the third party products out there.

Dave

Posted on Friday, March 28, 2008 by Stephen Ryan

A clarion specific tree control exists inside the standard clarion list control for dot net.

there is an example in the samples.

Add a comment

Clarion Magazine

The ClarionMag Blog

Get automatic notification of new items! RSS feeds are available for:

XML All blog entries
XML All new items, including blogs

Blog Categories

- »All Blog Entries
- »Clarion 7 Clarion.NET
- »Future Articles
- »News flashes
- »Nifty Stuff

More Clarion# on tap for May

Direct link

Posted Tuesday, April 29, 2008 by Dave Harms

May is looking like another Clarion#-intensive month here at the mag. I'd hoped to be revealing some of the web development work I've been doing with Clarion# (on the way to porting the ClarionMag server code from Java) but these things always take longer than planned. I have, however, been doing a bunch of test-driven development (TDD, if you're short on acronyms this week) with NUnit and Clarion#. I've covered NUnit before, but lately I've been using it intensively, and for certain situations I'm finding it an indispensible tool.

I've worked up a series of examples based on some of the web work I've been doing which I think you'll find interesting. And not coincidentally these examples provide a great lead-in to generic types which are a nifty addition to the Clarion language.

Really, I've become addicted to test-drive development and generics. Be careful; it could happen to you.

Besides all the good stuff in May there's the Aussie DevCon at the end of the month, where Bob Z promises to raise the curtain on the new AppGen. I can't personally make it to the conference this year but ClarionMag will have coverage.

Latest Clarion beta adds generics and other goodies

Direct link

Posted Thursday, April 17, 2008 by Dave Harms

Yesterday was a busy day. SoftVelocity released the latest C7 and Clarion# builds to beta testers, and posted no less than four blog entries covering the new features in the latest release.

First the bad news: no AppGen yet in the latest release (build 3276). But for the first time SV has released some information about the status of internal testing. Key points:

- Some apps (and dictionaries) convert cleanly
- Code generation is being tested
- Editors/window and report designers are being integrated

Z didn't say but I assume that code generation is being tested against the existing ABC templates.

The latest build also sports a completely rewritten IDE parser. Besides some speed improvements this should significantly improve code completion, which hasn't been that reliable or effective in past releases. It also promises a more useful class browser.

The blog entry also mentions a new "file schema pad" which is like the table schematic in the C6 AppGen. The Dictionary Editor and Synchronizer are code complete and in testing.

The Data Diagrammer's new Reports View is a handy way to get a sorted view of all aspects of your dictionary. I won't go into that in any detail here - just check out the blog post.

Generics

One of Z's blog posts coveres the introduction of *generic types*, often just called *generics*. These types use a special syntax with angle brackets, as in

```
<type>
```

where *type* can be any type such a simple long or string, all the way up to any interface or class you create. This may not seem immediately useful to you, but generics are really, really handy. Think of a situation where you don't know ahead of time what kind of object you're going to want to use in a certain situation. In Clarion you have the ANY data type for this situation; in Clarion# you're more likely to use a reference of the type object, since all classes inherit from object (in Clarion#, ANY is actually the Clarion.ClaAny class).

If you create, for instance, a queue of object references, you can assign anything you want to each queue element. But when you want to use one of those objects, all your code will see is the methods and properties of the object class, not of your class. To actually "see" your class the code has to first type cast the object reference back to the right data type. In Clarion# you do this with the tryas operator:

```
myobj = q.obj tryas MyClass
```

Casts are expensive; the runtime environment has to do a bunch of checks to make sure that you can safely cast the object to the specified type. If you attempt the wrong kind of cast you'll get an exception. The power of type casting is also the danger - if you can cast to anything, you can also cast to the wrong thing.

Generics maintain the flexibility of using object references while enforcing a type you specify at runtime. In a sense, generics are simply type parameters; just as you pass data to a method, you pass type information to either the class as a

whole or to a class's method (the class or the method must be set up to accept a type, of course). I'll only discuss the former approach here.

The idea of passing a type to a class is to tell the class that anywhere it has code that references a generic type (often written as <T>, assuming the class/method uses just one generic type) it should substitute the type you just gave it. Here's Z's example:

```
myStringList        List<string>
myIntList           List<Int32>
myPersonList        List<Person>
mp                  Person
   CODE
   myStringList = new List<string>()
   myStringList.Add('Joe')
   myStringList.Add('Jane')


   myIntList = new List<Int32>()
   myIntList.Add(2)
   myIntList.Add(3)
   myIntList.Add(4)
   myPersonList = new List<Person>()
   Loop i# = 1 to 5
      mp = new Person()
      mp.Name = 'Joe' & i#
      myPersonList.Add(mp)
   End
```

Keep in mind that List is really System.Collections.Generic.List. I could show you the C# class declaration, but for fun here's a Clarion equivalent of the List class declaration with the Add method:

```
List            class<T>
Add                 procedure(T item)
                end


List<T>.add         procedure(T item)
   code
   ! Add the passed item to a collection
```

Actually that's *not* what List really would be like in Clarion because List also implements some generic interfaces, which I've left those off to avoid further confusion, and contains a bunch of additional methods. The point of this example (yes, there is a point) is that when you create a new List this way:

```
myStringList = new List<string>()
```

you're telling List that wherever it sees <T> it should now see the string data type. And when you call the Add method, the T parameter is now a string parameter. That means that the class knows the data type you're passing, so it never has to do a cast when you want a string back.

That's a really quick intro to the idea of generics; there are a lot of variations on the theme. Generics are an important subject and one you can be sure will be covered in ClarionMag at some point, hopefully sooner rather than later. At this point generics and delegates are both pretty high on my hit list of Clarion# topics; I haven't sat down to either one yet so if someone out there would like to write an article or two on either of these (or both!) please let me know.

Delayed news items...

Direct link

Posted Wednesday, April 02, 2008 by Dave Harms

Lee White pointed out to me today that I'm well behind on the news items. Apologies, all. I've had my head down working on some cool Clarion# web stuff and I simply forgot. I've also been setting up a Windows Server 2008 web test box and learning a few things about remote administration. It's definitely a different world from the Linux servers I've been using for a decade or so, but the change isn't unwelcome.
