The Microsoft Journal for Developers

msdn® magazine

DECEMBER 2009 VOL 24 NO 12

msdn™ Microsoft®

GET THE POWER TO CREATE A KILLER APP

When an electrochemical reaction animated the dormant cells in a very powerful egg, Gort was hatched. With special powers and abilities to infuse ordinary applications with UIs that have extreme functionality, complete usability and the "wow-factor!", Gort empowers Killer Apps. Go to infragistics.com/killerapps to find out how you can start creating your own Killer Apps.

Infragistics Sales 800 231 8588
Infragistics Europe Sales +44 (0) 800 298 9055
Infragistics India +91-80-6785-1111

Infragistics®

KiLLER APPS. No EXCUSES.

All cats can purr,

Syncfusion™

Deliver innovation with ease

www.syncfusion.com 1 888-9DOTNET

only lions roar.

What does your grid do?

Although many grids share basic features, only Syncfusion's WPF grid has the power and performance to set your application apart. Our grid builds on years of Syncfusion's expertise with high-performance scenarios.

- Support for billions of rows
- Unmatched scrolling performance with large data sets
- Wide variety of styling options

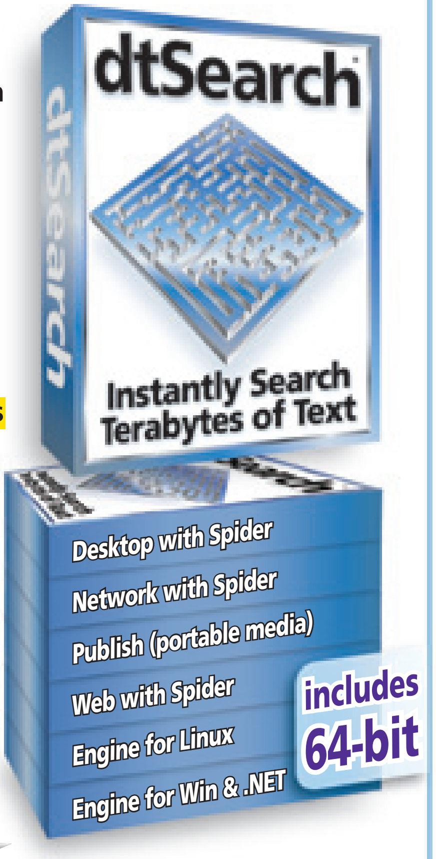
Syncfusion Essential Grid WPF is the fastest WPF grid on the market.

Experience it for yourself at
www.syncfusion.com/wpf-gridperformance

ASP.NET MVC ASP.NET Windows Forms Back Office Silverlight WPF Business Intelligence

Instantly Search Terabytes of Text

- 25+ full-text and fielded data search options (with Unicode support for hundreds of international languages)
- Built-in file parsers / converters highlight hits in popular file types
- Spider supports static and dynamic web data; highlights hits with links, formatting and images intact
- API supports .NET, C++, Java, SQL, etc. .NET Spider API



The Smart Choice for Text Retrieval® since 1991

"Bottom line: dtSearch manages a terabyte of text in a single index and returns results in less than a second" — *InfoWorld*

dtSearch "covers all data sources … powerful Web-based engines" — *eWEEK*

"Lightning fast ... performance was unmatched by any other product" — *Redmond Magazine*

See www.dtsearch.com for hundreds more reviews, and hundreds of developer case studies

Fully-Functional Evaluations

1-800-IT-FINDS • www.dtsearch.com

Microsoft msdn® magazine DECEMBER 2009 VOLUME 24 NUMBER 12

LUCINDA ROWLEY Director
DIEGO DAGUM Editorial Director
KERI GRASSL Site Manager

KEITH WARD Editor in Chief
TERRENCE DORSEY Technical Editor
DAVID RAMEL Features Editor
WENDY GONCHAR Managing Editor

SCOTT SHULTZ Creative Director
JOSHUA GOULD Art Director
ALAN TAO Senior Graphic Designer

CONTRIBUTING EDITORS Don Box, Keith Brown, Dino Esposito, Juval Lowy, Dr. James McCaffrey, Fritz Onion, John Papa, Ted Pattison, Charles Petzold, Jeff Prosise, Jeffrey Richter, John Robbins, Aaron Skonnard, Stephen Toub

RedmondMediaGroup

Henry Allain President, Redmond Media Group
Matt Morollo Vice President, Publishing
Doug Barney Vice President, Editorial Director
Michele Imgrund Director, Marketing
Tracy Cook Online Marketing Director

ADVERTISING SALES: 508-532-1418/mmorollo@1105media.com

Matt Morollo VP, Publishing
Chris Kourtoglou Regional Sales Manager
William Smith National Accounts Director
Danna Vedder Microsoft Account Manager
Jenny Hernandez-Asandas Director Print Production
Serena Barnes Production Coordinator/msdnadproduction@1105media.com

1105 MEDIA INC

Neal Vitale President & Chief Executive Officer
Richard Vitale Senior Vice President & Chief Financial Officer
Michael J. Valenti Executive Vice President
Christopher M. Coates Vice President, Finance & Administration
Abraham M. Langer Vice President, Digital Media, Audience Marketing
Erik A. Lindgren Vice President, Information Technology & Web Operations

Jeffrey S. Klein Chairman of the Board

MSDN Magazine (ISSN 1528-4859) is published monthly by 1105 Media, Inc., 9201 Oakdale Avenue, Ste. 101, Chatsworth, CA 91311. Periodicals postage paid at Chatsworth, CA 91311-9998, and at additional mailing offices. Annual subscription rates payable in US funds: U.S. $35; Canada $45; International $60. Single copies/back issues: U.S. $10, all others $12. Send orders with payment to: MSDN Magazine, P.O. Box 3167, Carol Stream, IL 60132, email MSDNmag@1105service.com or call (847) 763-9560. **POSTMASTER:** Send address changes to *MSDN Magazine*, P.O. Box 1081, Skokie, IL 60076-8081. Canada Publications Mail Agreement No: 40612608. Return Undeliverable Canadian Addresses to Circulation Dept. or Bleuchip International, P.O. Box 25542, London, ON N6C 6B2.

Corporate Address: 1105 Media, Inc., 9201 Oakdale Ave., Ste 101, Chatsworth, CA 91311, www.1105media.com

Media Kits: Direct your Media Kit requests to Matt Morollo, VP Publishing, 508-532-1418 (phone), 508-875-6622 (fax), mmorollo@1105media.com

Reprints: For single article reprints (in minimum quantities of 250-500), e-prints, plaques and posters contact: PARS International, Phone: 212-221-9595, E-mail: 1105reprints@parsintl.com, www.magreprints.com/QuickQuote.asp

List Rental: This publication's subscriber list, as well as other lists from 1105 Media, Inc., is available for rental. For more information, please contact our list manager, Merit Direct. Phone: 914-368-1000; E-mail: 1105media@meritdirect.com; Web: www.meritdirect.com/1105

BPA WORLDWIDE

Printed in the USA

Your best source for software development tools!

programmer's paradise®

LEADTOOLS Recognition SDK v16.5

by LEAD Technologies

Develop robust 32/64 bit document imaging and recognition functionality into your applications with accurate and high-speed multi-threaded OCR, OMR, and 1D/2D barcode engines.

- Supports text, OMR, image, and barcode fields
- Auto-registration and clean-up to improve recognition results
- Provided as both high and low level interface
- Includes comprehensive confidence reports to assess performance

Paradise #
L05 26301A01
$3,214.^99

programmers.com/LEAD

dtSearch Engine for Win & .NET

Add dtSearch's "blazing speeds" (CRN Test Center) searching and file format support

New 64-bit Version!

- dozens of full-text and fielded data search options
- file parsers/converters for hit-highlighted display of all popular file types
- Spider supports dynamic and static web data; highlights hits with links, images, etc. intact
- API supports .NET, C++, Java, SQL and more; new .NET Spider API

"Bottom line: dtSearch manages a terabyte of text in a single index and returns results in less than a second." —InfoWorld

3 Server Pack
Paradise #
D29 02101A08
$2,375.^99

programmers.com/dtsearch

VMware vSphere

Put time back into your day.

Your business depends on how you spend your time. You need to manage IT costs without losing time or performance. With proven cost-effective virtualization solutions from VMware, you can:

- Increase the productivity of your existing staff three times over
- Control downtime—whether planned or not
- Save more than 50% on the cost of managing, powering and cooling servers

Make your time (and money) count for more with virtualization from VMware.

VMware Advanced Acceleration Kit for 6 processors
Paradise #
V55 78101A01
$9,234.^99

programmers.com/vSphere

Pragma Fortress SSH —SSH Server for Windows

by Pragma Systems

Contains Windows SSH & SFTP Servers. Certified for Windows Server 2008. Works with PowerShell.

- Full-featured server with centralized & graphical management
- GSSAPI Kerberos & NTLM authentication
- Fastest SFTP & SCP file transfer
- Supports over 1000 sessions
- Runs console applications & allows history scroll back within the same session
- Runs in Windows 2008/2003/Vista/XP/2000

New Gen2 Release!

Paradise #
P35 04100A01
$698.^99

programmers.com/pragma

Enterprise Architect 7.5

Visualize, Document and Control Your Software Project

by Sparx Systems

Enterprise Architect is a comprehensive, integrated UML 2.1 modeling suite providing key benefits at each stage of system development. Enterprise Architect 7.5 supports UML, SysML, BPMN and other open standards to analyze, design, test and construct reliable, well understood systems. Additional plug-ins are also available for Zachman Framework, MODAF, DoDAF and TOGAF, and to integrate with Eclipse and Visual Studio 2005/2008.

Corporate Edition
1-4 Users
Paradise #
SP6 03101A02
$182.^99

programmers.com/sparxsystems

TX Text Control 15

Word Processing Components

NEW RELEASE!

TX Text Control is royalty-free, robust and powerful word processing software in reusable component form.

- .NET WinForms control for VB.NET and C#
- ActiveX for VB6, Delphi, VBScript/HTML, ASP
- File formats DOCX, DOC, RTF, HTML, XML, TXT
- PDF and PDF/A export, PDF text import
- Tables, headers & footers, text frames, bullets, structured numbered lists, multiple undo/redo, sections, merge fields, columns
- Ready-to-use toolbars and dialog boxes

Professional Edition
Paradise #
T79 02101A02
$848.^99

Download a demo today.

programmers.com/theimagingsource

FarPoint Spread for Windows Forms

The Best Grid is a Spreadsheet. Give your users the look, feel, and power of Microsoft® Excel®, without needing Excel installed on their machines. Join the professional developers around the world who consistently turn to FarPoint Spread to add powerful, extendable spreadsheet solutions to their COM, ASP.NET, .NET, BizTalk Server and SharePoint Server applications.

- World's #1 selling development spreadsheet
- Read/Write native Microsoft Excel Files
- Cross-sheet formula referencing
- Fully extensible models
- Royalty-free, run-time free

Paradise #
F02 01101A01
$936.^99

programmers.com/farpoint

Unify SQLBase Embedded Database

by Unify

SQLBase is an easy to deploy database for building desktop and Web applications in server environments without a DBA.

- Small footprint
- Zero admin deployment
- GUI admin
- Embed it and forget it

For applications with encrypted security, we offer SQLBase Treasury.

Reduce your database costs and simplify maintenance with Unify SQLBase.

NEW RELEASE!

Paradise #
C15 03101A01
$143.^99

programmers.com/unify

CA ERwin r7.3

CA ERwin® Data Modeler r7.3 – Product Plus 1 Year Enterprise Maintenance

CA ERwin Data Modeler is a data modeling solution that enables you to create and maintain databases, data warehouses and enterprise data resource models. These models help you visualize data structures so that you can effectively organize, manage and moderate data complexities, database technologies and the deployment environment.

Paradise #
P26 04201E01
$3,951.^99

programmers.com/ca

FREE WEBINAR SERIES: MORE Maximum Data Modeling with CA ERwin 7.3

In our last webinar series, we looked at CA ERwin's core functionality. In this second series, we'll provide a grounding in how CA ERwin r7.3's new features help you with Master Data Management, Metadata Management, Data Warehousing, Data Governance and Business Intelligence.

There will be six sessions in the series:

- What's New in CA ERwin 7.3
- MDM (Master Data Management) with CA ERwin and Data Profiling tool
- Collaborative model management with CA ERwin ModelManager
- Validate the integrity of your model with CA ERwin Validator
- Reporting: Crystal Reports, PDF, HTML
- SAPHIR Option: light at the end of the metadata tunnel

REGISTER TODAY: programmers.com/MDM_2009

AdminStudio & Application Virtualization Pack

by Flexera Software

One Application Software Deployment Tool for Reliable MSI Packaging, Application Virtualization, and Windows 7 Migration. Top choice of Microsoft®, Novell®, LANDesk® and other software management solutions. Cut MSI packaging time by up to 70%, Deploy software to desktops with 99% success or better. AdminStudio is the only MSI packaging solution to support multiple virtualization formats, including Microsoft® App-V™, VMware® ThinApp™ and Citrix® XenApp™.

Professional Upgrade from any Active AS Pro + Silver Mtn
Paradise #
I21 09201S06
$4,228.^99

programmers.com/flexera

Orion Network Performance Monitor

by Solarwinds

Orion Network Performance Monitor is a comprehensive fault and network performance management platform that scales with the rapid growth of your network and expands with your network management needs. It offers out-of-the-box network-centric views that are designed to deliver the critical information network engineers need. Orion NPM is the easiest product of its kind to use and maintain, meaning you will spend more time actually managing networks, not supporting Orion NPM.

Paradise #
S4A 08201E02
$4,606.^99

programmers.com/solarwinds

866-719-1528 **programmersparadise.com**

Prices subject to change. Not responsible for typographical errors.

Shaping a New Era in *MSDN Magazine*

I'm Diego Dagum, the new editorial director for *MSDN Magazine*. Because I'm new to the magazine, I'll tell you a bit about myself before getting into the details of this month's issue.

Developing software has always been a passion for me, since my teenage years when a new range of so-called *home computers* replaced the first wave of console games (these last mainly dominated by Atari). Home computers were also console games, but started coming with programming capabilities. There would typically be some BASIC dialect built-in, plus the possibility of getting alternative languages via cartridges like LOGO or Assembler.

That led to university training in computer science, with the goal of becoming a professional developer. Once graduated, I worked as a developer for different types of industries (from manufacturers to communications, from startups to large corporations and so on), having to learn not just the ways of the various platforms being used, but the tricks of the trade for each business. That helped me better match up the capabilities of technology with the need of businesses to "do more with less."

When you're able to analyze technologies not for what they are but for how much they help leverage business, you become an architect. That's what I've been doing as editor of *The Architecture Journal* (a sister magazine) for a year and a half now; I'll keep doing that job as well.

Back on the *MSDN Magazine* side, I'm working with our new editor in chief, Keith Ward, to make some changes to the magazine. A key goal of that is to better align our content with your needs as a developer. To that end, I've written a blog entry discussing that new focus at http://blogs.msdn.com/msdnmagazine/archive/2009/10/07/9904758.aspx.

To summarize here, there are five core areas we want to stress going forward.

1. Architectural background for enterprise development
2. More infrastructure awareness for a better understanding on what the platform already offers for security, health, scalability and so on
3. Best practices to adopt, followed by worst practices to avoid
4. Team consciousness, because the developer is just one in a series of stakeholders who play a role in the development process
5. A stronger focus on the platforms and tools developers currently use, and less emphasis on the next new thing

We are also tweaking our content, which will have an effect on features and our column lineup. These changes will be incorporated incrementally, and we'll be actively seeking your input, to help us shape the magazine to best serve your needs as active developers.

Regarding this issue, I want to highlight certain articles we are featuring:

- Nikhil Sachdeva explains how Pex, a tool developed by Microsoft Research, can help you keep your legacy code from rotting by automatically producing small test suites with high code and assertion coverage.
- Brian A. Randell and Marcel de Vries show us how to extend Visual Studio Team Explorer feature to add more level of interactions with Team Foundation Server.
- Ayende Rahien explores best practices for desktop applications that use NHibernate for object/relational mapping.
- Finally, David Laribee gives a particular look on the economics of software development, focusing on a way to turn down a high-costly maintenance codebase in favor of a more productive one.

I hope you enjoy this issue. Keith and I, together with our production team, will keep working on ways to improve *MSDN Magazine*. Tell us how we can make it even better by sending your comments to mmeditor@microsoft.com.

[illegible]

Visit us at msdn.microsoft.com/magazine. Questions, comments or suggestions for *MSDN Magazine*? Send them to the editor: mmeditor@microsoft.com.

"Best Defect Tracker & Project Management Software"
-asp.netPRO Magazine

FREE! 1-User License

4 years in a Row!

2006 2007 2008 2009 READERS' CHOICE asp.net PRO WINNER

OnTime 2009

- Defect Tracking
- Requirement Mgmt
- Help Desk
- Team Wiki
- Scrum/Agile

Used by Over 7,000 Dev Teams Worldwide

Track Everything
Track everything from bugs to software requirements, team member tasks, and more. 360-degree visibility.

Team Wiki
Finally, one place to collaborate on projects in a clean, organized, hierarchical structure.

Help Desk
Use OnTime for tracking Help Desk incidents - even monitor email accounts to generate help tickets.

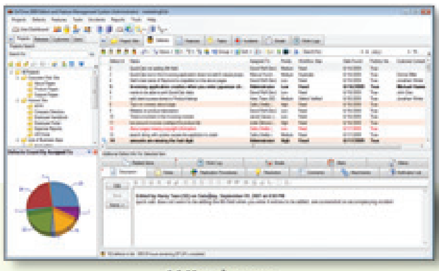
Scrum
Harness the power of scrum within the OnTime client. Use Sprints, Burndowns, Trending Charts and more.

Workflow
Define a hyper-fast, lightweight, agile workflow or a robust process-enforcement system.

Notifications
Ensure team members are in synch with automatic email notifications and alerts.

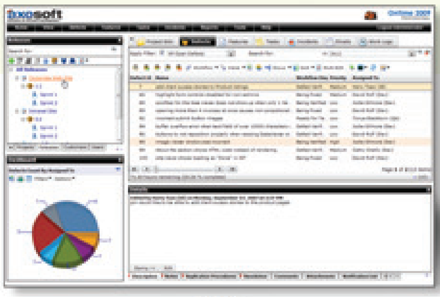
Customize
Unlimited custom fields, customize all field names, and decide when fields are visible, editable or required.

Security
Secure every aspect of the system by project, user roles, and even fields.


Windows

800.653.0024
Our U.S. and International sales teams look forward to answering your questions or setting up a live demo.
8:00 a.m. - 5:00 p.m. MST

Starts at: FREE
Single-user licenses of OnTime are always free. 5-user teams start at just $395. Visit axosoft.com for a specific quote.


Web

axosoft
Software for Software Development™

Visit **www.axosoft.com** for a video tour, live demos, downloads, and much more.

MOST WANTED

REWARD

30-day Free Trial

www.altova.com/wanted

Database and OR/M Tools, Oren Eini and Custom Visual Studio Tabs

Enhancing the LINQ to SQLand ADO.NET Entity Framework Designers

LINQ to SQL and the ADO.NET Entity Framework are two object-relational mapping (OR/M) implementations from Microsoft. Using these OR/Ms entails creating a .dbml or .edmx file that contains a mapping between the relational model and the object model. Under the covers, this mapping is defined via XML, but the Visual Studio designer makes creating such mappings as easy as drag and drop.

If you use LINQ to SQL or Entity Framework in your applications and are in charge of maintaining the mapping files, check out **Huagati DBML/EDMX Tools** (version 1.76), which adds a number of features to the Visual Studio LINQ to SQL and Entity Framework designers.

Observing prescribed naming conventions in the object model is a common challenge with drag and drop OR/M tools. Database object names often include prefixes like "tbl" or "usp_", or may be entirely capitalized or lowercase. When adding a database object to the designer, Visual Studio creates the corresponding class using the same prefixes and casing, which may run counter to your team's naming conventions. With Huagati DBML/EDMX Tools, you're just a couple of clicks away from renaming all of the classes and members in the object model. You can add or remove prefixes or suffixes, force proper casing, remove underscores and more.

One shortcoming of the Visual Studio LINQ to SQL designer is that there is no mechanism to update the object model to reflect the latest database changes. For example, when you first drag a database table onto the designer, an object is created with properties that correspond to the table's columns. If three new columns are added to the same table a few weeks later, you must return to the LINQ to SQL designer and either remove and then re-add the table onto the designer, or manually add three new properties to the corresponding class. This shortcoming is a non-issue with the Huagati DBML/EDMX Tools, which can report the differences between the relational model and the object model and automatically sync the object model with the relational model. The Huagati DBML/EDMX Tools can also generate the SQL statements needed to modify the relational model so that it mirrors the object model.

With Huagati DBML/EDMX Tools you're just a couple of clicks away from renaming all of the classes and members in the object model.

Another handy feature is the documentation tool, which retrieves the table and column descriptions from the database and includes them in the XML documentation of the object model.

The Huagati DBML/EDMX Tools includes a free 45-day trial version, along with Standard and Professional versions. The Standard version costs $50 per user license and supports up to 80 tables per model. The Professional version costs $120 per user license and imposes no limits.

Price: $50 to $120 per user license
huagati.com/dbmltools

Blogs of Note

One of my favorite .NET bloggers is Oren Eini, a prolific blogger and respected software developer who posts under the pseudonym Ayende Rahien. Eini is perhaps best known for his contributions to .NET open source projects. He's a contributor to NHibernate, a popular OR/M framework reviewed in the October 2006 issue of *MSDN Magazine* (msdn.microsoft.com/magazine/cc163540), and is the creator of Rhino Mocks, a .NET mocking framework covered in the January 2008 issue (msdn.microsoft.com/magazine/cc135981).

Eini's blog posts explore a spectrum of topics. Some entries examine a feature or pitfall of a particular framework, such as NHibernate or the Microsoft Entity Framework. Most are more general, imparting advice from the trenches regarding database design, OR/M usage, testing and software architecture and design. Posts often include screenshots, diagrams and code snippets.

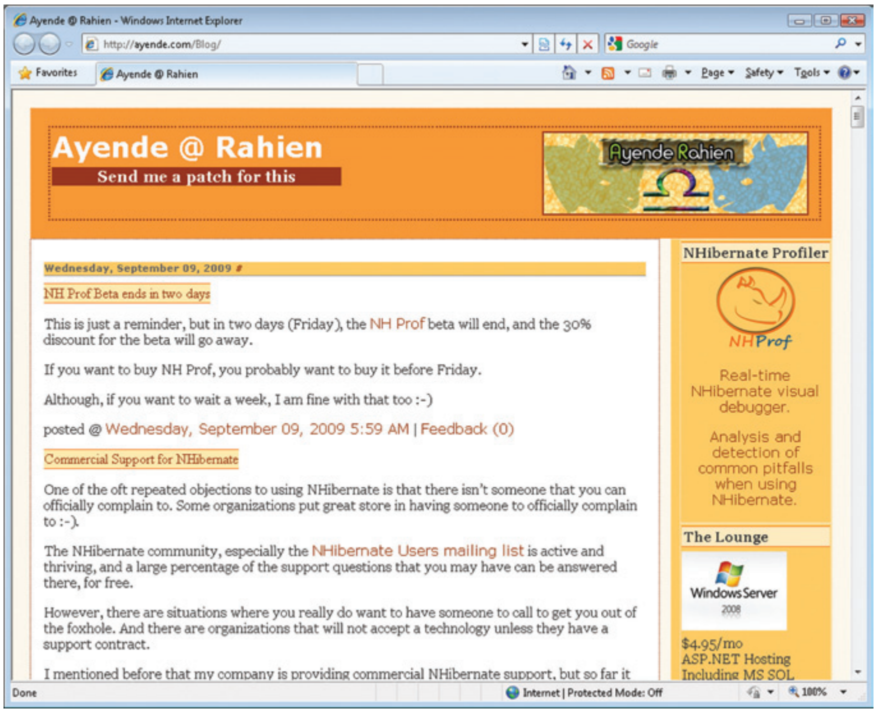
All prices confirmed at press time are subject to change. The opinions expressed in this column are solely those of the author and do not necessarily reflect the opinions at Microsoft.

Send your questions and comments for Scott to toolsmm@microsoft.com.

You'll find, for example, a series of posts on OR/M implementation challenges such as mapping between the relational and object models, hydrating entities and so on. And be sure to read "Solving the Select N+1 Problem" (ayende.com/Blog/archive/2008/12/01/solving-the-select-n1-problem.aspx), which explains how iterating over parent-child data can unwittingly lead to serious performance issues.

Eini's blog also contains types of posts you don't typically find in developer-focused blogs. For instance, there are a number of entries that are only a few sentences long, yet still manage to convey an important idea and get the reader thinking. Other posts contain a lengthy code snippet with a single statement like, "Find the bug." With these types of posts, along with his impressive output, it's not uncommon for there to be three or more posts *per* day. Oren's blog is a must-read for .NET developers and architects.

ayende.com/Blog


Ayende @ Rahien - Windows Internet Explorer
http://ayende.com/Blog/
Favorites
Ayende @ Rahien
Ayende @ Rahien
Send me a patch for this
Ayende Rahien
NHibernate Profiler
NHProf
Real-time NHibernate visual debugger.
Analysis and detection of common pitfalls when using NHibernate.
The Lounge
Windows Server 2008
$4.95/mo ASP.NET Hosting Including MS SQL
Done
Internet | Protected Mode: Off
100%

Wednesday, September 09, 2009 #

NH Prof Beta ends in two days

This is just a reminder, but in two days (Friday), the NH Prof beta will end, and the 30% discount for the beta will go away.

If you want to buy NH Prof, you probably want to buy it before Friday.

Although, if you want to wait a week, I am fine with that too :-)

posted @ Wednesday, September 09, 2009 5:59 AM | Feedback (0)

Commercial Support for NHibernate

One of the oft repeated objections to using NHibernate is that there isn't someone that you can officially complain to. Some organizations put great store in having someone to officially complain to :-).

The NHibernate community, especially the NHibernate Users mailing list is active and thriving, and a large percentage of the support questions that you may have can be answered there, for free.

However, there are situations where you really do want to have someone to call to get you out of the foxhole. And there are organizations that will not accept a technology unless they have a support contract.

I mentioned before that my company is providing commercial NHibernate support, but so far it

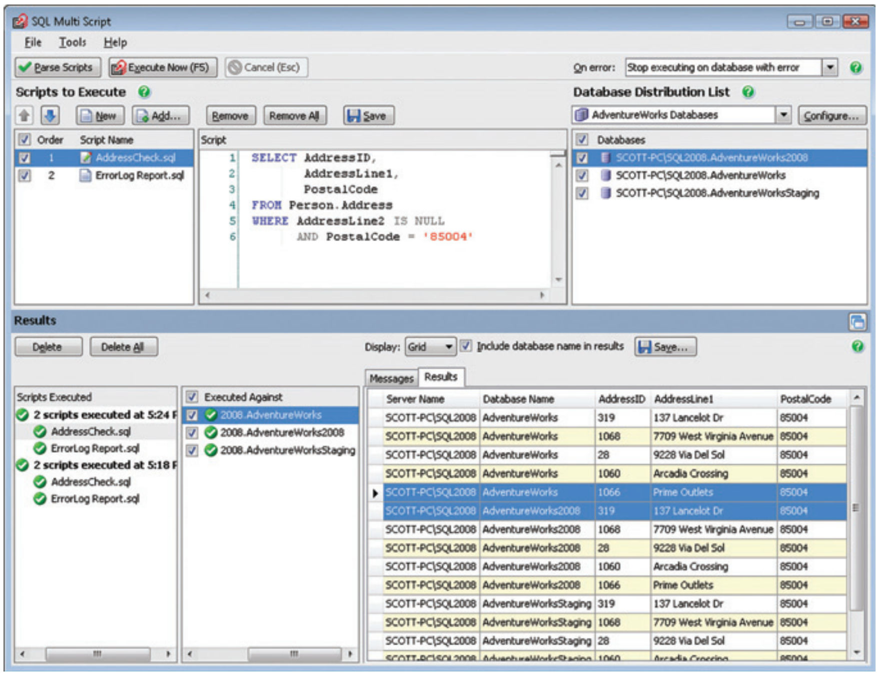
Oren Eini's blog

Run Simultaneous Queries Against Multiple Databases

Over the course of my career, I've helped build a number of multi-tenant applications—applications that have a single instance running on a hosted web server but are used by multiple organizations. A multi-tenant application must ensure that a user can view and manage only the data that belongs to her organization. To this end, multi-tenant applications that work with sensitive information often store each organization's data in a separate database to fully isolate the data (among other reasons).

Viewing data aggregated across multiple organizations can be a real challenge when each organization's data is stored in a separate database. Imagine that a developer finds a bug that has corrupted data for a particular organization. To determine if there's similar corruption for the other customers, the developer must run a query on every single database, emitting a separate resultset for each. As you can imagine, that gets tedious. Ideally, the results from each database would be aggregated into a single, unified output, which could then be sorted, filtered and so forth.

Over the years, I've queried multiple databases using a variety of techniques, including the undocumented *sp_MsForEachDb* stored procedure, writing batch files and building custom tools. Recently, a colleague introduced me to Red Gate Software's **SQL Multi Script** (version 1.1), and I haven't looked back. SQL Multi Script works much like you'd expect—enter the SQL statements to execute and select the databases to query against. SQL Multi Script then fires off the SQL statements to the specified databases and aggregates and displays the results.



Order	Script Name
1	AddressCheck.sql
2	ErrorLog Report.sql

```
SELECT AddressID,
       AddressLine1,
       PostalCode
FROM Person.Address
WHERE AddressLine2 IS NULL
     AND PostalCode = '85004'
```

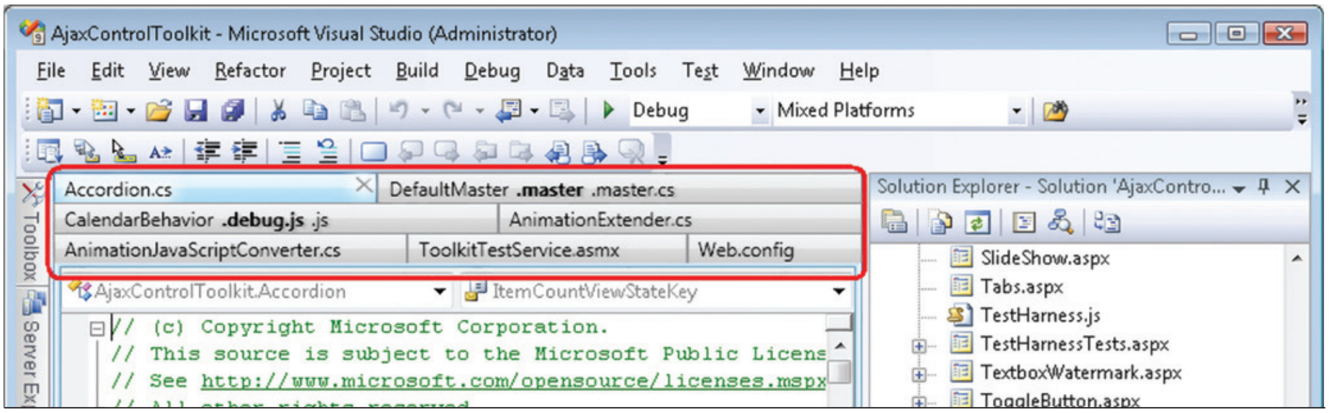
Databases
- SCOTT-PC\SQL2008.AdventureWorks2008
- SCOTT-PC\SQL2008.AdventureWorks
- SCOTT-PC\SQL2008.AdventureWorksStaging

Scripts Executed
- 2 scripts executed at 5:24 P
  - AddressCheck.sql
  - ErrorLog Report.sql
- 2 scripts executed at 5:18 P
  - AddressCheck.sql
  - ErrorLog Report.sql

Executed Against
- 2008.AdventureWorks
- 2008.AdventureWorks2008
- 2008.AdventureWorksStaging

Server Name	Database Name	AddressID	AddressLine1	PostalCode
SCOTT-PC\SQL2008	AdventureWorks	319	137 Lancelot Dr	85004
SCOTT-PC\SQL2008	AdventureWorks	1068	7709 West Virginia Avenue	85004
SCOTT-PC\SQL2008	AdventureWorks	28	9228 Via Del Sol	85004
SCOTT-PC\SQL2008	AdventureWorks	1060	Arcadia Crossing	85004
SCOTT-PC\SQL2008	AdventureWorks	1066	Prime Outlets	85004
SCOTT-PC\SQL2008	AdventureWorks2008	319	137 Lancelot Dr	85004
SCOTT-PC\SQL2008	AdventureWorks2008	1068	7709 West Virginia Avenue	85004
SCOTT-PC\SQL2008	AdventureWorks2008	28	9228 Via Del Sol	85004
SCOTT-PC\SQL2008	AdventureWorks2008	1060	Arcadia Crossing	85004
SCOTT-PC\SQL2008	AdventureWorks2008	1066	Prime Outlets	85004
SCOTT-PC\SQL2008	AdventureWorksStaging	319	137 Lancelot Dr	85004
SCOTT-PC\SQL2008	AdventureWorksStaging	1068	7709 West Virginia Avenue	85004
SCOTT-PC\SQL2008	AdventureWorksStaging	28	9228 Via Del Sol	85004
SCOTT-PC\SQL2008	AdventureWorksStaging	1060	Arcadia Crossing	85004

SQL Multi Script


AjaxControlToolkit - Microsoft Visual Studio (Administrator)
File Edit View Refactor Project Build Debug Data Tools Test Window Help
Debug
Mixed Platforms
Accordion.cs
DefaultMaster .master .master.cs
CalendarBehavior .debug.js .js
AnimationExtender.cs
AnimationJavaScriptConverter.cs
ToolkitTestService.asmx
Web.config
AjaxControlToolkit.Accordion
ItemCountViewStateKey
Solution Explorer - Solution 'AjaxContro...
SlideShow.aspx
Tabs.aspx
TestHarness.js
TestHarnessTests.aspx
TextboxWatermark.aspx
ToggleButton.aspx
Toolbox
Server Ex

```
// (c) Copyright Microsoft Corporation.
// This source is subject to the Microsoft Public Licens
// See http://www.microsoft.com/opensource/licenses.mspx
```

Tabs Studio

By default, SQL Multi Script sends the SQL statements to the databases in parallel. This can greatly reduce the time it takes to get back the results, especially when the databases reside on different servers. Alternatively, you can instruct SQL Multi Script to issue the statements serially, which is useful if you want to stop executing the script in the face of an error.

If you do get an error, SQL Multi Script offers four error handling options: continue executing the script (the default behavior); stop executing the current statement on the database, but continue with the other SQL statements; stop executing all statements on this database and move on to the next database in the list; or stop executing all statements on all databases.

SQL Multi Script's Results pane provides an aggregated view of the messages and data returned by the databases, along with a history of the scripts executed against the databases during the current session. The Results pane also includes a checkbox list of the databases that were queried; check or uncheck a database to show or remove its results from the aggregate. You can also click the Save button to save the aggregated results to a .CSV or .TXT file.

Price: $195

red-gate.com

Improve the Visual Studio Tabs

When working on a large project, it is not uncommon to have dozens of files open within Visual Studio. Unfortunately, the Visual Studio user interface leaves a bit to be desired when there are many open documents. By default, Visual Studio uses a Single Document Interface (SDI) with a series of tabs that show which documents are open. However, these tabs are laid out horizontally, which limits how many can be displayed on the screen. The Visual Studio Multi Document Interface (MDI) does not show tabs, but instead requires the user to go to the Window menu to view and switch among the open documents.

Unlike Visual Studio, Tabs Studio displays the tab for every open document in both the SDI and MDI configurations.

Tabs Studio (version 1.6), by Sergey Vlasov, is an add-in that replaces the built-in Visual Studio tabs with an improved and customizable set of tabs. Unlike Visual Studio, Tabs Studio displays the tab for every open document in both the SDI and MDI configurations. If there is not enough horizontal space to display every tab, Tabs Studio stacks them vertically.

Many types of components created in Visual Studio are implemented using multiple files. For example, creating an ASP.NET page named Default.aspx actually creates two files, Default.aspx and Default.aspx.cs (or Default.aspx.vb). Tabs Studio adds features that make it easier to work with such files.

Say you're working with Default.aspx and need to open Default.aspx.cs. Right-click on the Default.aspx tab and the context menu includes an Open Default.aspx.cs option. What's more, Tabs Studio groups related documents into a single tab. When both Default.aspx and Default.aspx.cs are opened, Tabs Studio will display a single tab that lists the file name without the extension (Default) along with the two extensions (.aspx and .aspx.cs). Click the .aspx extension in the tab to bring up Default.aspx, or click .aspx.cs to display Default.aspx.cs.

Tabs Studio also has a variety of configuration options. For instance, Tabs Studio allows the tabs' styles to be customized via XAML. You can configure the tabs' fonts, colors, shapes and more. And because the settings are defined using XAML, you can include conditional statements and other programmatic logic, making it possible to do things like specify the styling for the currently selected tab, or make the previously selected tab a different color from the other non-selected tabs. Tabs Studio can be further customized using add-ins. You can write your own or download any of the free add-ins available from the Tabs Studio Web site.

Tabs Studio is available for the non-Express Editions of Visual Studio 2005, 2008 and 2010.

Price: $34

tabsstudio.com

SCOTT MITCHELL, *author of numerous books and founder of 4GuysFromRolla.com, is an MVP who has been working with Microsoft Web technologies since 1998. Mitchell is an independent consultant, trainer and writer. Reach him at Mitchell@4guysfromrolla.com or via his blog at ScottOnWriting.NET.*

JESSE KAPLAN AND LUIZ FERNANDO SANTOS

In-Process Side-by-Side

As we built the .NET Framework 4, one of the more difficult problems we faced was maintaining compatibility with previous releases while still adding new features and functionality. We followed strict processes requiring approval for any changes that might introduce compatibility issues—most were rejected—and ran a compatibility lab with hundreds of real applications to find any unintentional breaks.

But every time we fix a bug there's the risk that some application depended on that wrong behavior. Sometimes applications take dependencies we have warned against, such as the behavior of private APIs or the description text of exceptions.

Since the .NET Framework was introduced, we've had a good solution for application compatibility: allow multiple versions of the .NET Framework to be installed on the same machine at the same time. This enables two different applications, built against two different versions and installed on one machine, to each run against the appropriate version.

Add-In Problems

This works fine when each application gets its own process, but add-ins are much more difficult problems. Imagine you are running a program such as Outlook that hosts COM add-ins, including managed COM add-ins, and you have two versions of the runtime—and add-ins built against each one—installed on your machine. Which runtime should you choose? Loading a newer add-in on an older runtime clearly is not going to work.

On the other hand, because of the high level of compatibility, an older add-in will usually run fine on a newer runtime. To give all add-ins the best chance of working, we always choose the latest runtime for managed COM activation. Even if you only have older add-ins installed, there is no way for us to know that when that add-in gets activated, so the latest runtime still gets loaded.

An unfortunate side effect of this activation policy is that when a user installs a new application with a new version of the runtime, completely unrelated applications that use managed COM add-ins, built against older versions, suddenly start running on a newer runtime and can fail.

For the .NET Framework 3.0 and 3.5, we solved this problem through an extremely strict policy: each release was additive and only added new assemblies to the prior version with the same runtime underneath. This prevented any compatibility issues when installing them on a machine running the .NET Framework 2.0. This means that when you are running an app on the .NET Framework 3.5, you are really running it on the 2.0 runtime, with a few extra assemblies on top of it. However, it also means that we couldn't innovate in the .NET 2.0 assemblies, which include key functionalities, such as the garbage collector, just in time (JIT) and base class libraries.

With the .NET Framework 4 we have implemented an approach that allows high compatibility, including never breaking existing add-ins, and also lets us innovate in the core. We can now run both .NET 2.0 and .NET 4 add-ins in the same process, at the same time. We call this approach In-Process Side-by-Side, or In-Proc SxS.

While In-Proc SxS solves the most common compatibility issues, it doesn't fix everything. In this column we'll describe more about why we decided to build In-Proc SxS, how it works and which problems it doesn't solve. For people writing normal applications or add-ins, In-Proc SxS mostly just works—the right things all happen automatically. For those of you who are writing hosts that can take advantage of In-Proc SxS, we'll also describe the updated hosting APIs and provide some guidelines for using them.

The Trip to Ray Ozzie's Office

Late in 2005 almost all high-level Microsoft executives were suddenly unable to check e-mail on any of their main machines. For no apparent reason, whenever they opened Outlook it would crash, restart and then crash again in a continuous loop. There were no recent updates to Outlook or anything else that seemingly could have been causing this. It was soon tracked down to a managed exception being thrown by a managed add-in. A friend of mine [*"Mine" refers to column co-author Jesse Kaplan—Ed.*] from the Visual Studio Tools for Office (VSTO) team—responsible for managed add-ins to Office—was sent to diagnose this problem on the machine of one of the most prominent victims of this bug: Ray Ozzie, who was chief technical officer at the time.

Once in Ray's office, my friend was quickly able to determine that a beta version of the .NET Framework 2.0 had been deployed via an internal beta program, and he identified which Office add-in was causing the problem. As one of the compatibility PMs on the CLR team, I installed the add-in and took it from there.

We quickly determined what went wrong: the add-in had a race condition in which it started up nine different threads, and after

Send your questions and comments for Kaplan and Santos to clrinout@microsoft.com.

DEVELOPER ROCK STARS

Turn up productivity and kick out reliable apps.

ENGINEER:

Vasanth Tovinkere

LATEST APP:

Intel® Parallel Amplifier

DEVELOPER TIP:

Pump up your performance.

Excessive locking can severely limit performance. Check the health of your application by using the Locks & Waits analysis mode in Intel Parallel Amplifier to see the effect locks have on the concurrency of your application.

Rock Your Code. You, too, can be a developer rock star with Intel® Parallel Studio.

> Create reliable, error-free apps from desktop to device.
> Optimize your code and boost productivity.
> Pull more performance from your multicore processor.

Get threaded. Turn it up to Intel® Parallel Studio.

Visit **www.intel.com/software/products/eval** for a free evaluation.

starting each one it initialized the data the thread processed (see **Figure 1**). The coders got lucky with the timing, but once the .NET Framework 2.0 was installed, the add-in was automatically rolled forward to .NET 2.0, for the reasons I outlined above. But .NET 2.0 was slightly faster at starting threads, so the latent race condition started to surface consistently.

This application failure drove home a hard lesson in compatibility: no matter how hard we try to avoid making behavior changes than can break applications, simple things such as a performance improvement can expose bugs in applications and add-ins that can cause them to fail when run against anything other than the runtime they were built and tested on. We realized that there was no way for us to evolve the platform in any meaningful way and ensure that applications like the one above can run perfectly on the latest version.

The Broken Installation

During our compatibility testing we came upon an application that ran fine if .NET 2.0 was installed after the application but failed if the application was installed on a machine that had both 1.1 (the version the application was built against) and 2.0. It took a while to figure out what was going on, but we tracked the problem down to a bit of code that was running inside the installer that was, again, floating forward to 2.0 and this time having problems finding the framework directory.

The detection logic was clearly fragile and actually wrong, as you can see here:

```
string sFrameworkVersion = System.Environment.Version.ToString();
string sWinFrameworkPath = session.Property["WindowsFolder"] +
"Microsoft.NET\\Framework\\v" +
sFrameworkVersion.Substring(0,8) + "\\";
```

But even after fixing that bug, the application still failed to execute properly after installing. Here's the fix:

```
string sWinFrameworkPath = System.Runtime.InteropServices.
RuntimeEnvironment.GetRuntimeDirectory();
```

It turns out that the installer was looking for the framework directory in order to get a path to caspol.exe and give the app permission to run in that framework. It broke even after finding the path because it had just granted itself permission to run on the 2.0 CLR even though the application itself runs on the 1.1 CLR. Here's the problem code:

```
System.Diagnostics.Process.Start(sWinFrameworkPath +
"caspol.exe  " + casPolArgs);
```

Compatibility Through In-Process Side-by-Side

The core issue causing problems in all of these cases, as we came to understand, is that it was impossible to make any significant changes or additions to our platform and still ensure that the latest version could run any application as well as older versions did.

From the beginning, the .NET Framework tried to solve this problem by supporting side-by-side installations of multiple versions of the framework on a machine and having each application choose which version it wanted to run on.

Unfortunately the limitation of one runtime per process meant that for managed COM components and extensibility scenarios, where there were multiple independent apps running in the same process, there was no single choice that would work for every one. This limitation meant that some components were not going to get the runtime they wanted and that, regardless of how hard we tried to maintain compatibility, some percentage of them would break.

Our new ability to load multiple versions of the runtime in a process solves these problems.

Figure 1 **Code from the Office Add-In**

```
Thread [] threads = new Thread[9];
for (int i=0; i<9; i++)
{
  Worker worker = new Worker();
  threads[i] = new ThreadStart(worker.Work);
  threads[i].Start();  //This line starts the thread executing
  worker.identity =i; //This line initializes a value that
                      //the thread needs to run properly
}
```

Overarching Principles

To help you better understand some of the decisions we made and the detailed behavior we describe later in the column, it's useful to discuss the guiding principles we held to while designing this feature.

1. Installing a new version of the .NET Framework should have no impact on existing applications.
2. Applications and add-ins should run against the version of the framework they were built and tested against.
3. There are situations, such as when using libraries, where we can't run code against the framework the libraries were built against, so we must still strive for 100 percent backward compatibility.

All existing applications and add-ins should continue to run against the versions of the framework they were built and configured to run on and should not see the new version unless they specifically ask for it. This has always been the rule for managed applications, but now it also applies to managed COM add-ins and consumers of the runtime hosting APIs.

In addition to making sure applications run against the version of the runtime they were built with, we still need to make sure it is easy to transition to a newer runtime, so we have kept compatibility for the .NET Framework 4 as high as or higher than it was with .NET 2.0.

Overview of Behavior

The .NET Framework 4 runtime—and all future runtimes—will be able to run in-process with one another. While we did not back-port this functionality to older runtimes (1.0 through 3.5), we did make sure that 4 and beyond will be able to run in-process with any single older runtime. In other words, you will be able to load 4, 5 and 2.0 in the same process, but you will not be able to load 1.1 and 2.0 in the same process. .NET Frameworks 2.0 through 3.5 all run on the 2.0 runtime and so have no conflicts with one another, as shown in **Figure 2**.

No existing applications or components should notice any difference when the .NET Framework 4 runtime is installed: they should continue to get whichever runtime they were built against. Applications and managed COM components built against .NET 4 will execute on the 4 runtime.

Hosts that wish to interact with the 4 runtime will need to specifically request it.

What Does In-Process Side-by-Side Mean to You?

End Users and System Administrators: You now can have confidence that when you install a new version of the runtime, either independently or with an application, it will have no impact on your machine, and all existing applications will continue to run as they did before.

Application Developers: In-Proc SxS has almost no impact on application developers. Applications have always defaulted to run against the version of the framework on which they were built and this has not changed. The only change in behavior we have made that impacts application developers is that we will no longer automatically run an application built against an older runtime on a newer version when the original version is not present. Instead we will prompt the user to download the original version and provide a link to make it easy to do so.

We still provide configuration options that allow you to indicate which versions of the framework you want your application to run against, so it is possible to run an older application on a newer runtime, but we won't do it automatically.

Library Developers and Consumers: In-Proc SxS does not solve the compatibility problems faced by library developers. Any libraries directly loaded by an application—either via a direct reference or an Assembly.Load*—will continue to load directly into the runtime and AppDomain of the application loading it. This means that if an application is recompiled to run against the .NET Framework 4 runtime and still has dependent assemblies built against .NET 2.0, those dependents will load on the .NET 4 runtime as well. Therefore, we still recommend testing your libraries against all version of the framework you wish to support. This is one of the reasons we have continued to maintain our high level of backward compatibility.

Managed COM Component Developers: In the past, these components would automatically run against the latest version of the runtime installed on the machine. Now, pre-.NET Framework 4 components will still get activated against the latest runtime (3.5 or earlier) and all newer components will be loaded against the version they were built on, as shown in **Figure 3**.

Shell Extension Developers: Shell extension is a general name applied to a wide variety of extensibility points inside the Windows shell. Two common examples are extensions that add to the right-click context menu for files and folders and those that provide custom icons or icon overlays for files and folders.

These extensions are exposed via a standard COM model, and their defining characteristic is that they are loaded in-process with any application. It is this last bit, and the fact that only one CLR has been allowed per process, that caused problems for managed shell extensions. To elaborate:

- The shell extension was written against runtime version N.
- It needs to be loadable in any application on the machine including those built against versions later and earlier than N.
- If the application was built against a later version than the extension, things are generally OK unless there are compatibility problems.
- If the application was built against an earlier version of the extension, it is guaranteed to fail—the older runtime obviously cannot run a shell extension that was built on a newer one.
- If somehow the shell extension was loaded before the application's managed code components, its choice of framework version could conflict with the app and break everything.

These problems led us to officially recommend against—and not support—the development of in-process shell extensions using managed code. This was a painful choice for us and for our customers as you can see in this MSDN forum explaining the problem: http://social.msdn.microsoft.com/forums/en-US/

Figure 2 **Will These Runtimes Load in the Same Process?**

.NET Framework Version

	1.1	2.0 through 3.5	4	5
1.1	N/A	No	Yes	Yes
2.0 through 3.5	No	N/A	Yes	Yes
4	Yes	Yes	N/A	Yes
5	Yes	Yes	Yes	N/A

Figure 3 **Managed COM Components and Runtime Interoperability**

Managed COM Components: Which version will my component run against?

Component Version	1.1	2.0 through 3.5	4	5
Machine/Process State				
1.1, 3.5, 4, 5 installed None loaded	3.5	3.5	4	5
1.1, 3.5, 4, 5 installed 1.1, 4 loaded	1.1	Fails to load*	4	5
1.1, 3.5, 4, 5 installed 3.5, 5 loaded	3.5	3.5	4	5
1.1, 3.5, 5 installed None loaded	3.5	3.5	Fails to load by default**	5

*These components would fail to load in the past as well.

**When you register components, you can now specify a range of versions they support, so you could configure this component to run against 5 or future runtimes if you have tested them.

netfxbcl/thread/1428326d-7950-42b4-ad94-8e962124043e. Shell extensions are very popular and one of the last places where developers of certain types of applications are forced to write native code. Unfortunately, because of our limitation allowing only one runtime per process, we could not support them.

With the ability to have multiple runtimes in process with any other runtime, we can now offer general support for writing managed shell extensions—even those that run in-process with arbitrary applications on the machine. We still do not support writing shell extensions using any version earlier than .NET Framework 4 because those versions of the runtime do not load in-process with one another and will cause failures in many cases.

Developers of shell extensions, managed and native, still have to take special care and ensure that they are able to run in a wide variety of environments and work well with others. As we get closer to release to manufacturing (RTM), we will provide guidance and samples that will help you develop high-quality managed shell extensions that play well in the Windows eco-system.

Hosts of Managed Code: If you host managed code using native COM activation, you will not have to do anything special to work with multiple runtimes. You can simply activate components as you always did and the runtime will load them according to the rules listed in **Figure 3**.

If you've ever used any of our pre-.NET Framework 4 hosting APIs, you have probably noticed that they all assume that only one runtime will ever be loaded in the process. Therefore, if you host the runtime using our native APIs, you will need to modify your host to be In-Proc-SxS-aware. As part of our new approach of having multiple runtimes in a process, we have deprecated the old, single-runtime-aware hosting APIs and added a new set of hosting APIs designed to help you manage a multi-runtime environment. MSDN will have complete documentation for the new APIs, but they will be relatively easy to use if you have experience using the current ones.

One of the most interesting challenges we faced when developing In-Proc SxS was the question of how to update the behavior of the existing, single-runtime-aware hosting APIs. A range of options was available, but when following the principles laid out earlier in this column we were left with the following guideline: the APIs should behave such that when the .NET Framework 4 is installed on a machine, they return the exact behavior they did before. This means that they can only be aware of one runtime in each process and that even if you used them in a way that would have previously activated the latest runtime on the machine, they will only give you the latest runtime with a version earlier than 4.

There are still ways to "bind" these APIs to the .NET Framework 4 runtime by explicitly passing the 4 version number to them or configuring your application in a certain way, but, again, this will happen only if you specifically request the 4 runtime and not if you requested the "latest."

In summary: code using the existing hosting APIs will continue to work when .NET Framework 4 is installed but will get a view of the process that sees only one runtime loaded. Also, to maintain this compatibility, they will usually be able to interact only with pre-4 versions. The details of which version is chosen for each of these older APIs will be available on MSDN, but the few rules above should help you understand how we determined these behaviors. If you want to interact with multiple runtimes, you will need to move to the new APIs.

C++/CLI Developers: C++/CLI, or managed C++, is an interesting technology that allows developers to mix both managed and native code in the same assembly and manage the transitions between the two largely without developer interaction.

Because of that architecture, there will be limits on how you use these assemblies in this new world. One of the fundamental problems is that if we allowed these assemblies to be loaded multiple times per process, we still would need to maintain isolation between both the managed and native data sections. That means loading both sections twice, which is not allowed by the native Windows loader. The full details of why we have the following restrictions are outside the scope of this column but will be available elsewhere as we get closer to RTM.

The basic restriction is that pre-.NET Framework 2.0-based C++/CLI assemblies can only load on the .NET 2.0 runtime. If you provide a 2.0 C++/CLI library and want it to be consumable from 4 and beyond, you need to recompile it with each version you want it to be loaded in. If you consume one of these libraries, you will either need to get an updated version from the library developer or, as a last resort, you can configure your application to block pre-4 runtimes from your process.

No More Hassle

The Microsoft .NET Framework 4 is the most backward-compatible release of .NET yet. By bringing In-Proc SxS to the table, Microsoft guarantees that the simple action of installing .NET 4 will not break any existing application and that everything already installed on the machine will work as well as it did before.

End users will no longer have to worry that installing the framework—either directly or with an application that requires it—will break any of the applications already on the machine.

Enterprises and IT professionals can adopt new versions of the framework as quickly or as gradually as they wish without having to worry about different versions used by different applications conflicting with one another.

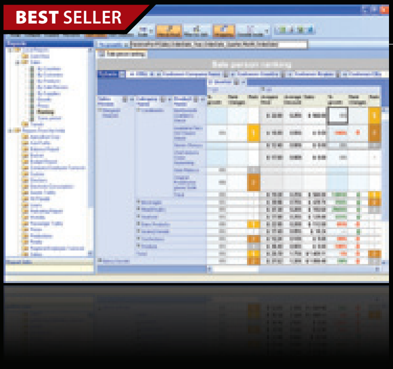
Developers can use the latest version of the framework to build their applications and will be able to reassure their customers that they will be safe to deploy.

Finally, hosts and add-in developers can be comfortable knowing that they will get the version of the framework they want, without impacting anyone else, so they can be confident their code will keep working even as new versions of the framework are installed. ■

JESSE KAPLAN *is the program manager of Managed/Native Interoperability for the CLR team at Microsoft. His past responsibilities include compatibility and extensibility.* **LUIZ SANTOS**, *formerly part of the CLR team, is a program manager in the SQL Connectivity team, where he is responsible for the ADO.NET Managed providers, including SqlClient, ODBCClient and OLEDBClient.*

THANKS to the following technical experts for reviewing this article: *Joshua Goodman, Simon Hall and Sean Selitrennikoff*

ComponentSource®
The Definitive Source of Software Components
www.componentsource.com



ContourCube | from $900.00

Contour Components

OLAP component for interactive reporting and data analysis.

- Embed Business Intelligence functionality into database applications
- Zero report coding - design reports with drag and drop
- Self-service interactive reporting - get hundreds of reports by managing rows/columns
- Royalty free - only development licenses are needed
- Provides extremely fast processing of large data volumes

TX Text Control .NET and .NET Server | from $499.59

TX TEXT CONTROL word processing components

Word processing components for Visual Studio .NET.

- Add professional word processing to your applications
- Royalty-free Windows Forms text box
- True WYSIWYG, nested tables, text frames, headers and footers, images, bullets, structured numbered lists, zoom, dialog boxes, section breaks, page columns
- Load, save and edit DOCX, DOC, PDF, PDF/A, RTF, HTML, TXT and XML

FusionCharts | from $195.02

InfoSoft Global empowering human thoughts

Interactive and animated charts for ASP and ASP.NET apps.

- Liven up your Web applications using animated Flash charts
- Create AJAX-enabled charts that can change at client-side without invoking server requests
- Export charts as images/PDF and data as CSV for use in reporting
- Also create gauges, financial charts, Gantt charts, funnel charts and over 550 maps
- Used by over 13,500 customers and 250,000 users in 110 countries

Janus WinForms Controls Suite | from $757.44

Janus systems

Add Outlook style interfaces to your WinForms applications.

- Janus GridEX for .NET (Outlook style grid)
- Janus Schedule for .NET and Timeline for .NET (Outlook style calendar view and journal)
- Janus ButtonBar for .NET and ExplorerBar for .NET (Outlook style shortcut bars)
- Janus UI Bars and Ribbon Control (menus, toolbars, panels, tab controls and Office 2007 ribbon)
- Now includes Office 2007 visual style for all controls

We accept purchase orders.
Contact us to apply for a credit account.

US Headquarters
ComponentSource
650 Claremore Prof Way
Suite 100
Woodstock
GA 30188-5188
USA

European Headquarters
ComponentSource
30 Greyfriars Road
Reading
Berkshire
RG1 1PE
United Kingdom

Asia / Pacific Headquarters
ComponentSource
3F Kojimachi Square Bldg
3-3 Kojimachi Chiyoda-ku
Tokyo
Japan
102-0083

Sales Hotline - US & Canada:
(888) 850-9911
www.componentsource.com

VISA DISCOVER

GSA Schedule Contract GS-35F-0188R

DINO ESPOSITO

Live Data Binding in ASP.NET AJAX 4.0

Recently I moved into a new office and went through the extremely pleasant and gratifying experience of packing up all the programming books I had collected in more than 10 years.

Those books could tell you quite clearly how Web programming evolved in the past decade, and which technologies followed one another, each improving (and in many cases revolutionizing) the other.

One book in particular caught my attention. It covered cutting-edge (well, for that era) programming techniques for Internet Explorer 4. I couldn't resist the temptation to flip through its pages.

In the late 1990s, most big names in the computer industry were engaged in their first attempts to make the Web and the browser a dynamic and highly programmable environment.

Data binding was just one of many popular features being researched and developed. While the basic idea of data binding hasn't changed significantly over the years, a real transformation did occur as far as its application to the Web world. The common approach to implementing data binding over the Web is radically different today than in the late 1990s, and much of the difference is due to Asynchronous JavaScript and XML (AJAX).

In this column I'll discuss various forms of client-side data binding as they are coming out in ASP.NET AJAX 4.0. In particular, I'll focus on some advanced features of data binding and observable objects.

Basic Principles

Generally speaking, data binding is simply a program's ability to bind some members of a target component to the members of a data source. Rapid Application Development (RAD) tools such as Microsoft Access and Microsoft Visual Basic made data binding a successful reality.

RAD tools offered an easy and effective infrastructure for developers to bind visual elements to the members of a data source. For a long time, the data source was uniquely identified with a record-set data stream originating from a SQL query.

In this regard, data binding nicely fits in a smart-client scenario but poses additional issues if employed in a Web-client scenario. For example, how would you flow records from the origin database server down to the requesting JavaScript-equipped browser?

Among many other things, that old book I opened up after years of oblivion discussed the data binding features of IE4 based on a couple of special ActiveX components, aptly named the Tabular Data Control (TDC) and Remote Data Services (RDS) control. **Figure 1** illustrates the overall architecture of client-side data binding as it was envisioned by IE4 architects in the beginning of the dynamic Web era.

A made-to-measure ActiveX control manages the connection with a remote data source and takes care of downloading (and optionally caching) data. The data source is any ODBC-compliant data source with RDS; it's a server-side text file with TDC.

The actual binding between source and target elements is implemented through browser-specific HTML attributes such as datasrc, datafld and dataformatas. Here's a quick example:

```
<span id="Label1" datasrc="#rdsCustomers"
datafld="CompanyName" />
```

The content of the bound field—CompanyName in the example—takes up the space reserved for the SPAN tag in the resulting Document Object Model (DOM).

What does the actual job of inserting data into the DOM? As mentioned, it's the browser that in this case does the trick. When the browser encounters any dataXXX attributes, it queries the data source control for data using an internal, contracted API. Any data it gets is then inserted into the final DOM and displayed to the user.

As you can see, the solution is strongly browser-specific and understandably never captured the heart of developers.

Client-side data binding was then set aside for a few years as ASP.NET and its server-side programming model gained wide acceptance.

In recent years, the advent of AJAX generated a lot of interest around client-side data binding, and engineers were back at work finding an effective (and this time cross-browser) way to make it happen.

Evolution in ASP.NET AJAX

Figure 2 shows the overall architecture of client-side data binding as it's implemented in ASP.NET AJAX 4.0.

Although the architectures depicted in **Figure 1** and **Figure 2** may look similar at first glance, they actually differ in a number of key aspects.

First and foremost, with ASP.NET AJAX 4.0 you can build client-side data-binding solutions that work with any modern browser. The glue code required to actually bind source data to target elements is now part of the Microsoft Ajax JavaScript library. As such, it can be hosted in any browser.

This column is based on a prelease version of ASP.NET AJAX 4.0. All information is subject to change.

Send your questions and comments for Dino to cutting@microsoft.com.

Furthermore, for binding you no longer have proprietary and non-standard HTML attributes such as datasrc and datafld that the browser must resolve. Instead, binding information is specified using XHTML-compliant, namespaced custom attributes resolved by code in the JavaScript library. It can also be done imperatively.

Old-Style versus New-Style

Another significant difference lies in the structure of the data source object. In old-style data binding, you use a sort of black-box proxy that manages data retrieval for you. In ASP.NET AJAX 4.0, you don't need any such ActiveX or binary components. All you need is a JavaScript object with the source data.

The built-in binding infrastructure links together fields on the JavaScript source object and elements of the DOM.

In ASP.NET AJAX 4.0, such a binding infrastructure is built into the Sys.UI.DataView component. Functionally speaking, the DataView object operates in much the same way as the RDS control does in IE4-style client data binding. It connects directly to a remote endpoint to get and expose data.

However, there are some differences. The DataView object is a client control entirely written in JavaScript that requires no special support from the browser to run. As you can see, it's quite different from the RDS ActiveX control.

Moreover, the DataView control doesn't directly interface with a relational data source. Instead, it connects to any JavaScript Object Notation (JSON)- or JSON With Padding (JSONP)-enabled service, such as a Windows Communication Foundation endpoint, and exchanges data using JSON. The service can wrap any physical data store (including a relational database) or even be a plain wrapper around an Entity Framework model. As **Figure 2** illustrates, in ASP.NET AJAX 4.0 you can even have data binding in place without an outbound network connection to the server. If, say, upon loading, a page downloads some data onto the client machine, you can have data binding at work on locally cached data without any further roundtrip to the server.

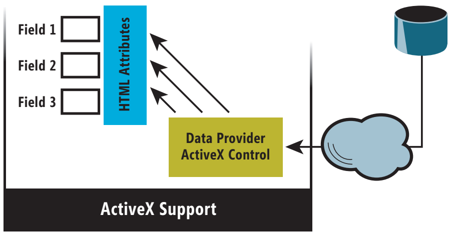


Figure 1 **Client-Side Data Binding in Internet Explorer 4**

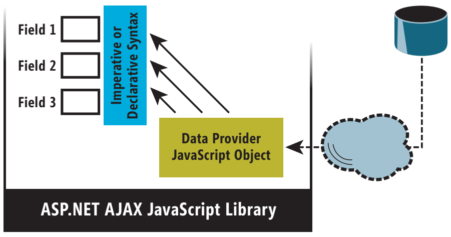

Field 1
Field 2
Field 3
Imperative or Declarative Syntax
Data Provider JavaScript Object
ASP.NET AJAX JavaScript Library

Figure 2 **Client-Side Data Binding in ASP.NET AJAX 4.0**

A Brief Summary

In the October 2009 installment of the Cutting Edge column (msdn.microsoft.com/magazine/ee309508.aspx), I covered the basics of data binding in ASP.NET AJAX 4.0 from a developer's perspective. While you can still refer to that article for deep coverage, I'm going to provide here a brief summary of the most important programming facts for client data binding.

In ASP.NET AJAX 4.0, client-side data binding can occur within a proper HTML template. A template is a DOM tree decorated with the sys-template CSS attribute:

```
<div sys:attach="dataview1" class="sys-template">
   ...
</div>
```

The sys-template is a conventional name for a user-defined CSS style that at the very minimum includes the following:

```
<style type="text/css">
   .sys-template { display:none; }
</style>
```

Decorating a given HTML tag with the sys-template attribute is not enough, however. You must also add some behavior to the tag that enables it to process any binding expressions inside. In this context, a behavior is just an instance of a made-to-measure JavaScript component or control.

A behavior can be attached to an HTML tag, either by instantiating the behavior programmatically or by using a declarative approach (by adding markup to the template tags in the page). For the declarative approach, the behavior must be associated with a public name (namespace prefix).

Here's an example of assigning a public name to an instance of the DataView component:

```
<body xmlns:sys="javascript:Sys"
      xmlns:dataview1="javascript:Sys.UI.DataView">
...
</body>
```

The sys:attach attribute gets a public name and creates an instance of the referenced behavior object. In the first code example in this section, the DIV tag is empowered with the behavior expressed by the object named dataview1. As you can guess, dataview1 is just the public name of the Sys.UI.DataView JavaScript object.

Once a DataView instance has been attached to an ASP.NET AJAX 4.0 template, the DataView can successfully process any binding expressions contained in the templates. Here's a sample template with the simplest binding syntax:

```
<ul id="imageListView" class="sys-template"
     sys:attach="dataview1"
    dataview1:data="{{ imageArray }}">
    <li>
        <span>{{ Name }}</span>
        <span>{{ Description }}</span>
    </li>
</ul>
```

The {{expression}} token tells the DataView to process the embedded JavaScript expression and assign the result to the DOM or the specified DataView property. For example, the code above assigns the content of a JavaScript variable named imageArray to a DataView property named data. In this way, you define the data source of the binding operation. This information is used to expand any other binding expressions within the template, such as those used above in the body of the two SPAN tags. Name and Description are expected to be public properties on the data item or items assigned to the data property of the DataView.

Inline Expression Evaluation

The {{expression}} syntax indicates the simplest type of binding supported by ASP.NET AJAX 4.0. Any bindings expressed in this way are evaluated only when the template is rendered. They are never updated unless the template is refreshed programmatically.

An inline expression is evaluated against the current state of the data source at rendering time, but doesn't automatically capture any further changes made to the data source.

An alternative, richer binding model is also supported that gives you the same programming power of XAML data binding in Windows Presentation Foundation and Silverlight applications. Such a set of advanced data binding features are commonly referred to as *live binding*. Let's find out more.

Live Binding

Live binding ensures that the value displayed in the user interface is automatically updated any time the bound value in the data source changes. For example, suppose you establish a live binding between a SPAN tag and the CompanyName property in a data source. The SPAN tag displays the current value of the CompanyName property at the time of rendering. However, the content of the SPAN tag will be automatically updated any time the bound data source property undergoes changes. Live binding can be applied to any two objects, whether DOM elements or JavaScript objects.

A different syntax is required for live binding expressions. Here's an example:

```
<span>{binding CompanyName}</span>
```

You use a single pair of curly brackets to wrap the expression, and prefix it with the keyword binding. As you can see, the overall syntax has much in common with the equivalent XAML syntax, and that isn't coincidental.

It should be noted that the content of the SPAN tag shown earlier is updated whenever a change is detected on the currently bound data item; the reverse won't work. If the content of the SPAN is updated, that change won't be propagated to the source object.

Live binding also can be described as a one-way binding that may happen repeatedly as updates on the data source are detected.

Hold on! The strategy employed to detect changes on bindings is a key point and I'll have more to say about it in a moment, but not before introducing two-way binding.

Two-Way Data Binding

Two-way data binding is a special form of live binding that uses two channels to detect changes on the binding. When two-way binding is in place between two objects, the following happens:

- If the data source changes, the target object is automatically refreshed.
- If the target object changes, the new state is propagated to the underlying data source.

Put another way, two-way data binding ensures that the source and target objects are always in sync.

No Tier Crossing

The following may perhaps sound like a foregone conclusion, but let me make it clear anyway. Imagine you have two-way binding between a piece of user interface and some data that a service retrieved from a database.

Figure 3 **The Sys.BindingMode Enumeration**

Member	Value	Description
auto	0	Data flow is twoWay if the target is an input element, select element, textArea element or component that implements the Sys. INotifyPropertyChange Interface. Data flow is oneWay otherwise.
oneTime	1	Data is transferred from the source object to the target only one time when the template is rendered.
oneWay	2	Data is transferred from the source object to the target whenever the source is updated.
twoWay	3	Data is transferred from the source object to the target whenever the source is updated, and from target to source whenever the target is updated.
oneWaytoSource	4	Data is transferred from target to source whenever the target is updated.

The data downloaded from the server and bound to the visual elements of the user interface represent a snapshot of the domain model. Say, for example, it represents a Customer object. If the displayed Customer object is modified in a two-way binding page, then all detected changes are mirrored to the client-side Customer object, but in no way will they be propagated to the server. Two-way data binding doesn't cross any tiers.

In terms of syntax, two-way binding is nearly the same as one-way live binding.

An ASP.NET AJAX binding is expressed via a Sys.Binding JavaScript object. You can control the direction of the data flow through the mode attribute on the Sys.Binding object that the framework creates for you any time you use the live binding syntax. (Note that no Sys.Binding object is created when you opt for plain {{…}} inline expressions.)

The following code snippet shows how to set up two-way data binding using the mode attribute:

```
<span id="Label1">{binding CompanyName, mode=twoWay}></span
```

Possible values for the mode attribute are summarized in **Figure 3**.

Live Binding in Action

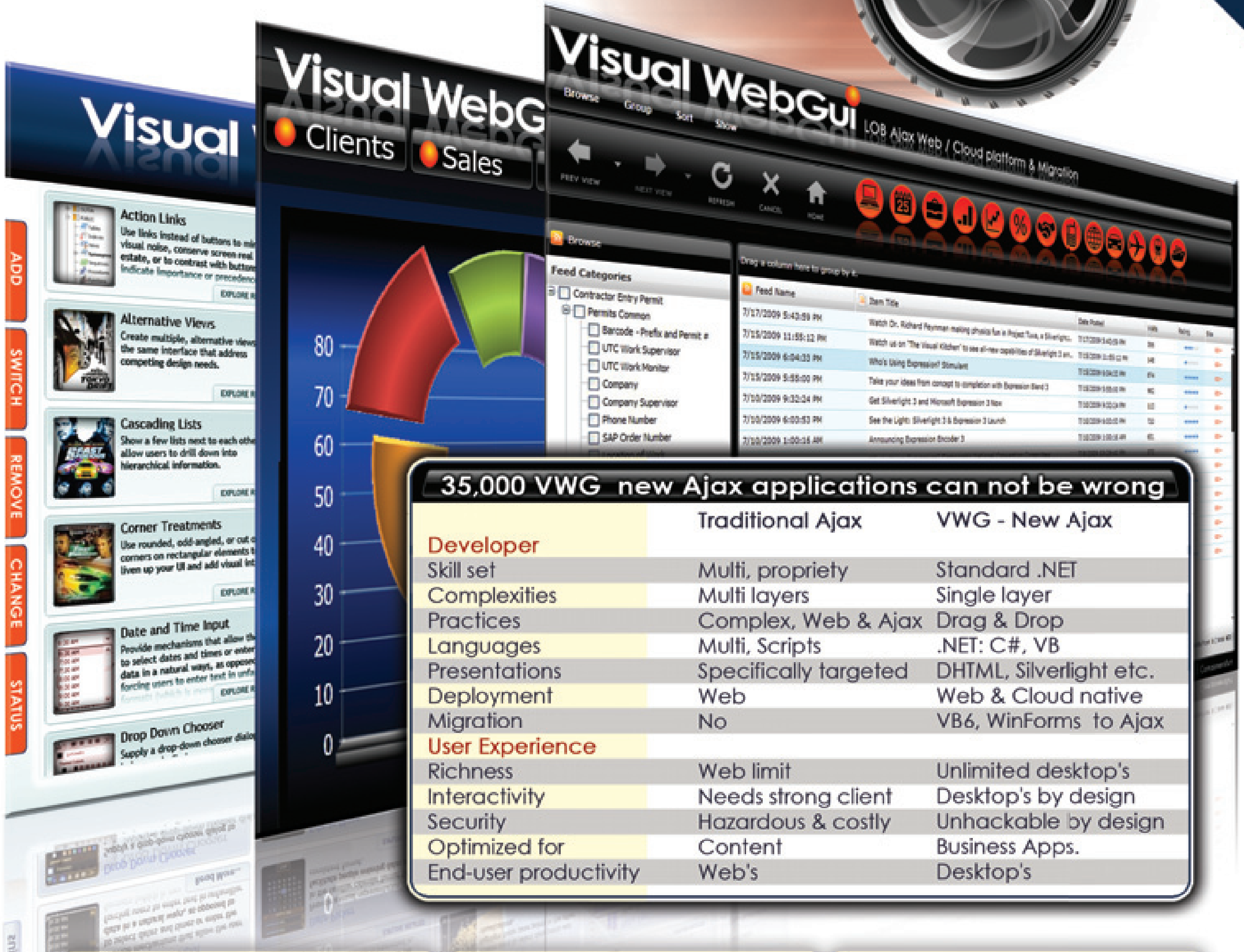
Figure 4 shows sample code that demonstrates live, two-way data binding. The page contains one template that is rendered using a DataView control. The data source object is a JavaScript array named theCustomers. Don't be fooled by the fact that theCustomers is a local object statically defined within the client browser. What really matters is that theCustomers is ultimately assigned to the data property of the DataView object. The DataView object has a rich programming interface that allows you to put into the data property any content, including content downloaded from a Web service.

For each bound data item, the template emits an LI tag that includes a text box. The text box is bound to the CompanyName property of the source. In the same template, a SPAN tag is also bound to the CompanyName property of the source.

We didn't invent traditional Ajax...

But boy, we made it so much better for business applications

Visit us at Tech-Ed Europe 2009 & PDC09



35,000 VWG new Ajax applications can not be wrong

	Traditional Ajax	VWG - New Ajax
Developer		
Skill set	Multi, propriety	Standard .NET
Complexities	Multi layers	Single layer
Practices	Complex, Web & Ajax	Drag & Drop
Languages	Multi, Scripts	.NET: C#, VB
Presentations	Specifically targeted	DHTML, Silverlight etc.
Deployment	Web	Web & Cloud native
Migration	No	VB6, WinForms to Ajax
User Experience		
Richness	Web limit	Unlimited desktop's
Interactivity	Needs strong client	Desktop's by design
Security	Hazardous & costly	Unhackable by design
Optimized for	Content	Business Apps.
End-user productivity	Web's	Desktop's

No other framework had dared before!

VWG Withstood a $10,000 **Hacking challenge**

New $10,000 Code Challenge!

Write VWG Webmail with **Less Code Lines**

DOWNLOAD FREE TRIAL

Free Open Source or 30 Days Evaluation

www.VisualWebGui.com

LOB Ajax Web / Cloud platform & Migration **Visual WebGui**

Figure 4 **Live, two-way Binding Sample**

```
<asp:Content ID="Content2" runat="server" ContentPlaceHolderID="PH_Head">

    <link rel="stylesheet" type="text/css"
      href="../../Css/lessantique.css" />
    <style type="text/css">
        .sys-template { display:none; }
    </style>

    <script type="text/javascript">
        var theCustomers = [
           { ID: "ALFKI", CompanyName:
            "Alfred Futterkiste" },
           { ID: "CONTS", CompanyName:
           "Contoso" }
        ];
    </script>
</asp:Content>

<asp:Content ID="Content5"
  ContentPlaceHolderID="PH_Body"
  runat="server">

    <asp:ScriptManagerProxy
    runat="server">
        <Scripts>
          <asp:ScriptReference Path=
"http://ajax.microsoft.com/ajax/beta/0910/MicrosoftAjaxTemplates.js" />
        </Scripts>
    </asp:ScriptManagerProxy>

    <div id="customerList">
       <ul class="sys-template"
         sys:attach="dataview" dataview:data="{{ theCustomers }}">
         <li>
           <span><b>{binding ID}</b></span>
           <input type="text" id="TextBox1"
             value="{binding CompanyName}" />
           <br />
           <span>Currently displaying...
<b>{binding CompanyName}</b></span>
         </li>
       </ul>
    </div>
</asp:Content>
```

Note that live bindings are not limited to the template they belong to. You can have the same expression—say, {binding CompanyName}—in two different templates. As long as the same data source object (or a compatible object) is attached to both templates, the binding will always be correctly resolved. **Figure 5** shows the page in action.

Initially the text box and the SPAN tag contain the same data. However, at some point the user may start editing the name of the company in the text box. Nothing happens until the user tabs out of the text box.

The editing phase is considered complete as soon as the text box loses focus. At this point, the two-way data binding mechanism triggers and updates the underlying data source object. Because the SPAN tag is also bound to the same data property through live binding, any changes are propagated.

To prevent the automatic update of the data source when an INPUT element is involved, you set the mode property explicitly, as shown below:

```
<input type="text"
value="{binding CompanyName, mode=oneWay}"
/>
```

Enter this change to the code in **Figure 4** and see the difference.

Detecting Changes

When the mode property of a binding is not set explicitly, it takes the auto value as described in **Figure 3**. So when you attach a binding to any HTML elements that refer to input scenarios (such as INPUT, SELECT or TEXTAREA), the property mode defaults to twoWay. As a result, all the changes to the target made via the browser's user interface are automatically transferred to the source.

Note that there are two variations of a oneWay binding. The standard oneWay binding detects changes in the source and reflects them in the user interface. The alternate oneWayToSource does the reverse: it detects changes in the target and reflects them in the source object. Try using the following code:

```
<input type="text"
value="{binding CompanyName, mode=oneWayToSource}"
/>
```

The initial display of the page will contain empty text boxes. As you type, though, the new text is detected and processed as expected.

The two-way data binding option is also the default option for any bound JavaScript object that happens to implement the Sys.INotifyPropertyChange interface. (The interface is part of the Microsoft Ajax JavaScript library.)

When I first introduced live binding, I said that the target is updated whenever the source changes. The following explains which changes the framework can detect and how.

HTML input elements fire standard events when their state is changed or, at a minimum, they notify when they enter or exit an editing phase. Because these events are part of the HTML standard, any data binding solution based on that standard will work on any browsers.

For an update to one side of the binding to be reflected in the other, it has to be done in a way that is possible to detect. Suppose you have a binding where the source is a JavaScript array, as shown here:

```
<ul class="sys-template" sys:attach="dataview"
    dataview:data="{{ theCustomers }}">
  <li>
     <span ><b>{binding CompanyName}</b></span>
  </li>
</ul>
```

Try to update the CompanyName property on an object within the array. The markup shows a button that if clicked runs the enterChanges JavaScript function, shown here:

```
<span>{binding CompanyName}</span>
<input type="button" value="Enter changes"
onclick="enterChanges()" />
...
<script type="text/javascript">
    function enterChanges() {
        theCustomers[0].CompanyName =
       "This is a new name";
    }
</script>
```

The enterChanges function updates the CompanyName property on the first object in the array. As you can see, this clearly is an operation that updates the state of a bound object.

In the case of live binding, you should expect the SPAN tag to display the new value. If you try that, though, you will see that nothing happens.

That is because there is no cross-browser way to be notified of updates occurring to a plain old JavaScript object such as that. So

CRYSTAL REPORTS®

CRYSTAL REPORTS® IT JUST WORKS

A LEADER IN REPORTING

More than 5 million units shipped to developers since 1992.

THAT WORKS

Integrate reports with Java or .NET, access virtually any database, and publish to the Web with support for Firefox, Safari, and Internet Explorer.

WITH THE LATEST INNOVATIONS

Embed Flash to create compelling mashups, manage dashboards and reports on a single server, and share interactive reports affordably via new licensing options.

TEST DRIVE IT TODAY:

Visit us at sap.com/crystalreports/dev or contact us at 1-888-333-6007.

Visual Studio READERS 2008 CHOICE

1996 to 2008

18th Annual JOLT PRODUCTIVITY AWARD

MEMBERS CHOICE WINNER THE CODE PROJECT 2009

SAP®

Get ready for…
a data layer you can build in **60 seconds**

Presenting Telerik OpenAccess – finally an ORM that's easy to use.

Forward and Reverse Mapping Capabilities
OpenAccess meets your specific needs by letting you choose whether to start the mapping from an existing database or just persist the application classes that you already have. OpenAccess also supports a rich variety of mappings for collections and class hierarchies.

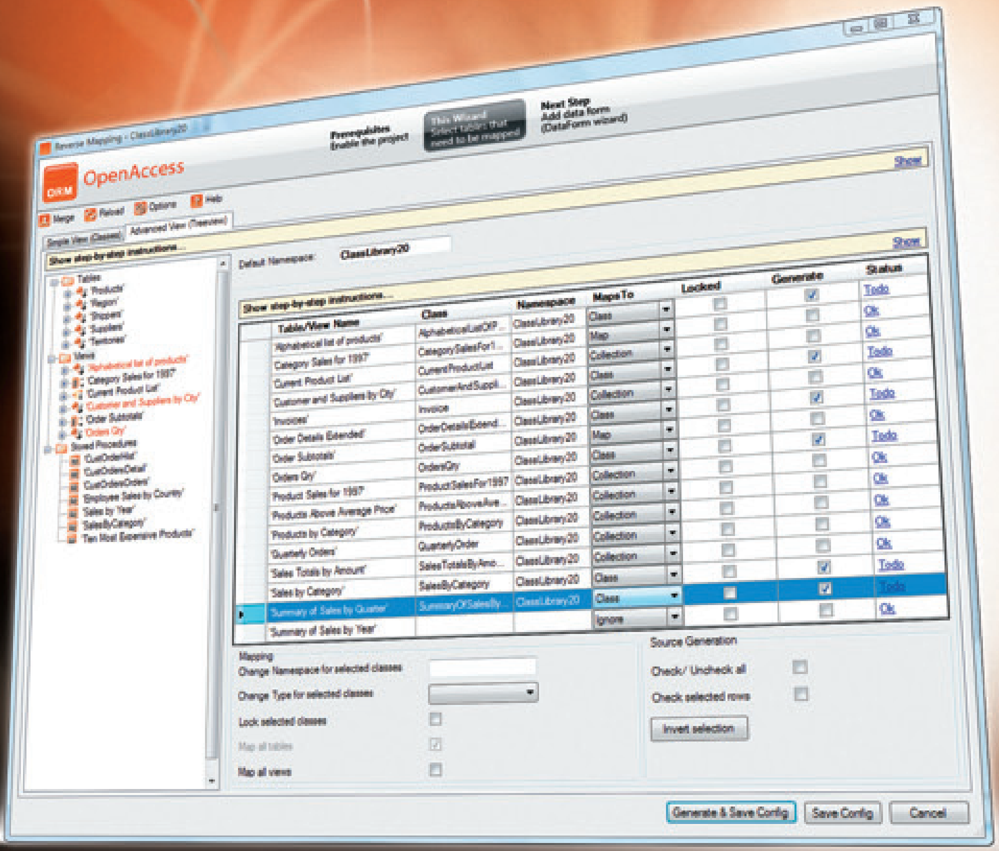
Tight Visual Studio Integration
OpenAccess it tightly integrated in Visual Studio .Net so you don't have to leave your favorite IDE. You will see your persistent classes in the solution explorer, nicely separated from your business logic code.

Intuitive Point-and-click Wizards
With OpenAccess ORM you can set up your persistent model codelessly and with just a few clicks. Intuitive yet powerful wizards will help you master the mapping process even if you haven't ever used an ORM before.

Full LINQ Support
OpenAccess ORM offers a complete support for LINQ, providing all features being used in the "Microsoft 101 LINQ Samples".

Advanced Features Out-of-the-box
OpenAccess offers an array of built-in heavy-duty features such as full support for SQL Azure, distributed level 2 cache, fetch plans, support for n-tier and disconnected applications, and many more.

US Sales: +1.888.365.2779 • Germany Sales: +49.89.8780687.70 • Europe HQ: +359.2.80.99.850 • e-mail: sales@telerik.com


Reverse Mapping - ClassLibrary20

Prerequisites
Enable the project

This Wizard
Select tables that need to be mapped

Next Step
Add data form
(DataForm wizard)

OpenAccess

Merge · Reload · Options · Help

Simple View (Classes) · Advanced View (Treeview)

Show step-by-step instructions...

- Tables
  - 'Products'
  - 'Region'
  - 'Shippers'
  - 'Suppliers'
  - 'Territories'
- Views
  - 'Alphabetical list of products'
  - 'Category Sales for 1997'
  - 'Current Product List'
  - 'Customer and Suppliers by City'
  - 'Order Subtotals'
  - 'Orders Qry'
- Stored Procedures
  - 'CustOrderHist'
  - 'CustOrdersDetail'
  - 'CustOrdersOrders'
  - 'Employee Sales by Country'
  - 'Sales by Year'
  - 'SalesByCategory'
  - 'Ten Most Expensive Products'

Default Namespace: ClassLibrary20

Show step-by-step instructions... Show

Table/View Name	Class	Namespace	Maps To	Locked	Generate	Status
'Alphabetical list of products'	AlphabeticalListOfP...	ClassLibrary20	Class	☐	✓	Todo
'Category Sales for 1997'	CategorySalesFor1...	ClassLibrary20	Map	☐	☐	Ok
'Current Product List'	CurrentProductList	ClassLibrary20	Collection	☐	☐	Ok
'Customer and Suppliers by City'	CustomerAndSuppli...	ClassLibrary20	Class	☐	✓	Todo
'Invoices'	Invoice	ClassLibrary20	Collection	☐	☐	Ok
'Order Details Extended'	OrderDetailsExtend...	ClassLibrary20	Class	☐	✓	Todo
'Order Subtotals'	OrderSubtotal	ClassLibrary20	Map	☐	☐	Ok
'Orders Qry'	OrdersQry	ClassLibrary20	Class	☐	✓	Todo
'Product Sales for 1997'	ProductSalesFor1997	ClassLibrary20	Collection	☐	☐	Ok
'Products Above Average Price'	ProductsAboveAve...	ClassLibrary20	Collection	☐	☐	Ok
'Products by Category'	ProductsByCategory	ClassLibrary20	Collection	☐	☐	Ok
'Quarterly Orders'	QuarterlyOrder	ClassLibrary20	Collection	☐	☐	Ok
'Sales Totals by Amount'	SalesTotalsByAmo...	ClassLibrary20	Collection	☐	☐	Ok
'Sales by Category'	SalesByCategory	ClassLibrary20	Class	☐	✓	Todo
'Summary of Sales by Quarter'	SummaryOfSalesBy...	ClassLibrary20	Class	☐	✓	Todo
'Summary of Sales by Year'			Ignore	☐	☐	Ok

Mapping
Change Namespace for selected classes
Change Type for selected classes
Lock selected classes ☐
Map all tables ✓
Map all views ☐

Source Generation
Check/ Uncheck all ☐
Check selected rows ☐
Invert selection

Generate & Save Config · Save Config · Cancel

UI COMPONENTS	DATA	PRODUCTIVITY TOOLS	AUTOMATED TESTING	TFS TOOLS	CMS
ASP.NET AJAX	OpenAccess ORM	JustCode	Web Testing Tools	Work Item Manager	Sitefinity
Silverlight	Reporting			Project Dashboard	
ASP.NET MVC					
WinForms					
WPF					

www.telerik.com/ORM

telerik
deliver more than expected

changes happen but the binding isn't aware of them and the user interface isn't refreshed.

Would polling the state of a plain JavaScript object be a viable solution? Probably not, and the development team reasonably ruled out that option, essentially for scalability reasons.

In the end, is using input HTML elements bound to data the only possibility for making data changes in a way that will successfully trigger other live bindings? Well, not exactly. The Microsoft Ajax JavaScript library features a static API through which you can "observe" the changes of any JavaScript object. This API is also available in a flavor that transforms a plain JavaScript object into an observable object for the binding machinery to detect updates.

Observable JavaScript Objects

An observable JavaScript object is an object endowed with additional functionality that raises change notifications when modified. Additional functionality is codified through the Sys.Observer interface. Note that changes made directly, without going through the interface, will not raise change notifications and will be ignored by the binding infrastructure.

Observable objects fit perfectly in a scenario where you want to establish live bindings between visual elements and JavaScript objects, such as those you might get from a remote Web service.

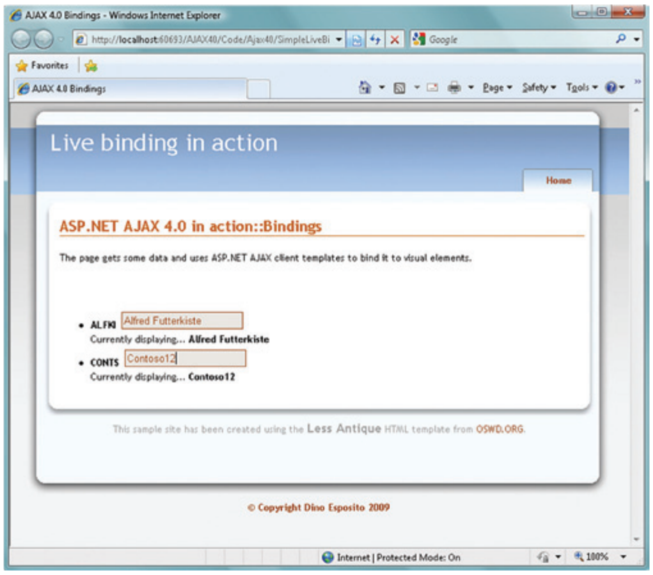
There are two ways to work with observable objects. One entails that you make a given object observable by adding some dynamic code to it—not enough to make a plain JavaScript object a complex thing, but enough to add new capabilities. Here is an example:

```
<script type="text/javascript">
    var theCustomers = [
            { ID: "ALFKI", CompanyName:
            "Alfred Futterkiste" },
            { ID: "CONTS", CompanyName:
            "Contoso" }
        ];
    function pageLoad() {
        Sys.Observer.makeObservable(theCustomers);
    }

    function onInsert() {
        var newCustomer = { ID: "ANYNA",
        CompanyName: "AnyNameThatWorks Inc" };
        theCustomers.add(newCustomer);
    }
</script>
```

The Sys.Observer.makeObservable method takes a JavaScript object (including arrays) and adds methods to it that you can use to make changes to the object that the bindings can detect. Note that having an observable array provides methods for changing the array in an observable way—so you can detect insertions and deletions. But it does not automatically provide the corresponding methods for modifying the properties of the individual items in the array in an observable way. For that, you can separately call makeObservable on the individual items, and they will then also have additional methods added.

As I mentioned earlier, the following code associated with a click event won't trigger the binding:

```
<script type="text/javascript">
    function enterChanges() {
        theCustomers[0].CompanyName =
        "This is a new name";
    }
</script>
```


AJAX 4.0 Bindings - Windows Internet Explorer

http://localhost:60693/AJAX40/Code/Ajax40/SimpleLiveBi

Live binding in action

Home

ASP.NET AJAX 4.0 in action::Bindings

The page gets some data and uses ASP.NET AJAX client templates to bind it to visual elements.

- ALFKI Alfred Futterkiste
  Currently displaying... **Alfred Futterkiste**
- CONTS Contoso12
  Currently displaying... **Contoso12**

This sample site has been created using the Less Antique HTML template from OSWD.ORG.

© Copyright Dino Esposito 2009

Internet | Protected Mode: On

Figure 5 **Live Binding in Action**

This code, however, will trigger the binding:

```
<script type="text/javascript">
    function enterChanges() {
       System.Observer.setValue(theCustomers[0],
       "CompanyName", "New name");
    }
</script>
```

What if the observed object has child objects? No worries: the setValue method knows how to handle the "dotted" syntax:

```
System.Observer.setValue(theCustomers[0],
"Company.Address.City", "Rome");
```

Finally, note that the observer pattern can be applied to any object you may encounter in the context of a Web page, including DOM elements, behaviors and even browser objects such as window.

Static and Dynamic

Most times when you need data binding in an application, you also need it to be live, at least one-way, if not two-way. In ASP.NET AJAX 4.0, data binding can be both static—that is, a simple inline evaluation of data values during rendering—and dynamic, in the sense that it can detect changes in source or target and apply them.Not all updates can be detected and used to refresh bindings. ASP.NET AJAX 4.0 easily recognizes changes entered into bound objects through visual elements. But for changes entered programmatically into JavaScript objects or arrays, there's no reliable cross-browser way to have live detection of changes. The trick in ASP.NET AJAX consists of providing a way to make changes so that they're observable and thus can be detected by live bindings. This takes the form of appending some observable operations to the object or, as an alternative, using ad hoc Sys.Observer static methods to conduct updates. ■

DINO ESPOSITO *is the author of the upcoming "Programming ASP.NET MVC 2" (Microsoft Press, 2010). Based in Italy, Esposito is a frequent speaker at industry events worldwide. You can join his blog at weblogs.asp.net/despos.*

THANKS to the following technical experts for reviewing this article:
Dave Reed and Boris Rivers-Moore

RadControls *for*

Silverlight 3

Now with 35+ advanced UI controls, full .NET RIA Support, VS Extensions, free Testing tool, 5x better performance

UI COMPONENTS	DATA	PRODUCTIVITY TOOLS	AUTOMATED TESTING	TFS TOOLS	CMS
ASP.NET AJAX	OpenAccess ORM	JustCode	Web Testing Tools	Work Item Manager	Sitefinity
Silverlight	Reporting			Project Dashboard	
ASP.NET MVC					
WinForms					
WPF					

telerik

deliver more than expected

www.telerik.com

Europe HQ: +359.2.80.99.850 • US Sales: +1.888.365.2779 • Germany Sales: +49.89.8780687.70 e-mail: sales@telerik.com

JAMES MCCAFFREY

Pairwise Testing with QICT

A solid knowledge of pairwise testing principles is essential for all software testers, developers and managers. In this month's column, I explain exactly what pairwise testing is and provide you with complete C# source code for a production-quality pairwise testing tool named QICT. In short, pairwise testing is a technique that allows you to reduce a large, unmanageable set of test-case inputs to a much smaller set that is likely to reveal bugs in the system under test. The best way to explain pairwise testing, and to show you where I'm headed in this article, is by way of two screenshots. Consider the dummy Windows Form-based application shown in **Figure 1**. The application has four input parameters. The first parameter is a TextBox control that can accept "a" or "b". The second parameter is a group of RadioButton controls that can take a value of "c", "d", "e" or "f". The third parameter is a ComboBox control that can take a value of "g", "h" or "i". The fourth parameter is a CheckBox control that takes a value of either "j" or "k". So one test-case input set would be { "a", "c", "g", "j" }. The dummy application has a total of 2 * 4 * 3 * 2 = 48 possible input sets, which is certainly manageable. But imagine a card-playing application of some sort with five parameters, where each parameter can take on one of 52 values (to represent a card from a normal deck of playing cards, with replacement). In this situation there would be 52 * 52 * 52 * 52 * 52 = 380,204,032 possible input sets, which is likely to be unmanageable unless you could programmatically generate expected values for each test set input.

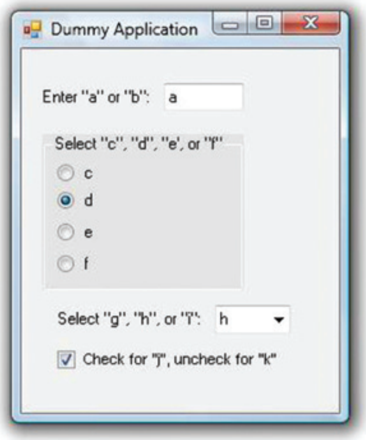

Dummy Application
Enter "a" or "b": a
Select "c", "d", "e", or "f"
c
● d
e
f
Select "g", "h", or "i": h
✓ Check for "j", uncheck for "k"

Figure 1 **A Dummy Application with Four Input Parameters**

The idea of pairwise testing is to generate a list of test sets that capture all possible pairs of parameter values from each parameter. For the example shown in **Figure 1**, there are a total of 44 such input pairs:

```
(a,c), (a,d), (a,e), (a,f), (a,g), (a,h), (a,i), (a,j), (a,k), (b,c),
(b,d), (b,e), (b,f), (b,g), (b,h), (b,i), (b,j), (b,k), (c,g), (c,h),
(c,i), (c,j), (c,k), (d,g), (d,h), (d,i), (d,j), (d,k), (e,g), (e,h),
(e,i), (e,j), (e,k), (f,g), (f,h), (f,i), (f,j), (f,k), (g,j), (g,k),
(h,j), (h,k), (i,j), (i,k)
```

Now the test set { "a", "c", "g", "j" } captures six of the 44 pairs: (a,c), (a,g), (a,j), (c,g), (c,j) and (g,j). So the goal of pairwise test set generation is to produce a collection of test sets that capture all 44 pairs. Take a look at the screenshot in **Figure 2**.

The screenshot shows a tool named qict.exe generating a collection of 12 test sets that capture all 44 input pairs for the scenario shown in **Figure 1**. If you trace through each pair of values in the 12 test sets generated in **Figure 2**, you'll see that they do in fact capture all 44 pairs listed above. So in this situation, we have reduced our possible test-case inputs from 48 test cases to 12 test cases. The savings aren't very significant for this small example, but as I'll show in a moment, using pairwise testing can dramatically reduce the number of test-case inputs in many situations. The underlying assumption of pairwise testing is that software bugs are more frequently found in code that involves the interaction of values from different parameters than in code that involves values from within a particular parameter. In other words, for the dummy application in **Figure 1**, application code that deals with inputs "a" and "g" is more likely to introduce a logic error than code that deals with inputs "a" and "b". This is a notion that is, in fact, supported by some research.

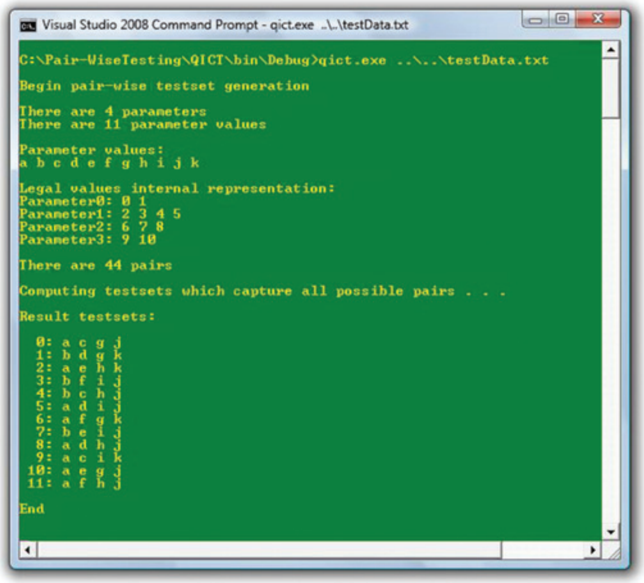
Using the PICT Tool

There are several pairwise test set generation tools available to you. My favorite tool in most situations is the PICT (Pairwise Independent Combinatorial Testing) tool. PICT was written by my colleague Jacek Czerwonka, who adapted code from an existing internal-Microsoft pairwise tool. PICT is available as a free download from several locations, including the Microsoft Tester Center page at msdn.microsoft.com/testing/bb980925.aspx. If you search the Internet, you will also find several other pairwise test set generation tools. However, PICT is a single executable that runs from a shell command line. PICT is very fast, very powerful and should meet your pairwise testing needs in most situations. I named the tool presented in this article QICT (which doesn't stand for anything in particular) to acknowledge the importance of the PICT tool.

So, why yet another pairwise test set generator? There are several reasons. First, although PICT is a wonderful tool, it is written in native C++ code and the source code is not available. The QICT tool presented here is, as far as I can tell, the first production-quality pairwise tool written with managed C# code. The availability of the code allows you to freely modify QICT to meet your own

Code download available at code.msdn.microsoft.com/mag1209TestRun.

Send your questions and comments for James to testrun@microsoft.com.


Visual Studio 2008 Command Prompt - qict.exe ..\..\testData.txt

```
C:\Pair-WiseTesting\QICT\bin\Debug>qict.exe ..\..\testData.txt

Begin pair-wise testset generation

There are 4 parameters
There are 11 parameter values

Parameter values:
a b c d e f g h i j k

Legal values internal representation:
Parameter0: 0 1
Parameter1: 2 3 4 5
Parameter2: 6 7 8
Parameter3: 9 10

There are 44 pairs

Computing testsets which capture all possible pairs . . .

Result testsets:

  0: a c g j
  1: b d g k
  2: a e h k
  3: b f i j
  4: b c h j
  5: a d i j
  6: a f g k
  7: b e i j
  8: a d h j
  9: a c i k
 10: a e g j
 11: a f h j

End
```

Figure 2 **Pairwise Test Set Generation with the QICT Tool**

needs. For example, you can modify QICT to directly read its input from an XML file or a SQL database, or you can modify QICT to directly emit results in a custom output format. And you may want to experiment with the tool's logic, say, for example, by introducing constraints (test input sets that are not permitted), by introducing required test sets, or changing how the tool generates its test set collection. Additionally, the availability of QICT source code allows you to copy and place pairwise test set generation code directly into a .NET application or test tool. Finally, although source code for a few pairwise test set generation tools is available on the Internet, some of these tools are quite inefficient. For example, consider a situation with 20 parameters, each of which has 10 values. For this scenario there are 10 * 10 * 10 * . . . * 10 (20 times) = 1020 = 100,000,000,000,000,000,000 possible test-case inputs. This is a lot of test cases. The PICT tool reduces this to only 217 pairwise test sets, and the QICT tool produces either 219 or 216 test sets (depending upon the seed value of a random number generator, as I'll explain shortly). However, one widely referenced pairwise test set generation tool written in Perl produces 664 sets. Finally, with the QICT source code available and this article's explanation of the algorithms used, you can recast QICT to other languages, such as Perl, Python, Java or JavaScript if you wish.

The QICT Tool

The code for the QICT tool is slightly too long to present in its entirety in this column, but the entire source code is available from the MSDN Code Gallery at code.msdn.microsoft.com. I will describe the algorithms and data structures I use, along with snippets of key code, so you'll have enough information to use and modify QICT as needed. The essence of how QICT works is to generate one test set at a time, using greedy algorithms to place each parameter value, until all possible pairs have been captured. The high-level algorithm for QICT is presented in **Figure 3**.

The key to implementing this high-level algorithm is determining what kind of data structures to use and what the various "best" options are. The QICT source code begins like this:

```
static void Main(string[] args)
{
  string file = args[0];
  Random r = new Random(2);
  int numberParameters = 0;
  int numberParameterValues = 0;
  int numberPairs = 0;
  int poolSize = 20;
```

I coded QICT using a traditional procedural style rather than taking an object-oriented approach so you can more easily refactor QICT to languages with limited OOP support, such as Perl and JavaScript. I first read an input file from the command line. As you can see, in order to keep my code clean and simple, I have left out normal error-checking you'd want to include. The input file for QICT is the same as that used by PICT, a simple text file that looks like:

```
Param0: a, b
Param1: c, d, e, f
etc.
```

Parameter names are followed by a colon character and a comma-delimited list of legal values for that parameter. Parameter values must be distinct. Next, I instantiate a Random object. The choice of a seed value of 2 is arbitrary, but any value will make QICT produce the same results for an input set every time it is run. I'll explain the purpose of the pseudo-random number object shortly. I declare three int variables that will be assigned values when the input file is read. For the example shown in **Figure 2**, numberParameters is 4, numberParameterValues is 11 and numberPairs is 44. The poolSize variable stores the number of candidate test sets to generate for each test set. If you experiment with QICT a bit, you'll see that the tool is impacted in a rather surprisingly minor way by adjusting the value for poolSize.

Figure 3 **QICT Algorithm**

```
read input file
create internal data structures

create an empty testset collection
while (number of unused pairs > 0)
  for i := 1 to candidate poolSize
    create an empty candidate testset
    pick the "best" unused pair
    place best pair values into testset
    foreach remaining parameter position
      pick a "best" parameter value
      place the best value into testset
    end foreach
  end for
  determine "best" candidate testset
  add best testset to testset collection
  update unused pairs list
end while

display testset collection
```

The heart of QICT is the declaration of the main data structures. The first four objects are:

```
int[][] legalValues = null;
string[] parameterValues = null;
int[,] allPairsDisplay = null;
List<int[]> unusedPairs = null;
```

The legalValues object is a jagged array where each cell in turn holds an array of int values. The legalValues array holds an in-memory representation of the input file, so cell 0 of legal values holds an array that in turn holds the values 0 (to represent parameter value "a") and 1 (to represent "b"). It turns out that working directly with string values is rather inefficient and that representing parameter values as integers yields significantly faster performance. The parameterValues string array holds the actual parameter values and is used at the end of QICT to display results as strings rather than ints. So, for the preceding example, cell 0 holds "a", cell 1 holds "b" and so on through cell 10, which holds "k". The allPairsDisplay object is a two-dimensional array of ints. It is populated by all possible pairs. For our example, cell [0,0] holds 0 (for "a") and cell [0,1] holds 2 (for "c")—the first possible pair. Cell [1,0] holds 0 and cell [1,1] holds 3 to represent the second pair, (a,d). The unusedPairs object is a generic List of int arrays. The first item in unusedPairs is initially {0,2}. I use a List collection for unusedPairs rather than an array because each time a new test set is added to the test sets collection, I remove the pairs generated by the new test set from unusedPairs. Additionally, this means I have a convenient stopping condition that will occur when unusedPairs.Count reaches 0.

The next four main program data structures are:

```
int[,] unusedPairsSearch = null;
int[] parameterPositions = null;
int[] unusedCounts = null;
List<int[]> testSets = null;
```

Most pairwise test set generation tools, including QICT, perform a huge number of searches. An efficient lookup approach is crucial for reasonable performance. Here I declare a two-dimensional array named unusedPairsSearch. It is a square array with size numberParameterValues by numberParameterValues, where each cell holds a 1 if the corresponding pair has not been used, and a 0 if the corresponding pair has been used or is not a valid pair. Initially, the first three rows of unusedPairsSearch for the example in **Figure 2** are:

```
0 0 1 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1 1
```

And so forth.

So row one means pairs (0,0) and (0,1)—that is, (a,a) and (a,b)—are not valid, while pairs (0,2), (0,3), . . . (0,10)—that is (a,c), (a,d) through (a,k)—have not yet been captured by a test set. The parameterPositions array holds the location within a test set of a specified parameter value. After initialization this array holds values:

```
0 0 1 1 1 1 2 2 2 3 3
```

The index of parameterPositions represents a parameter value, and the corresponding cell value represents its position in a test set. So the fourth cell from the left has index = 3 and value = 1, meaning parameter value 3 ("d") belongs at position 1 (the second slot) in a test set. The unusedCounts object is a one-dimensional array that holds the number of times a particular parameter value appears in the unusedPairs array. Initially unusedCounts holds:

```
9 9 7 7 7 7 8 8 8 9 9
```

The index represents a parameter value, and the corresponding cell value is the unused count. So, the fourth cell from the left has index = 3 and value = 7, meaning parameter value 3 ("d") initially appears in 7 unused pairs— (a,d), (b,d), (d,g), (d,h), (d,i), (d,j) and (d,k). The testSets object holds the pairwise test set results. It is initially empty but grows every time a new test set is generated. Each test set is represented by an int array. So, in **Figure 2**, the first test set in the result is {"a", "c", "g", "j" }, which is stored in the testSets List as an array with values {0,2,6,9}.

With the key data structures in place, QICT reads the input file to determine values for numberParameters and numberParameterValues, and to populate the legalValues and parameterValues arrays. I use the relatively crude approach of performing an initial read of the file, and then resetting the file pointer and performing a second pass through the file. Once legalValues is populated, I can scan through it to determine the number of pairs for the input:

```
for (int i = 0; i <= legalValues.Length - 2; ++i) {
  for (int j = i + 1; j <= legalValues.Length - 1; ++j) {
    numberPairs += (legalValues[i].Length * legalValues[j].Length);
  }
}
Console.WriteLine("\nThere are " + numberPairs + " pairs ");
```

After initialization, the first row of legalValues holds {0,1} and the second row holds {2,3,4,5}. Notice that the pairs determined by these two rows are (0,2), (0,3), (0,4), (0,5), (1,2), (1,3), (1,4), and (1,5), and that in general the number of pairs determined by any two rows in legalValues is the product of the number of values in the two rows, which equals the row Length property of the rows. The next part of QICT code populates the unusedPairs List:

```
unusedPairs = new List<int[]>();
for (int i = 0; i <= legalValues.Length - 2; ++i) {
  for (int j = i + 1; j <= legalValues.Length - 1; ++j) {
    int[] firstRow = legalValues[i];
    int[] secondRow = legalValues[j];
    for (int x = 0; x < firstRow.Length; ++x) {
      for (int y = 0; y < secondRow.Length; ++y) {
        int[] aPair = new int[2];
        aPair[0] = firstRow[x];
        aPair[1] = secondRow[y];
        unusedPairs.Add(aPair);
      }
    }
  }
}
```

Here I grab each pair of rows from legalValues using indexes i and j. Next, I walk through the values in each row pair using indexes x and y. Extensive use of multiple nested for loops like this is a hallmark of combinatorial code. When I write such code, I always draw by hand on a piece of paper the arrays involved because it's quite easy to make mistakes without a diagram. After populating the unusedPairs List, I use the same nested loop structure to populate the allPairsDisplay and unusedPairsSearch arrays. The initialization code next populates the parameterPositions array by iterating through legalValues:

```
parameterPositions = new int[numberParameterValues];
int k = 0;
for (int i = 0; i < legalValues.Length; ++i) {
  int[] curr = legalValues[i];
  for (int j = 0; j < curr.Length; ++j) {
    parameterPositions[k++] = i;
  }
}
```

The initialization code concludes by populating the unusedCounts array:

```
unusedCounts = new int[numberParameterValues];
for (int i = 0; i < allPairsDisplay.GetLength(0); ++i) {
  ++unusedCounts[allPairsDisplay[i, 0]];
  ++unusedCounts[allPairsDisplay[i, 1]];
}
```

INFUSE YOUR UI WITH THE POWER TO EMPOWER DECISION MAKERS

NetAdvantage for Silverlight Data Visualization is a comprehensive collection of User Interface Controls to **Build** Rich Dashboards, **Visualize** Business Data and **Empower** Decision Makers. Go to infragistics.com/sldv today to get the power of Infragistics behind you and create high end BI applications without writing a lot of code.

Infragistics Sales 800 231 8588
Infragistics Europe Sales +44 (0) 800 298 9055
Infragistics India +91-80-6785-1111

Infragistics
KILLER APPS. NO EXCUSES.

WINDOWS 7 IS HERE... VISUAL STUDIO 2010 IS COMING
PREPARE FOR THE WORLD OF
TOMORROW
WITH STUDIO ENTERPRISE 2009 V3

100'S OF CONTROLS, 7 PLATFORMS, FEATURING

WinForms • WPF • ASP.NET • Silverlight • iPhone • Mobile • ActiveX

Grids • Charts • Reports • Schedules • Menus • Toolbars • Ribbon • Data Input • Editors • PDF

DEVELOPER APPROVED FOR 20+ YEARS!

Location	Deposits	Withdrawls
NEW YORK CITY, NY	3307	
DUBAI, INDIA		1593
PITTSBURGH, PA		3977
BEIJING, CHINA		4745
DENVER, CO		3294
CHICAGO, IL	3587	
PARIS, FRANCE		4222

DOWNLOAD YOUR FREE TRIAL AT

WWW.COMPONENTONE.COM/PREPARE

ComponentOne **ComponentOne Sales:** 1.800.858.2739 or 1.412.681.4343

Figure 4 **Filling Test Set with Best Parameter Values**

```
for (int i = 2; i < numberParameters; ++i) {
  int currPos = ordering[i];
  int[] possibleValues = legalValues[currPos];
  int currentCount = 0;
  int highestCount = 0;
  int bestJ = 0;
  for (int j = 0; j < possibleValues.Length; ++j) {
    currentCount = 0;
    for (int p = 0; p < i; ++p) {
      int[] candidatePair = new int[] { possibleValues[j],
        testSet[ordering[p]] };
      if (unusedPairsSearch[candidatePair[0], candidatePair[1]] == 1 ||
        unusedPairsSearch[candidatePair[1], candidatePair[0]] == 1)
        ++currentCount;
    }
    if (currentCount > highestCount) {
      highestCount = currentCount;
      bestJ = j;
    }
  }
  testSet[currPos] = possibleValues[bestJ];
}
```

Here, as in many of the QICT routines, I take advantage of the fact that C# automatically initializes all cells in int arrays to 0. If you wish to recast QICT to an object-oriented style, all these initialization routines would likely best be placed either in an object constructor or perhaps in an explicit Initialize() method. The main processing loop begins:

```
testSets = new List<int[]>();
while (unusedPairs.Count > 0) {
  int[][] candidateSets = new int[poolSize][];
  for (int candidate = 0; candidate < poolSize; ++candidate) {
    int[] testSet = new int[numberParameters];
  // fill candidate testSets
  }
  // copy best testSet into testSets collection; upate data structues
}
```

Because the number of candidate test sets is known to be poolSize, I can instantiate an array rather than use a dynamic-sized List object. Notice that the size of the unusedPairs collection controls the main processing loop exit. Now it's time to pick the "best" unused pair—and now things start to become really interesting:

```
int bestWeight = 0;
int indexOfBestPair = 0;
for (int i = 0; i < unusedPairs.Count; ++i) {
  int[] curr = unusedPairs[i];
  int weight = unusedCounts[curr[0]] + unusedCounts[curr[1]];
  if (weight > bestWeight) {
    bestWeight = weight;
    indexOfBestPair = i;
  }
}
```

Here I define best to mean the unused pair that has the highest sum of unused individual parameter values. For example, if "a" appears one time in the current list of unused pairs, "b" appears two times, "c" three times and "d" four times, then pair (a,c) has weight 1 + 3 = 4, and pair (b,d) has weight (b,d) 2 + 4 = 6, so pair (b,d) would be selected over (a,c).

There are many other weighting schemes you might wish to explore. For example, using some sort of multiplication would give higher weights to pairs with extreme values of unused counts compared with pairs that have unused counts closer together. Another possibility is to track used counts—the number of times parameter values appear in the test sets already added to the result testSets collection—and pick as the best pair the one that has the least used counts. Once the best unused pair has been determined, I create a two-cell array to hold the pair values and determine the positions within a test set where each value belongs:

```
int[] best = new int[2];
unusedPairs[indexOfBestPair].CopyTo(best, 0);
int firstPos = parameterPositions[best[0]];
int secondPos = parameterPositions[best[1]];
```

At this point I have an empty test set and a pair of values to place in the test set, and I know the location within the test set where the values belong. The next step is to generate parameter values for the remaining positions in the test set. Now, rather than fill the test set positions in some fixed order (from low index to high), it turns out that it is much better to fill the test set in random order. First, I generate an array that holds the parameter positions in sequential order:

```
int[] ordering = new int[numberParameters];
for (int i = 0; i < numberParameters; ++i)
  ordering[i] = i;
```

Next, I rearrange the order by placing the known locations of the first two values from the best pair into the first two cells of the ordering array:

```
ordering[0] = firstPos;
ordering[firstPos] = 0;
int t = ordering[1];
ordering[1] = secondPos;
ordering[secondPos] = t;
```

And now I shuffle the remaining positions (from cell 2 and up) using the Knuth shuffle algorithm. This is why I created a Random object at the beginning of the QICT code. The number of test sets produced by QICT is surprisingly sensitive to the value of the pseudo-random number generator seed value, so you may want to experiment with several seed values. For the situation with 20, 10-value parameters I described earlier, using a seed value of 2 generates 219 test sets, and a seed value of 6 generates 216 test sets, but a seed value of 0 yields 221 test sets.

```
for (int i = 2; i < ordering.Length; i++) {
  int j = r.Next(i, ordering.Length);
  int temp = ordering[j];
  ordering[j] = ordering[i];
  ordering[i] = temp;
}
```

After shuffling, I place the two values from the best pair into the candidate test set:

```
testSet[firstPos] = best[0];
testSet[secondPos] = best[1];
```

Now comes the most important part of the QICT algorithm. I must determine the best parameter values to place in each of the empty test set positions. The technique I use is another greedy approach. For each parameter position, I test each possible legal value at that position, by counting how many unused pairs in the test value, when combined with the other values already in the test set capture. Then I select the parameter value that captures the most unused pairs. The code to do this is the trickiest part of QICT and is listed in **Figure 4**.

The outermost loop in **Figure 4** is a count of the total number of test set positions (given by numberParameters), less two (because two spots are used by the best pair). Inside that loop I fetch the position of the current spot to fill by looking into the ordering array I created earlier. The currentCount variable holds the number of unused pairs captured by the test parameter value. Notice that because I am filling test set positions in random order, the candidate pair of values can be out of order, so I need to check two possibilities when I do a lookup into the unusedPairsSearch array. At the end of the code in **Figure 4**, I will have a candidate test set that has values in every position that were selected using greedy algorithms. Now I simply add this candidate test set into the collection of candidates:

```
candidateSets[candidate] = testSet;
```

At this point I have n = poolSize candidate test sets and I need to select the best of these to add into the primary testSet result collection. I could assume that the first candidate test set captures the most unused pairs and simply iterate through each candidate starting at position 0, but again, introducing some randomness produces better results. I pick a random spot within the candidates and assume it is the best candidate:

```
int indexOfBestCandidate = r.Next(candidateSets.Length);
int mostPairsCaptured =
  NumberPairsCaptured(candidateSets[indexOfBestCandidate],
    unusedPairsSearch);
```

Here I use a little helper function named NumberPairsCaptured() to determine how many unused pairs are captured by a given test set. The helper function is:

```
static int NumberPairsCaptured(int[] ts, int[,] unusedPairsSearch)
{
  int ans = 0;
  for (int i = 0; i <= ts.Length - 2; ++i) {
    for (int j = i + 1; j <= ts.Length - 1; ++j) {
      if (unusedPairsSearch[ts[i], ts[j]] == 1)
        ++ans;
    }
  }
  return ans;
}
```

Now I walk through each candidate test set, keeping track of the location of the one that captures the most unused pairs:

```
for (int i = 0; i < candidateSets.Length; ++i) {
  int pairsCaptured = NumberPairsCaptured(candidateSets[i],
    unusedPairsSearch);
  if (pairsCaptured > mostPairsCaptured) {
    mostPairsCaptured = pairsCaptured;
    indexOfBestCandidate = i;
  }
}
```

And now I copy the best candidate test set into the main result testSets List object:

```
int[] bestTestSet = new int[numberParameters];
candidateSets[indexOfBestCandidate].CopyTo(bestTestSet, 0);
testSets.Add(bestTestSet);
```

At this point, I have generated and added a new test set, so I must update all the data structures that are affected, namely, the unusedPairs List (by removing all pairs that are generated by the new test set), the unusedCounts array (by decrementing the count for each parameter value in the new test set), and the unusedPairsSearch matrix (by flipping the values associated with each pair generated by the new test set from 1 to 0).

Now I'm at the end of my main processing loop. I continue generating candidates, selecting the best candidate, adding the best candidate to testSets and updating data structures operations. The processing will end when the number of unused pairs reaches 0.

Then I display the final results:

```
Console.WriteLine("\nResult testsets: \n");
for (int i = 0; i < testSets.Count; ++i) {
  Console.Write(i.ToString().PadLeft(3) + ": ");
  int[] curr = testSets[i];
  for (int j = 0; j < numberParameters; ++j) {
    Console.Write(parameterValues[curr[j]] + " ");
  }
  Console.WriteLine("");
}
  Console.WriteLine("");
}
```

As I mentioned earlier, if you are modifying QICT to suit your own particular testing scenario, you may want to emit results directly to an XML file, a SQL database or some other form of storage.

Produce Better Systems

Pairwise testing is a combinatorial technique with probabilistic factors. Pairwise test set generation is an important technique, but it isn't magic. Remember that pairwise techniques simply reduce the number of test-case inputs in situations where you just have too many test cases to deal with. Pairwise test set generation does not create test-case expected results. You should always begin by using normal testing principles, such as looking at boundary conditions, using pure random input and so on, and then use pairwise testing to supplement your test-case generation. Additionally, as a general rule of thumb, more testing is better, so there's no reason why you can't add additional test-case inputs to those produced by pairwise generation tools. Although pairwise testing is useful in many situations, be sure to use it only when appropriate.

I have found pairwise test set generation to be very useful for configuration testing, for module testing methods that accept enumerated values and for testing SQL databases where each column in a table has a relatively small number of different values. Pairwise testing is not necessarily a good approach for scenarios where you have a relatively small number of test-case inputs, or when you can programmatically produce test-case-expected results (and therefore deal with a large test-case input set). And pairwise testing is not normally usable when the input values to the system under test are not discrete. However, even in situations where the number of possible parameter values is very large, you may be able to effectively use pairwise test-case input generation by separating parameter values into equivalence classes. When used properly, pairwise test set generation is an important technique that can help you produce better software systems. ■

DR. JAMES MCCAFFREY *works for Volt Information Sciences Inc., where he manages technical training for software engineers based at Microsoft's Redmond, Wash., campus. He has worked on several Microsoft products, including Internet Explorer and MSN Search, and is the author of ".NET Test Automation: A Problem-Solution Approach" (Apress, 2006). James can be reached at jmccaffrey@volt.com or v-jammc@microsoft.com.*

THANKS to the following technical experts for reviewing this article: *Jacek Czerwonka*

Automated Unit Tests for Legacy Code with Pex

Nikhil Sachdeva

In my previous life I was a consultant. One of my clients, a leading bank, wanted to automate its loan origination process. The bank already had a system in place that included a Windows-based application, a proprietary back end and a mainframe system that was also the heart of their solution. My job was to integrate the existing systems with a set of applications being developed for the accounts division.

The client had built a Web service that communicated with the mainframe. At first it seemed pretty simple. All I had to do was to hook into the service, get the information and pass it to the new accounts application. But it's never that simple.

This article is based on a prerelease version of Pex. All information is subject to change.

This article discusses:

- Keeping legacy code fresh
- Unit testing with Pex
- Factories, stubs, and moles
- Updating tests with code changes

Technologies discussed:

Visual Studio 2008, Pex

Code download available at:

code.msdn.microsoft.com/mag200912PEX

During implementation I found that the new systems expected a Loan Originator attribute, but the Web service GetLoanDetails method did not return that information. It turns out that the service was created years ago by a developer who's no longer with the company. The bank has been using the service without any modifications because it has many layers and everyone is afraid of breaking something.

That Web service is legacy code.

Eventually, we built a lightweight service wrapper so the new systems could call it for any new information and continue using the old service as well. It would have been much easier to modify the existing service if a consistent and testable design had been followed.

Keeping Code Fresh

Legacy code is something that was developed in the past and is still used, but is hard to maintain and change. The reasons for sustaining these systems generally revolve around the cost and time involved in building a similar new system—though sometimes there is a lack of awareness about the future implications of current coding efforts.

The fact is that over a period of time code begins to rot. This can be because of requirement changes, a not-very-well-thought-through design, applied anti-patterns, or lack of appropriate tests. The end result is code that is hard to maintain and difficult to change.

There are many approaches to prevent your code from rotting, but one of the most effective can be to write code that is testable and then generate sufficient unit tests for the code. Unit tests act as agents that continuously probe the system for uncovered paths, identify troublesome errors, and provide indicators whether changes introduced into a subsystem have a good or bad effect on the overall software. Unit tests give confidence to the developer that the any code change will not introduce any regressions.

Having said that, creating and maintaining a good unit test suite can be a challenge in itself. It's possible to end up writing more code for the test suite than the code under test. Another challenge is dependencies within the code. The more complex the solution, the more dependencies you are likely to find between classes. Mocks and stubs are a recognized way of removing these dependencies and testing the code in isolation, but these require additional knowledge and experience from the developer to create effective unit tests.

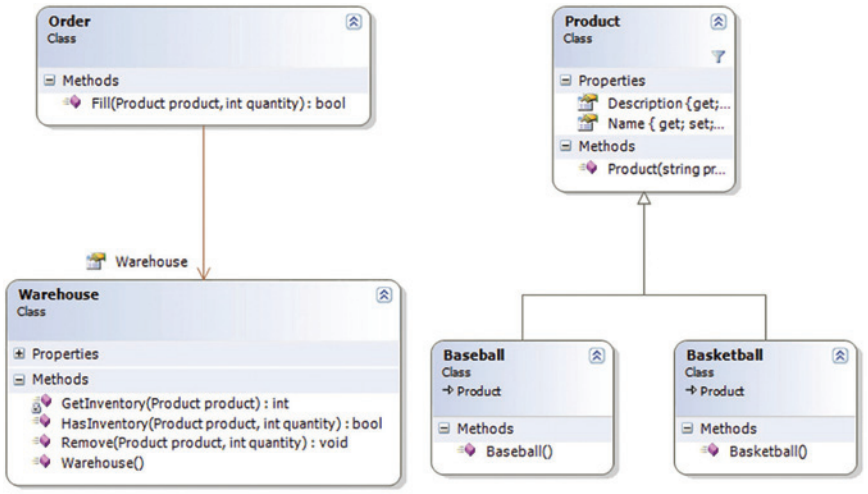


Figure 1 **A Legacy Order Fulfillment System**

Pex to the Rescue

Pex (research.microsoft.com/projects/pex/) is a tool developed by Microsoft Research to automatically and systematically produce the minimal set of test inputs needed to execute a finite number of finite paths. Pex automatically produces a small test suite with high code and assertion coverage.

Pex finds interesting input-output values of your methods, which can then be saved as a small test suite with high code coverage. Pex performs a systematic analysis, hunting for boundary conditions, exceptions and assertion failures that you can debug right away. Pex also enables parameterized unit testing (PUT), an extension of unit testing that reduces test maintenance costs, and it utilizes dynamic symbolic execution to probe through the code under test to create a test suite covering most branches of execution.

While exploring the code, Pex produces a unit test suite that contains tests that cover all branches that Pex can exercise.

A PUT is simply a method that takes parameters, calls the code under test and states assertions. A sample PUT looks like this:

```
void AddItem(List<int> list, int item) {
  list.Add(item);
  Assert.True(list[list.Count - 1] == item);
}
```

The PUT concept is derived from a broader terminology called data-driven testing (DDT), which has been used for a long time in traditional unit tests to make the tests repeatable. Because traditional unit tests are closed in nature, the only way to provide input values to them is through external sources such as an XML file, spreadsheets, or a database. While the DDT approach works well, there is additional overhead involved in maintaining and changing the external data file, and the inputs are dependent on the developer's knowledge about the system.

Pex does not rely on external sources, and instead provides inputs to the test method by passing values to the corresponding PUT. Because the PUT is an open method, it can be arranged to accept any number of inputs. Moreover, Pex does not generate randomized values for the PUT. It relies on introspection of the method under test and generates meaningful values based on factors like boundary conditions, acceptable type values and a state-of-the-art constraint solver called Z3 (research.microsoft.com/um/redmond/projects/z3/). This ensures that all relevant paths of the method under test are covered.

The beauty of Pex is that it generates traditional unit tests from the PUT. These unit tests can be run directly in a unit testing framework like MSTest in Visual Studio without any modifications. Pex provides extensions to generate unit tests for frameworks like NUnit or xUnit.NET. You can also create your own custom extension. A Pex generated traditional unit test looks like this:

```
[TestMethod]
[PexGeneratedBy(typeof(TestClass))]
void AddItem01() {
  AddItem(new List<int>(), 0);
}
```

Dynamic symbolic execution is Pex's answer to exploratory testing. Using this technique, Pex executes the code multiple times to understand the program behavior. It monitors the control and data flow and builds a constraint system for the test inputs.

Unit Testing with Pex

The first step is to create a PUT for the code under test. The PUT can be generated manually by the developer or by using the Visual Studio add-in for Pex. You can tailor the PUT by modifying parameters, adding Pex factories and stubs, integrating with mocks, adding assertions and so on. Pex factories and stubs are covered later in this article.

Currently the Visual Studio add-in for Pex creates PUTs only in C#, but the code under test can be in any .NET language.

The second step after the PUT has been set up is to run the Pex exploration. This is where Pex does its magic. It analyzes the PUT to identify what is being tested. It then starts the inspection of the code under test by passing through each branch and evaluating the possible input values. Pex repeatedly executes the code under test. After each run, it picks a branch that was not covered previously, builds a constraint system (a predicate over the test inputs) to reach that branch, then uses a constraint solver to determine new test inputs, if any. The test is executed again with the new inputs, and this process repeats.

On each run, Pex might discover new code and dig deeper into the implementation. In this way, Pex explores the behavior of the code.

While exploring the code, Pex produces a unit test suite that contains tests that cover all branches that Pex can exercise. These tests are standard unit test that can run in Visual Studio Test Editor. If you find a lower coverage for some sections, you might think of revising your code by refactoring and then applying the same cycle again to achieve higher code coverage and a more comprehensive test suite.

Pex automatically produces a small test suite with high code and assertion coverage.

Pex can help in reducing the amount of effort and time involved when dealing with legacy code. Because Pex automatically explores the different branches and code paths, you don't have to understand all details of the entire code base. Another benefit is that the developer works at the PUT level. Writing a PUT is often much simpler than writing closed unit tests because you focus on the problem scenario rather than on all possible test cases for the functionality.

Putting Pex to Work

Let's use Pex on a piece of legacy code and see how it can help make the code more maintainable and easy to test.

Fabrikam, a leading manufacturer of basketballs and baseballs, provides an online portal where the user can view available products and place orders for those products. The inventory details are in a custom data store accessed through a Warehouse component that provides connectivity to the data store and operations like HasInventory and Remove. An Order component provides a Fill method that processes an order based on the product and quantity passed by the user.

The Order and WareHouse components are tightly coupled to each other. These components were developed years ago and no current employees have a thorough understanding of the system. No unit tests were created during development and, probably as a result, the components are very unstable. The current design is shown in **Figure 1**.

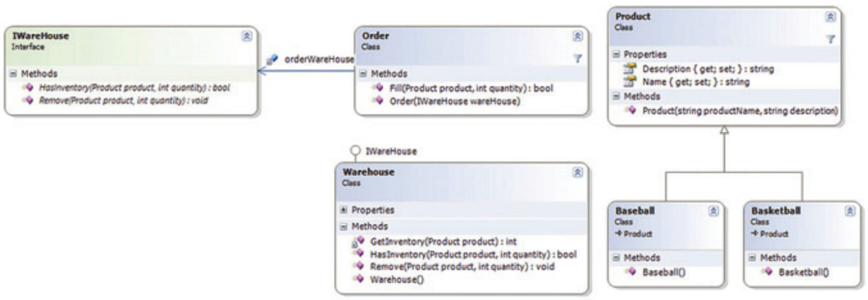

IWareHouse
Interface
Methods
HasInventory(Product product, int quantity) : bool
Remove(Product product, int quantity) : void
orderWareHouse
Order
Class
Methods
Fill(Product product, int quantity) : bool
Order(IWareHouse wareHouse)
Product
Class
Properties
Description { get; set; } : string
Name { get; set; } : string
Methods
Product(string productName, string description)
IWareHouse
Warehouse
Class
Properties
Methods
GetInventory(Product product) : int
HasInventory(Product product, int quantity) : bool
Remove(Product product, int quantity) : void
Warehouse()
Baseball
Class
Product
Methods
Baseball()
Basketball
Class
Product
Methods
Basketball()

Figure 2 **Revised System with IWareHouse**

The Fill method of the Order class looks like this:

```
public class Order {
  public bool Fill(Product product, int quantity) {
    // Check if WareHouse has any inventory
    Warehouse wareHouse = new Warehouse();
    if (wareHouse.HasInventory(product, quantity)) {
      // Subtract the quantity from the product in the warehouse
      wareHouse.Remove(product, quantity);
      return true;
    }

    return false;
  }
}
```

There are a few key items that require attention here. First, Order and Warehouse are tightly coupled. The classes depend on implementations that make them less extensible and difficult to use mock or stub frameworks. There are no unit tests available so any change might introduce regressions resulting in an unstable system.

The Warehouse component was written long ago and the current development team has no knowledge of how to change it or the implications of any changes. To make matters more complicated, Order cannot work with other implementations of Warehouse without modifications.

Let's try to refactor the code and then use Pex to generate unit tests. I will refactor the Warehouse and Order objects, and then create unit tests for the Fill method of the Order class.

Refactoring legacy code is obviously a challenge. An approach in these cases might be to at least make the code testable so that

Figure 3 **Revised Order Class**

```
public class Order {
  readonly IWareHouse orderWareHouse;

  // Use constructor injection to provide a wareHouse object
  public Order(IWareHouse wareHouse) {
    this.orderWareHouse = wareHouse;
  }

  public bool Fill(Product product, int quantity) {
    // Check if WareHouse has any inventory
    if (this.orderWareHouse.HasInventory(product, quantity)) {
      // Update the quantity for the product
      this.orderWareHouse.Remove(product, quantity);
      return true;
    }
    return false;
  }
}
```

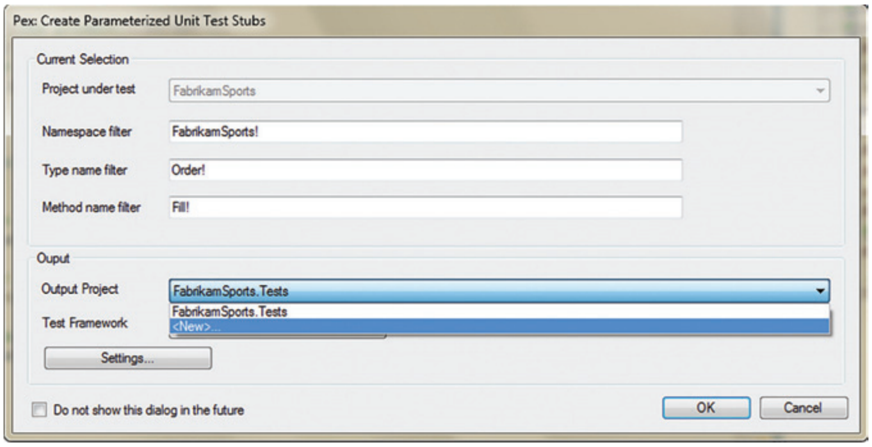
sufficient unit tests can be generated. I will be applying only the bare minimal patterns that make the code testable.

The first problem here is that Order uses a specific implementation of Warehouse. This makes it difficult to decouple the Warehouse implementation from the Order. Let's modify the code a bit to make it more flexible and testable.

I start by creating an interface IWareHouse and modify the Warehouse object to implement this interface. Any new Warehouse will require this interface to be implemented.

Because Order has a direct dependency on Warehouse, they are tightly coupled. I use dependency injection to open the class for extension of its behavior. Using this approach, an IWareHouse instance will be passed to Order at runtime. The new design is shown in **Figure 2**.

The new Order class is shown in **Figure 3**.


Pex: Create Parameterized Unit Test Stubs

Current Selection
- Project under test: FabrikamSports
- Namespace filter: FabrikamSports!
- Type name filter: Order!
- Method name filter: Fill!

Ouput
- Output Project: FabrikamSports.Tests
  - FabrikamSports.Tests
  - <New>...
- Test Framework

Figure 4 **Setting up a New Pex Project**

Creating Parameterized Unit Tests

Let's now use Pex to generate tests for the refactored code. Pex provides a Visual Studio add-in that makes generating PUTs easy. Right-click the project, class, or method for which the PUTs need to be generated and click Pex | Create Parameterized Unit Test Stubs. I started by selecting the Fill method of the Order class.

Pex allows you to select an existing unit test project or to create a new one. It also gives you options to filter tests based on method or type name (see **Figure 4**).

Pex generates the following PUT for the Fill method.

```
[PexClass(typeof(Order))]
[TestClass]
public partial class OrderTest {
  [PexMethod]
  public bool Fill([PexAssumeUnderTest] Order target,
    Product product, int quantity) {

    // Create product factory for Product
    bool result = target.Fill(product, quantity);
    return result;
  }
}
```

The OrderTest is not just a normal TestClass; it has been annotated with a PexClass attribute, indicating that it was created by Pex. There is currently no TestMethod as you would expect in a standard Visual Studio unit test. Instead, you have a PexMethod. This method is a parameterized unit test. Later, when you let Pex explore the code under test, it will create another partial class that contains the standard unit tests annotated with the TestMethod attributes. These generated tests will be accessible through the Visual Studio Test Editor.

Notice that the PUT for the Fill method takes three parameters.

`[PexAssumeUnderTest] Order target`

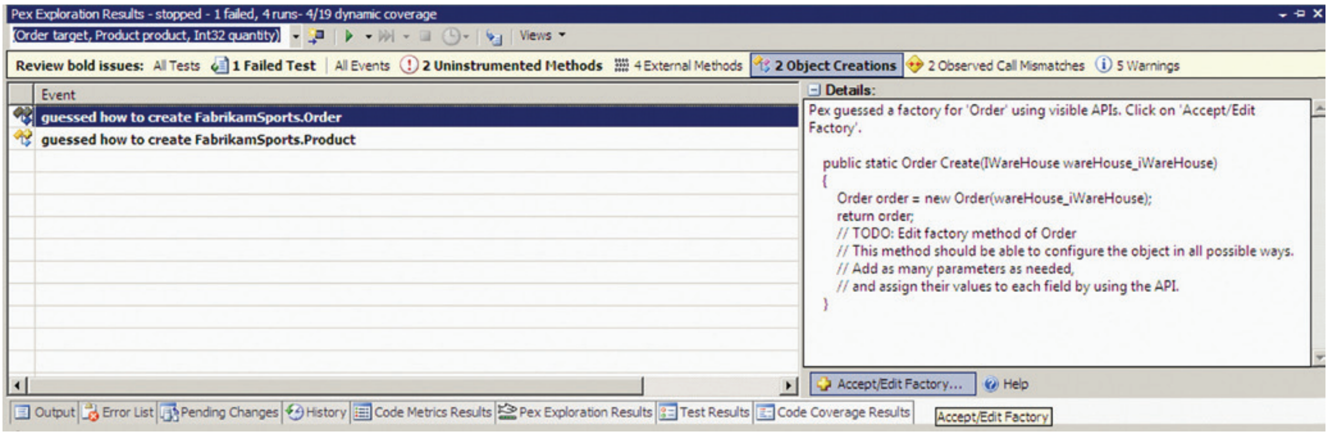
This is the class under test itself. The PexAssumeUnderTest attribute tells Pex that it should only pass non-null values of the exact specified type.

`Product product`

This is the Product base class. Pex will try to create instances of the product class automatically. For more granular control you can provide Pex with factory methods. Pex will use these factories to create instances of complex classes.

`int quantity`

Pex will provide values for the quantity based on the method under test. It will try to inject values that are meaningful for the test rather than junk values.


Pex Exploration Results - stopped - 1 failed, 4 runs- 4/19 dynamic coverage

(Order target, Product product, Int32 quantity)

Review bold issues: All Tests | 1 Failed Test | All Events | 2 Uninstrumented Methods | 4 External Methods | 2 Object Creations | 2 Observed Call Mismatches | 5 Warnings

Event
- guessed how to create FabrikamSports.Order
- guessed how to create FabrikamSports.Product

Details:

Pex guessed a factory for 'Order' using visible APIs. Click on 'Accept/Edit Factory'.

```
public static Order Create(IWareHouse wareHouse_iWareHouse)
{
  Order order = new Order(wareHouse_iWareHouse);
  return order;
  // TODO: Edit factory method of Order
  // This method should be able to configure the object in all possible ways.
  // Add as many parameters as needed,
  // and assign their values to each field by using the API.
}
```

Figure 5 **Creating the Default Factory**

Pex Factories

As I mentioned before, Pex uses a constraint solver to determine new test inputs for parameters. The inputs can be standard .NET types or custom business entities. During exploration, Pex actually creates instances of these types so that the program under test can behave in different interesting ways. If the class is visible and has a visible default constructor, Pex can create an instance of the class. If all the fields are visible, it can generate values for them as well. However, if the fields are encapsulated with properties or not exposed to the outside world, Pex requires help to create the object so as to achieve a better code coverage.

Pex provides two hooks to create and link required objects to the Pex exploration. The user can provide factories for complex objects so that Pex can explore different object states. This is achieved through Pex Factory method. The types that can be created through such factories are called explorable types. We will be using this approach in this article.

The other approach is to define the invariants of the objects private fields so Pex can manufacture different object states directly.

Coming back to the example scenario, if you run a Pex exploration for the generated parameterized tests, the Pex Exploration Results window will show a message "2 Object Creations." This is not an error. During exploration, Pex encountered a complex class (Order in this case) and created a default factory for that class. This factory was required by Pex to understand the program behavior better.

The default factory created by Pex is a vanilla implementation of the required class. You can tailor this factory to provide your own custom implementation. Click on Accept/Edit Factory to inject the code into your project (see **Figure 5**). Alternatively, you can create a static class with a static method that is annotated with the PexFactoryMethod attribute. While exploring, Pex will search in the test project for any static classes with methods with this attribute and use them accordingly.

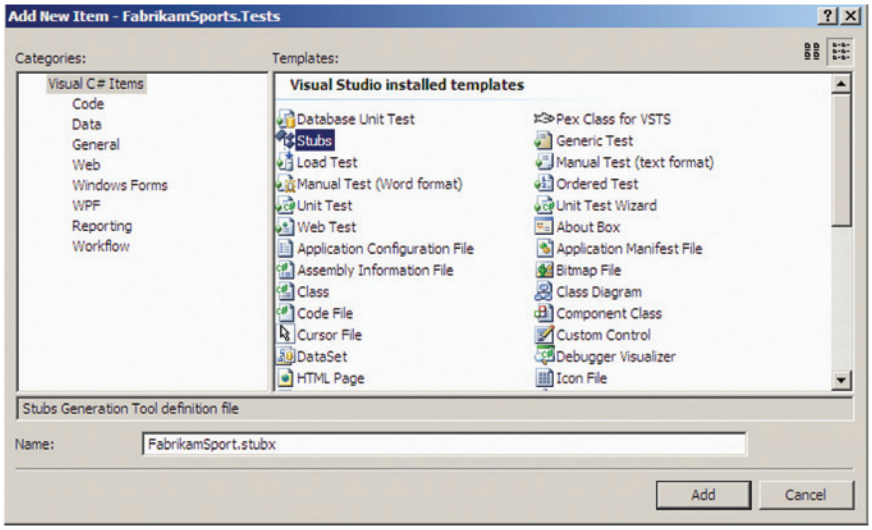
OrderFactory looks like this:

```
public static partial class OrderFactory {
  [PexFactoryMethod(typeof(Order))]
  public static Order Create(IWareHouse wareHouseIWareHouse) {

    Order order = new Order(wareHouseIWareHouse);
    return order;
  }
}
```

If you write factory methods in other assemblies, you can tell Pex to use them in a declarative fashion by using the assembly level PexExplorableFromFactoriesFromType or PexExplorableFromFactoriesFromAssembly attributes, for example.

```
[assembly: PexExplorableFromFactoriesFromType(
  typeof(MyTypeInAnotherAssemblyContainingFactories))]
```

If Pex creates very few tests or fails to create interesting tests by just throwing a NullReferenceException on the object it should create, then this is a good indication that Pex might require a custom factory. Otherwise, Pex comes with a set of heuristics to create object factories that work in many cases.



Stubs Generation Tool definition file

Name: FabrikamSport.stubx

Figure 6 **Creating a New Stub**

Pex Stubs Framework

In software development, the notion of a test stub refers to a dummy implementation that can replace a possibly complex component to facilitate testing. While the idea of stubs is simple, most existing frameworks that can help to create and maintain dummy implementations are actually quite complex. The Pex team has developed a new lightweight framework, which they simply call Stubs. Stubs generates stub types for .NET interfaces and non-sealed classes.

In this framework, a stub of type T provides a default implementation of each abstract member of T, and a mechanism to dynamically specify a custom implementation of each member. (Optionally, stubs can also be generated for non-abstract virtual members.) The stub

Figure 7 **Pex-Generated IWareHouse Stub**

```
/// <summary>Stub of method System.Boolean
/// FabrikamSports.IWareHouse.HasInventory(
/// FabrikamSports.Product product, System.Int32 quantity)
/// </summary>
[System.Diagnostics.DebuggerHidden]
bool FabrikamSports.IWareHouse.HasInventory(
  FabrikamSports.Product product, int quantity) {

  StubDelegates.Func<FabrikamSports.Product, int, bool> sh
    = this.HasInventory;
  if (sh != (StubDelegates.Func<FabrikamSports.Product,
    int, bool>)null)
    return sh.Invoke(product, quantity);
  else {
    var stub = base.FallbackBehavior;
    return stub.Result<FabrikamSports.Stubs.SIWareHouse,
      bool>(this);
  }
}

/// <summary>Stub of method System.Boolean
/// FabrikamSports.IWareHouse.HasInventory(
/// FabrikamSports.Product product, System.Int32 quantity)
/// </summary>
public StubDelegates.Func<FabrikamSports.Product, int, bool>
HasInventory;
```

types are generated as C# code. The framework relies solely on delegates to dynamically specify the behavior of stub members. Stubs supports the .NET Framework 2.0 and higher and integrates with Visual Studio 2008 and higher.

In my example scenario the Order type has a dependency on the Warehouse object. Remember, I refactored the code to implement dependency injection so that Warehouse access can be provided from outside to the Order type. That comes in handy when creating the stubs.

Creating a stub is fairly simple. All you need is a .stubx file. If you created the test project through Pex, you should already have it. If not, this file can be created from within Visual Studio. Right-click on the test project and select Add New Item. A Stubs template is available (see **Figure 6**).

The file appears as a standard XML file in Visual Studio. In the assembly element, specify the name of the assembly for which the stubs need to be created and save the .stubx file.

```
<Stubs xmlns="http://schemas.microsoft.com/stubs/2008/">
  <Assembly Name="FabrikamSports" />
</Stubs>
```

Pex will automatically create the necessary stub methods for all the types in the assembly.

The generated stub methods have corresponding delegate fields that provide hooks for stubbed implementations. By default, Pex will provide an implementation for the delegate. You can also

A test stub refers to dummy implementation that can replace a possibly complex component to facilitate testing.

provide a lambda expression to attach behavior to the delegate or use the PexChoose type to let Pex automatically generate values for the method.

For example, to provide choices for the HasInventory method, I can have something like this:

```
var wareHouse = new SIWareHouse() {
  HasInventoryProductInt32 = (p, q) => {
    Assert.IsNotNull(p);
    Assert.IsTrue(q > 0);
    return products.GetItem(p) >= q;
  }
};
```

In fact, using PexChoose is already the default behavior for stubs when using Pex, as the test project created by Pex contains the following assembly-level attribute:

```
[assembly: PexChooseAsStubFallbackBehavior]
```

The SIWareHouse type is generated by the Pex Stubs framework. It implements the IWareHouse interface. Let's take a closer look at the code generated by Pex for the SIWareHouse stub. The source code for a .stubx file is created in a partial class with the name <StubsxFilename>.designer.cs as shown in **Figure 7**.

Figure 8 **Parameterized Model of IWareHouse**

```
public sealed class PWareHouse : IWareHouse {
  PexChosenIndexedValue<Product, int> products;

  public PWareHouse() {
    this.products =
      new PexChosenIndexedValue<Product, int>(
      this, "Products", quantity => quantity >= 0);
  }

  public bool HasInventory(Product product, int quantity) {
    int availableQuantity = this.products.GetItem(product);
    return quantity - availableQuantity > 0;
  }

  public void Remove(Product product, int quantity) {
    int availableQuantity =
    this.products.GetItem(product);
    this.products.SetItem(product,
      availableQuantity - quantity);
  }
}
```

Stubs created a public delegate field for the HasInventory method and invoked it in the HasInventory implementation. If no implementation is available, Pex calls the FallBackBehaviour.Result method, which would use PexChoose if the [assembly: PexChooseAsStubFallbackBehavior] is present, and throws a StubNotImplementedException otherwise.

To use the stubbed implementation of IWareHouse, I will tweak the Parameterized unit test a bit. I already modified the Order class to be able to take an IWareHouse implementation in its constructor. I now create a SIWareHouse instance, then pass that to the Order class so it uses the custom implementations of the IWareHouse methods. The revised PUT is shown here:

```
[PexMethod]
public bool Fill(Product product, int quantity) {
  // Customize the default implementation of SIWareHouse
  var wareHouse = new SIWareHouse() {
    HasInventoryProductInt32 = (p, q) =>
    PexChoose.FromCall(this).ChooseValue<bool>(
    "return value")
  };

  var target = new Order(wareHouse);
  // act
  bool result = target.Fill(product, quantity);
  return result;
}
```

Figure 9 **Using Mole Types on Fill**

```
[PexMethod]
public bool Fill([PexAssumeUnderTest]Order target,
  Product product, int quantity) {

  var products = new PexChosenIndexedValue<Product, int>(
    this, "products");
  // Attach a mole of WareHouse type
  var wareHouse = new MWarehouse {
    HasInventoryProductInt32 = (p, q) => {
      Assert.IsNotNull(p);
      return products.GetItem(p) >= q;
    }
  };

  // Run the fill method for the lifetime of the mole
  // so it uses MWareHouse
  bool result = target.Fill(product, quantity);
  return result;
}
```

Figure 10 **Pex-Generated Tests**

```
[TestMethod]
[PexGeneratedBy(typeof(OrderTest))]
public void Fill15()
{
    Warehouse warehouse;
    Order order;
    Product product;
    bool b;
    warehouse = new Warehouse();
    order = OrderFactory.Create((IWareHouse)warehouse);
    product = new Product("Base ball", (string)null);
    b = this.Fill(order, product, 0);
    Assert.AreEqual<bool>(true, b);
}
[TestMethod]
[PexGeneratedBy(typeof(OrderTest))]
public void Fill16()
{
    Warehouse warehouse;
    Order order;
    Product product;
    bool b;
    warehouse = new Warehouse();
    order = OrderFactory.Create((IWareHouse)warehouse);
    product = new Product("Basket Ball", (string)null);
    b = this.Fill(order, product, 0);
    Assert.AreEqual<bool>(true, b);
}
[TestMethod]
[PexGeneratedBy(typeof(OrderTest))]
public void Fill17()
{
    Warehouse warehouse;
    Order order;
    Product product;
    bool b;
    warehouse = new Warehouse();
    order = OrderFactory.Create((IWareHouse)warehouse);
    product = new Product((string)null, (string)null);
    b = this.Fill(order, product, 1);
    Assert.AreEqual<bool>(false, b);
}
```

Stubs actually automatically provides default implementations for the stub methods, so you could have simply run the PUT without any modifications as well.

Parameterized Models

For a more granular control of the stubbed implementation, Pex supports a concept called a parameterized model. This is a way to write stubs that do not have one particular fixed behavior. The abstraction that Pex provides through this concept is that the developer does not need to worry about the variations of the implementation. Pex will explore different return values of the method based on how they are used by the code under test. Parameterized models are a powerful feature that allows you to take complete control over how the stubs should be processed while at the same time letting Pex evaluate the variant values for the input parameters.

A parameterized model for IWareHouse might look like the code in **Figure 8**.

Essentially I have created my own stubbed implementation for IWareHouse, but notice that I do not provide values for Quantity and Product. Instead I let Pex generate those values. PexChosenIndexedValue automatically provides the values for the objects, allowing only one stubbed implementation with variant parameter values.

For simplicity I will let Pex provide the HasInventory implementation of the IWareHouse type. I will add code to the OrderFactory class that I created earlier. Every time an Order instance is created by Pex it will use a stubbed Warehouse instance.

Moles

So far I've focused on two principles—refactor your code to make it testable, then use Pex to generate unit tests. This approach lets you clean your code, eventually leading to more maintainable software. However, refactoring legacy code can be a big challenge in itself. There can be numerous organizational or technical constraints that might prevent a developer from refactoring the current source code. How do you deal with this?

In scenarios where legacy code is hard to refactor, the approach should be to at least create sufficient unit tests for the business logic so that you can verify the robustness of each module of the system. Mock frameworks like TypeMock (learn.typemock.com) have been around for a while. They let you create unit tests without actually modifying the code base. This approach proves very beneficial specifically for large legacy codebases.

Pex comes with a feature called Moles that lets you achieve the same goals. It allows you to generate Pex unit tests for legacy code without actually refactoring your source code. Moles are really meant to test otherwise untestable parts of the system, such as static methods and sealed classes.

Moles work in a similar fashion to stubs: Pex code-generates mole types that expose delegate properties for each method. You can attach a delegate, and then attach a mole. At that point, all your custom delegates get wired up magically by the Pex profiler.

Pex automatically creates moles for all static, sealed and public interfaces as specified in the .stubx file. A Pex Mole looks very similar to a Stubs type (see the code download for an example).

Using moles is fairly simple. You can provide implementations for the Mole methods and use them in your PUT. Notice that because Mole injects the stubbed implementations during runtime, you do not have to change your codebase at all to be able to generate unit tests using moles.

Let's use moles on the legacy Fill method

```
public class Order {
  public bool Fill(Product product, int quantity) {
    // Check if warehouse has any inventory
    Warehouse wareHouse = new Warehouse();
    if (wareHouse.HasInventory(product, quantity)) {
      // Subtract the quantity from the product
      // in the warehouse
      wareHouse.Remove(product, quantity);
      return true;
    }

    return false;
  }
}
```

I create a PUT for the Fill method that leverages the Mole types (see **Figure 9**).

MWareHouse is a Mole type that was created automatically by Pex when generating the stubs and moles via the .stubx file. I provide a custom implementation for the HasInventory delegate of the MWareHouse type, and then call the Fill method. Notice that

faster than a speeding bullet,
more powerful than a locomotive,
able to **simplify connectivity** with a single component.

The Professional Developer's #1 Choice for Communications, Security, & E-Business Components

Internet Communications

IP*Works! Products

TCP/IP	**Secure SNMP**
SSL	**Zip**
S/MIME	**EDI AS2**
Secure Shell	

Our flagship product line. The result of more than a decade of research and development in building Internet connectivity tools for professional software developers.

Components for: Web and Web Services, Email and News, File Transfer, Sockets and Streaming, Remote Access, Instant Messaging, SSL and Secure Shell Security, Certificate Management & Creation, S/Mime Encryption, SNMP Network Management, File & Streaming Compression, and E-Business (B2B). Transactions.

Internet Business

IBiz Integrator Products

QuickBooks	**First Data**
E-Payment	**USPS**
E-Banking	**FedEx**
Vital/TSYS	**Amazon**
Paymentech	**PayPal**

The IBiz product line provides small and medium-sized companies with enterprise-class software components for Internet business integration. Built on top of our award winning IP*Works! tools, these easy-to-use components enable developers to rapidly integrate common business processes.

Components for: Accounting Integration (QuickBooks), Credit Card Processing, ACH / E-Check Processing, Shipping and Tracking, Banking &Financial Transactions, and Services Integration.

Enterprise Adapters

BizTalk & SQL Server Adapters

AS2 / EDI-INT	**GISB / NAESB**
SFTP / FTPS	**OFTP**
XMPP (Jabber)	**Secure Shell**
SMS Paging	**Secure Email**
AWS Integration	

The /n software Adapters extend Microsoft BizTalk and SQL Server with advanced Internet communications, security, and e-business capabilities, including robust, highly scalable, fully integrated implementations of B2B messaging and secure communications protocols.

Components for: EDI-INT AS2 Integration, Secure File Transfer, Secure Shell Remote Execution, Secure Email, RosettaNet Connectivity, OFTP Integration, NAESB / GISB Messaging, and more.

WWW.NSOFTWARE.COM - [ASK ABOUT OUR MSDN-STYLE ALL INCLUSIVE RED CARPET SUBSCRIPTIONS]

/n software

nowhere do I provide an implementation of the Warehouse object to the Order type constructors. Pex will attach the MWareHouse instance to the Order type at run time. For the lifetime of the PUT any code written inside the block will leverage the MWareHouse type implementation wherever a Warehouse implementation will be required in the legacy code.

When Pex generates traditional unit tests that use moles, it attaches the attribute [HostType("Pex")] to them, so that they will execute with the Pex profiler, which will allow the moles to become active.

Putting It All Together

I talked about the various features of Pex and how to use them. Now it's time to actually run the PUT and observe the results. To run an exploration for the Fill method of Order, simply right-click on the PUT and select Run Pex Exploration. You can optionally run the exploration on a class or the entire project.

While Pex exploration is running, a partial class is created along with the PUT class file. This partial class contains all the standard unit tests that Pex will generate for the PUT. For the Fill method, Pex generates standard unit tests using various test inputs. The tests are shown in **Figure 10**.

The key point to observe here is the variations for the Product type. Although I did not provide any factory for it, Pex was able to create different variations for the type.

Also note that the generated test contains an assertion. When a PUT returns values, Pex will embed the values that were returned at test generation time into the generated test code as assertions. As a result, the generated test is often able to detect breaking changes in the future, even if they don't violate other assertions in the program code, or cause exceptions at the level of the execution engine.

The Pex exploration results window (see **Figure 11**) provides details of the unit tests generated by Pex. It also provides information about the factories Pex created and the events that occurred during the exploration. Notice in **Figure 11** that two tests failed. Pex shows a NullReferenceException against both of them. This can be a common problem where you miss out placing validation checks in code paths that eventually might lead to exceptions when running in production.

Pex not only generates tests, but also analyzes the code for improvement. It provides a set of suggestions that can make the code even more stable. These suggestions are not just descriptive message, but actual code for the problem area. With a click of a button, Pex injects the code into the actual source file. Select the failed test in the Pex exploration result window. In the bottom-right corner, a button appears titled Add Precondition. Clicking this button adds the code into your source file.

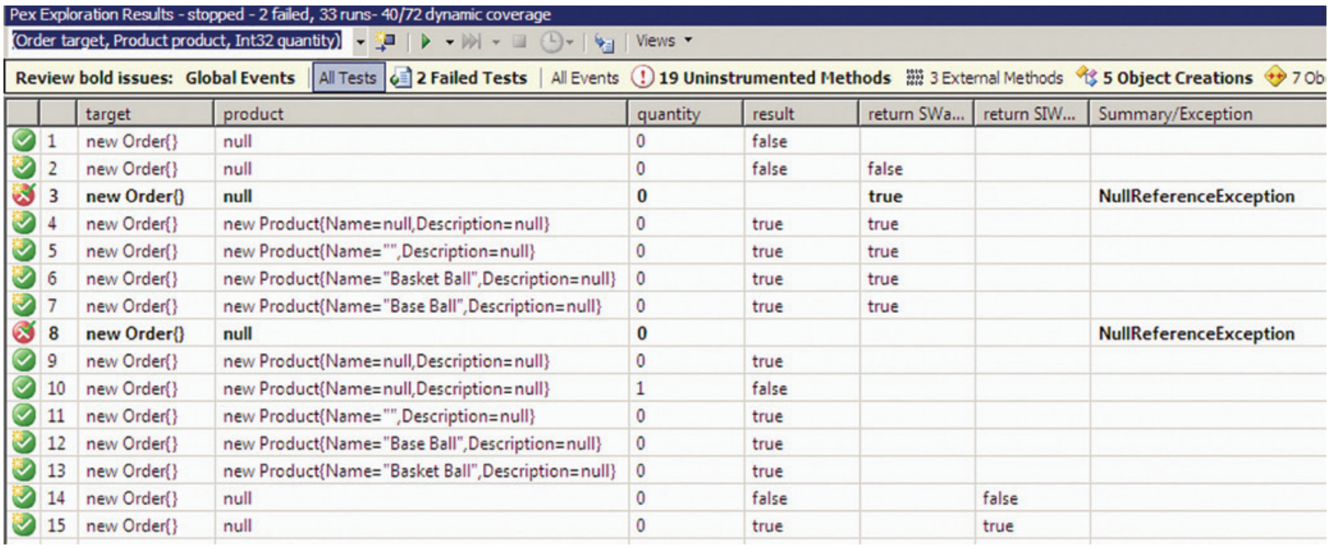
The Moles feature allows you to generate Pex unit tests for legacy code without actually refactoring your source code.

These generated tests are normal MSTest unit tests and can be run from the Visual Studio Test Editor as well. If you open the editor, all the Pex generated tests will be available as standard unit tests. Pex can generate similar tests for other unit testing frameworks like NUnit and xUnit.

Pex also has built-in support for generating coverage reports. These reports provide comprehensive details around the dynamic coverage for the code under test. You can enable reports from Pex options in the Tools menu of Visual Studio, then open them by clicking Views | Report in the Pex menu bar after an exploration has finished.

Making your Tests Future Ready

So far you've seen how Pex was able to generate code coverage for legacy code with minor refactoring of the source code. The beauty of


Pex Exploration Results - stopped - 2 failed, 33 runs- 40/72 dynamic coverage
(Order target, Product product, Int32 quantity)
Views
Review bold issues: Global Events
All Tests
2 Failed Tests
All Events
19 Uninstrumented Methods
3 External Methods
5 Object Creations
7 Ob

		target	product	quantity	result	return SWa...	return SIW...	Summary/Exception
✓	1	new Order()	null	0	false			
✓	2	new Order()	null	0	false	false		
✗	**3**	**new Order()**	**null**	**0**		**true**		**NullReferenceException**
✓	4	new Order()	new Product{Name=null,Description=null}	0	true	true		
✓	5	new Order()	new Product{Name="",Description=null}	0	true	true		
✓	6	new Order()	new Product{Name="Basket Ball",Description=null}	0	true	true		
✓	7	new Order()	new Product{Name="Base Ball",Description=null}	0	true	true		
✗	**8**	**new Order()**	**null**	**0**				**NullReferenceException**
✓	9	new Order()	new Product{Name=null,Description=null}	0	true			
✓	10	new Order()	new Product{Name=null,Description=null}	1	false			
✓	11	new Order()	new Product{Name="",Description=null}	0	true			
✓	12	new Order()	new Product{Name="Base Ball",Description=null}	0	true			
✓	13	new Order()	new Product{Name="Basket Ball",Description=null}	0	true			
✓	14	new Order()	null	0	false		false	
✓	15	new Order()	null	0	true		true	

Figure 11 **Pex Exploration Results**

Pex is that it relieves the developer from writing unit tests and generates them automatically, thereby reducing the overall testing effort.

One of the major pain points while unit testing is maintaining the test suite itself. As you progress in a project, you typically make lot of modifications to existing code. Because the unit tests are dependent on the source code, any code change impacts the corresponding unit tests. They might break or reduce the code coverage. Over a period of time keeping the test suite live becomes a challenge.

Pex comes in handy in this situation. Because Pex is based on an exploratory approach, it can search for any new changes in the code base and create new test cases based on them.

The prime purpose of regression testing is to detect whether modifications or additions to existing code have affected the codebase adversely, either through the introduction of functional bugs or creation of new error conditions.

Pex not only generates tests, but also analyzes the code for improvement.

Pex can generate a regression test suite automatically. When this regression test suite is executed in the future, it will detect breaking changes that cause assertions in the program code to fail, that cause exceptions at the level of the execution engine (NullReferenceException), or that cause assertions embedded in the generated tests to fail. Each time a Pex exploration is run fresh, unit tests are generated for the code under observation. Any change in behavior is picked up by Pex and the corresponding unit tests are generated for it.

Change is Inevitable

Over a period of time, the developer team at Fabrikam realized that it made sense to have a ProductId attribute added to the Product class so that if the company adds new products to their catalog they can be uniquely identified.

Also the Order class was not saving the orders to a data store, so a new private method SaveOrders was added to the Order class. This method will be called by the Fill method when the product has some inventory.

The modified Fill method class looks like this:

```
public bool Fill(Product product, int quantity) {
  if (product == null) {
    throw new ArgumentException();
  }

  if (this.orderWareHouse.HasInventory(product, quantity)) {
    this.SaveOrder(product.ProductId, quantity);

    this.orderWareHouse.Remove(product, quantity);
  return true;
  }

  return false;
}
```

Because the signature of the Fill method was not changed, I do not need to revise the PUT. I simply run the Pex Exploration again.

Figure 12 **Additional Pex-generated Unit Tests**

```
[TestMethod]
[PexGeneratedBy(typeof(OrderTest))]
public void Fill12()
{
    using (PexChooseStubBehavior.NewTest())
    {
      SIWareHouse sIWareHouse;
      Order order;
      Product product;
      bool b;
      sIWareHouse = new SIWareHouse();
      order = OrderFactory.Create((IWareHouse)sIWareHouse);
      product = new Product((string)null, (string)null);
      b = this.Fill(order, product, 0);
      Assert.AreEqual<bool>(false, b);
    }
}

[TestMethod]
[PexGeneratedBy(typeof(OrderTest))]
public void Fill13()
{
    using (PexChooseStubBehavior.NewTest())
    {
      SIWareHouse sIWareHouse;
      Order order;
      Product product;
      bool b;
      sIWareHouse = new SIWareHouse();
      order = OrderFactory.Create((IWareHouse)sIWareHouse);
      product = new Product((string)null, (string)null);
      IPexChoiceRecorder choices = PexChoose.NewTest();
      choices.NextSegment(3)
          .OnCall(0,
      "SIWareHouse.global::FabrikamSports.IWareHouse.
HasInventory(Product, Int32)")
          .Returns((object)true);
      b = this.Fill(order, product, 0);
      Assert.AreEqual<bool>(true, b);
    }
}
```

Pex runs the exploration, but this time it generates inputs using the new Product definition, utilizing the ProductId as well. It generates a fresh test suite, taking into account the changes made to the Fill method. The code coverage comes out to be 100 percent—ensuring that all new and existing code paths have been evaluated.

Additional unit tests are generated by Pex to test the variations of the added ProductId field and the changes made to the Fill method (see **Figure 12**). Here PexChooseStubBehavior sets the fallback behavior for the stubs; instead of just throwing a StubNotImplementedException, the stubbed method will call PexChoose to provide the possible return values. Running the tests in Visual Studio, the code coverage comes out to be 100 percent again.

Acknowledgements

I would like to thank Peli de Halleux and Nikolai Tillman for encouraging me to write this article, special gratitude to Peli for his tireless support, valuable comments and exhaustive review. ■

NIKHIL SACHDEVA *is a software development engineer for the OCTO-SE team at Microsoft. You can contact him at blogs.msdn.com/erudition. You can also post your queries around Pex at social.msdn.microsoft.com/Forums/en/pex/threads.*

THANKS to the following technical experts for reviewing this article: *Peli de Halleux*

Using Agile Techniques to Pay Back Technical Debt

David Laribee

In every codebase, there are the dark corners and alleys you fear. Code that's impossibly brittle; code that bites back with regression bugs; code that when you attempt to follow, will drive you mad.

Ward Cunningham created a beautiful metaphor for the hard-to-change, error-prone parts of code when he likened it to financial debt. Technical debt prevents you from moving forward, from profiting, from staying "in the black." As in the real world, there's cheap debt, debt with an interest lower than you can make in a low-risk financial instrument. Then there's the expensive stuff, the high-interest credit card fees that pile on even more debt.

Technical debt is a drag. It can kill productivity, making maintenance annoying, difficult, or, in some cases, impossible. Beyond the obvious economic downside, there's a real psychological cost to technical debt. No developer enjoys sitting down to his computer in the morning knowing he's about to face impossibly brittle, complicated source code. The frustration and helplessness thus engendered is often a root cause of more systemic problems, such as developer turnover— just one of the real economic costs of technical debt.

Every codebase I've worked on or reviewed contains some measure of technical debt. One class of debt is fairly harmless: byzantine dependencies among bizarrely named types in stable, rarely modified recesses of your system. Another is sloppy code that is easily fixed on the spot, but often ignored in the rush to address higher-priority problems. There are many more examples.

This article outlines a workflow and several tactics for dealing with the high-interest debt. The processes and practices I'll detail aren't new. They are taken straight from the Lean/Agile playbook.

This article discusses:

- The case for fixing debt
- Basic debt-finding workflow
- Prioritizing items as a team
- Selling your plan

Technologies discussed:

Agile

The Case for Fixing Debt

The question, "should we fix technical debt," is a no-brainer in my book. Of course you should. Technical debt works against your goals because it slows you down over time. There's a well-known visualization called the cost of change curve (see **Figure 1**), that illustrates the difference between the 100-percent-quality-test-driven approach and the cowboy-coder-hacking-with-duct-tape approach.

The cost of change curve illustrates that high quality, simple, and easy to follow designs may cost more initially, but incur less technical debt—subsequent additions and modifications to the code are less costly over time. In the quality curve (blue), you can see the initial cost is higher, but it's predictable over time. The hack-it curve (red) gets a lower cost of entry, but future development, maintenance, and the total cost of owning a product and its code becomes ever more expensive.

Ward Cunningham's First Law of Programming" (`c2.com/cgi-bin/wiki?FirstLawOfProgramming`) states, "lowering quality lengthens development time."

"Quality software takes the least amount of time to develop. If you have code that is simple as possible, tests that are complete and a design that fits just right, additions and changes happen in the fastest possible way because the impact is lowest. Consequently, if you hack something out, the more you hack the slower you go because the cost of addition or change grows with each line of code."

Simply put, technical debt will decrease the throughput of your team over time.

One of my great joys in software development is relishing the feeling of raw productivity. The competing and converse feeling, for me at least, is pain. It's painful when I'm not productive and it's pain that robs me of potential productivity, the so-called "good days at work." There are many sources of pain in software development, but none more obvious than a rigid and chaotic codebase. This psychological effect takes a toll on team morale which, in turn, causes productivity to lag.

Systems Thinking

In order to fix technical debt, you need to cultivate buy-in from stakeholders and teammates alike. To do this, you need to start thinking systemically. Systems thinking is long-range thinking. It is investment thinking. It's the idea that effort you put in today will let you progress at a predictable and sustained pace in the future.

Perhaps it's easiest to explain systems thinking with an analogy. I live in downtown Atlanta, Georgia, in a quaint little neighborhood called Inman Park. I'm mostly very happy there. I reserve, however, some irritation related to the seemingly complete ignorance of urban planning. The streets in Atlanta are byzantine, maze-like, madness-provoking. When you miss your turn, you can't simply loop back. If you do, you'll be sent on a spiraling path to who-knows-where. There seems to be little rhyme or reason to the planning of roads in this otherwise very pleasant corner of the world.

Contrast this with the orderly streets and avenues of Manhattan in New York City (most of it, anyway). It's as if a Marine Corps drill instructor designed the city. Avenues stretch the length of the island, north to south, and streets form tidy, latitudinal markers down its length. Furthermore, both streets and avenues are named in numerical sequence: First Avenue, Second Avenue, 42nd Street, 43rd Street, and so on. You'll rarely walk more than a block in the wrong direction.

Technical debt is a drag. It can kill productivity, making maintenance annoying, difficult or, in some cases, impossible.

What are the root causes for the difference between Atlanta and New York City in this dimension of comparison?

In Atlanta the streets were formed by cattle wearing down paths. You heard me right, *cattle paths*. Some need arose to frequently move between the urban center and the suburbs, at which point some cowboy thought, "golly, wouldn't it be easiest to turn these here cattle paths into roads?"

The New York State Legislature applied vision and forethought to the design of the ever-growing and largest city in the state. They chose the gridiron plan, with orderly, predictable streets and avenues. They were thinking of the future.

This story gets to the essence of systems thinking. While legislative processes are slow, investment in time and commitment to vision pays the greatest dividend for the *lifetime of a system*. True you'll have to deal with crazy cabs on the mean streets of Manhattan, but you'll be able to find your way around in a matter of days. In Atlanta, it's been a year of getting lost, and I thank the system thinkers responsible for the Global Positioning System (GPS) each and every day.

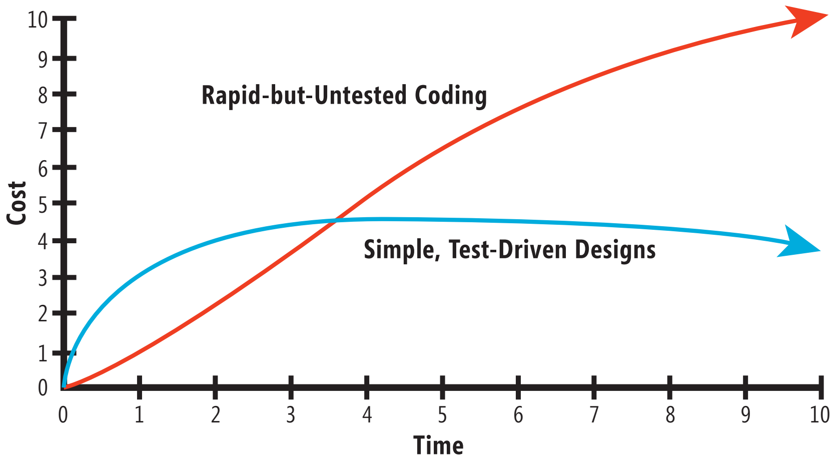


Figure 1 **Cost of Change Curve**

Products over Projects

The idea that you have a development team that completes a project then throws it over the wall to a maintenance team is fundamentally flawed. Make no mistakes, you are working on a product, and if it succeeds, it's going to live a long, long time.

If you have even a couple of years of experience as a professional developer, you've probably experienced the increasing gravity effect. You develop a piece of software that isn't meant to last or be complicated or change. And six months later, what are you doing? Modifying it? Extending it? Fixing bugs?

Useful software has a sometimes nasty habit of sticking around for a very long time. It's up to you to pick the metaphor you want to roll with. Will you tend a forest of beautiful California Redwoods, living entities enduring the centuries and reaching the highest heights, or will you allow the relentless Kudzu vine to starve your forest of light?

Basic Workflow

At this point, I hope I've convinced you that technical debt can take an awful toll on both your mental health and your customer's bottom line. I hope, also, that you accept the need to take a longer-range view on the products you're creating.

Now let's figure out how you can dig yourself out of this hole.

No matter the shop, my sense is that the basic workflow for tackling technical debt—indeed any kind of improvement—is repeatable. Essentially, you want to do four things:

1. Identify where you have debt. How much is each debt item affecting your company's bottom line and team's productivity?
2. Build a business case and forge a consensus on priority with those affected by the debt, both team and stakeholders.
3. Fix the debt you've chosen head on with proven tactics.
4. Repeat. Go back to step 1 to identify additional debt and hold the line on the improvements you've made.

It's worth mentioning, for the software process nerds out there, that this workflow is adapted from a business management approach called the Theory of Constraints (ToC) created by Eliyahu Goldratt (goldrattconsulting.com). ToC is a systems-thinking model that provides a framework for improving the overall throughput of the system. This is a gross simplification, but ToC is predicated on the idea that a system (a manufacturing facility, for example) is only as productive as its biggest bottleneck. Value, such as a feature request or automobile or any sellable item, is conceived of, designed, produced, and deployed. A feature may be requested by a customer, internal or external, and that feature flows through your business (the system), transforming from an idea to a tangible result. What happens when these features pile up in front of your quality assurance team? What happens when there's more demand for development than a development team can fulfill? You get a bottleneck and the whole system slows down.

There are many sources of pain in software development, but none more obvious than a rigid and chaotic codebase.

It's very likely that you have many areas of debt—many bottlenecks—in your codebase. Finding the debt that slows you down the most will have the greatest net effect on increasing your throughput. Understanding, then tackling debt and resulting improvements as a team—as a system—is the most effective way to make positive change, because more eyes and hands on the code equates to less risk and better designs.

Identify Areas of Debt

It's important that you be able to point at the problem areas. If you haven't been keeping track of them on a wiki or a shared list or in code comments, your first task is to find the debt.

If you're working on a team, I suggest calling a meeting to develop a concrete list of the top areas of debt in your code. An exhaustive list isn't important. Focus on capturing the big-ticket items. This meeting is your first opportunity, as a leader on your team, to start forging consensus. A more-than-simple majority of members should agree and understand an item for it to make the list.

Once you have the list, make it durable. Create a wiki topic, write it on a whiteboard (with "DO NOT ERASE" written prominently in one corner), or whatever works in your situation. The medium you choose should be visible, permanent and easy to use. It should be in your face on a regular basis. You need to return to this list and groom it. Human beings have a limited amount of short-term memory, so I suggest keeping a list of between five and nine of the most bothersome items. Don't worry so much about keeping an inventory—important items will surface again if they're really, well, important.

Using Metrics to Find Trouble Areas

Sometimes it's hard to find debt, especially if a team is new to a codebase. In cases where there's no collective memory or oral tradition to draw on, you can use a static analysis tool such as NDepend (ndepend.com) to probe the code for the more troublesome spots.

Tools are, at best, assistive or perhaps even a second choice. Tools won't tell you what to do. They will, however, give you inputs to decision-making. There is no single metric for code debt, but people who work on a product day in and day out can surely point to those dark corners that cause the most pain. Static analysis tools will tell you where you have implementation debt. Sadly, they will not tell you where you have debt due to factors like poor naming, discoverability, performance, and other more qualitative design and architectural considerations.

Knowing your test coverage (if you have tests) can be another valuable tool for discovering hidden debt. Clearly, if there's a big part of your system that lacks solid test coverage, how can you be certain that a change won't have dramatic effects on the quality of your next release? Regression bugs are likely to appear, creating bottlenecks for QA and potential embarrassment and loss of revenue due to customer-found defects.

Use the log feature of your version control system to generate a report of changes over the last month or two. Find the parts of your system that receive the most activity, changes or additions, and scrutinize them for technical debt. This will help you find the bottlenecks that are challenging you today; there's very little value in fixing debt in those parts of your system that change rarely.

Human Bottlenecks

You might have a bottleneck if there's only one developer capable of dealing with a component, subsystem, or whole application.

Individual code ownership and knowledge silos (where "Dave works on the Accounts Receivables module"—now there's a painful memory), can block delivery if that person leaves the team or has a pile of other work to do. Finding places in your project where individual ownership is happening lets you consider the benefits and scope of improving the design so other individuals can share the load. Eliminate the bottleneck.

Finding the debt that slows you down the most will have the greatest net effect on increasing your throughput.

There are tremendous benefits that derive from the eXtreme Programming practice of collective ownership (extremeprogramming.org/rules/collective.html). With collective ownership, any developer on your team is allowed to change any code in your codebase "to add functionality, fix bugs, improve designs or refactor. No one person becomes a bottleneck for changes."

Ah! There's that word again, "bottleneck." By enabling collective ownership, you eliminate the dark parts of your system that only a single programmer—who may walk off the job or get hit by a bus—knows about. There is less risk with a codebase that's collectively owned.

In my experience, the design is also much better. Two, three, or four heads are almost certainly better than one. In a collectively owned codebase, a team design ethos emerges and supplants individual idiosyncrasies and quirks.

I call collective code ownership a practice, but collective ownership is really an emergent property of a well-functioning team. Think about it—how many of you show up and work on "your code" versus code shared by an entire team? What are often called teams in software development are really workgroups with an assignment editor where programming tasks are doled out based on who's worked on a particular feature, subsystem or module in the past.

Prioritize as a Team

I've said before that it's important you involve the whole team in efforts to improve. As an Agile Coach, I hold closely to the mantra that *people support a world they help to create*. If you don't have a critical mass of support, an effort to foster a culture of continuous improvement can be very difficult to get off the ground, much less sustain.

Obtaining consensus is key. You want the majority of team members to support the current improvement initiative you've selected. I've used with some success Luke Hohmann's "Buy a Feature" approach from his book *Innovation Games* (innovationgames.com). I'll attempt a gross over-simplification of the game, and urge you to check out the book if it seems like something that'll work in your environment.

1. Generate a short list (5-9 items) of things you want to improve. Ideally these items are in your short-term path.
2. Qualify the items in terms of difficulty. I like to use the abstract notion of a T-shirt size: small, medium, large or extra-large (see the Estimating Improvement Opportunities sidebar for more information on this practice).
3. Give your features a price based on their size. For example, small items may cost $50, medium items $100, and so on.
4. Give everyone a certain amount of money. They key here is to introduce scarcity into the game. You want people to have to pool their money to buy the features they're interested in. You want to price, say, medium features at a cost where no one individual can buy them. It's valuable to find where more than a single individual sees the priority since you're trying to build consensus.
5. Run a short game, perhaps 20 or 30 minutes in length, where people can discuss, collude, and pitch their case. This can be quite chaotic and also quite fun, and you'll see where the seats of influence are in your team.
6. Review the items that were bought and by what margins they were bought. You can choose to rank your list by the purchased features or, better yet, use the results of the Buy a Feature game in combination with other techniques, such as an awareness of the next release plan.

Sell the Plan

Now that you've got a plan, it's time to communicate the value of eliminating debt to your project sponsors. In reality, this step can happen in parallel with identification. Involve your customer from the very beginning. After all, development of the plan is going to take time, effort, and (ultimately) money. You want to

Estimating Improvement Opportunities

I mentioned estimating debt items or improvement opportunities roughly in terms of T-shirt sizes. This is a common technique used in Agile development methodologies. The idea is that you're collecting things in terms of relative size. The smalls go together as do the mediums, larges, and so on.

It's not super-important that you bring a lot of accuracy to the table here. Remember: these are relative measures and not commitments. You want to get a rough idea of the difficulty, and the theory is that after estimating a number of items, things will start to even out. Even though one medium item actually takes a pair of developers two weeks to complete while another takes a month, on average a medium will take about three weeks.

Over time, however, you'll start to gather good examples of what a large or small item really is, and this will aid you in future estimates because you'll have a basis of comparison. I've used several examples of the various sizes in the past as an aid for estimating a new batch of work to good effect.

This can be a tough pill to swallow for management. They'll initially want to know exactly how long a thing might take and, truth be told, you might need to invest more time in a precise, time-based estimate.

XCEED

DataGrid

for WPF

The most advanced WPF data presentation control

Microsoft®
Visual Studio® Team System 2010

"Using Xceed DataGrid for WPF in Microsoft Visual Studio Team System 2010 helped us greatly reduce the time and resources necessary for developing all the data presentation features we needed. Working with Xceed has been a pleasure."

Norman Guadagno
Director of Product Marketing
for Microsoft Visual Studio Team System

IBM®
U2 SystemBuilder™

"IBM U2 researched existing third party solutions and identified Xceed DataGrid for WPF as the most suitable tool. The datagrid's performance, solid foundation, takes true advantage of WPF and provides the extensibility required for IBM U2 customers to take their applications to the next level in presentation design and styling."

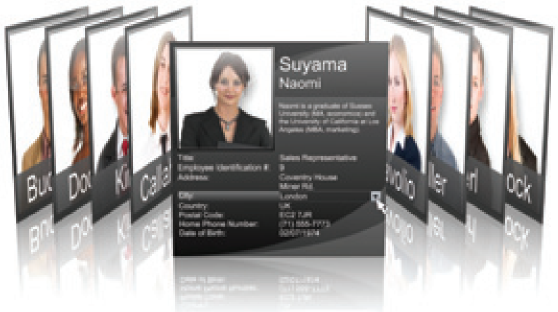
Vincent Smith
U2 Tools Product Manager at IBM

Microsoft® Visual Studio® PARTNER

XCEED
MULTI-TALENTED COMPONENTS

Datagrids, transformed

- Display & edit data in *stunning 2D* or *3D*
- *Highest-performance* WPF datagrid
- Most *adopted*, most *mature* WPF control
- *150* features, *10* major releases in *3* years

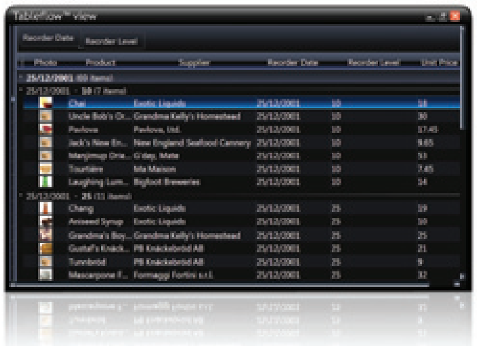


Suyama
Naomi

Naomi is a graduate of Sussex University (BA, economics) and the University of California at Los Angeles (MBA, marketing).

Title:	Sales Representative
Employee Identification #:	9
Address:	Coventry House
	Miner Rd
City:	London
Country:	UK
Postal Code:	EC2 7JR
Home Phone Number:	(71) 555-7773
Date of Birth:	02/07/1974

The Coverflow™ 3D view provides a true 3D experience to add to your application. Also offered is a classic 2D card view.



Tableflow™ View

Reorder Date
Reorder Level

Photo	Product	Supplier	Reorder Date	Reorder Level	Unit Price
25/12/2001 (00 items)					
25/12/2001 - 10 (7 items)					
	Chai	Exotic Liquids	25/12/2001	10	18
	Uncle Bob's Or...	Grandma Kelly's Homestead	25/12/2001	10	30
	Pavlova	Pavlova, Ltd.	25/12/2001	10	17.45
	Jack's New En...	New England Seafood Cannery	25/12/2001	10	9.65
	Manjimup Drie...	G'day, Mate	25/12/2001	10	53
	Tourtière	Ma Maison	25/12/2001	10	7.45
	Laughing Lum...	Bigfoot Breweries	25/12/2001	10	14
25/12/2001 - 25 (11 items)					
	Chang	Exotic Liquids	25/12/2001	25	19
	Aniseed Syrup	Exotic Liquids	25/12/2001	25	10
	Grandma's Boy...	Grandma Kelly's Homestead	25/12/2001	25	25
	Gustaf's Knäck...	PB Knäckebröd AB	25/12/2001	25	21
	Tunnbröd	PB Knäckebröd AB	25/12/2001	25	9
	Mascarpone F...	Formaggi Fortini s.r.l.	25/12/2001	25	32

The new smooth-scrolling Tableflow™ view provides an amazingly rich, fluid, and high-performance user experience.

Try it live on xceed.com

XCEED
MULTI-TALENTED COMPONENTS

Surprise!

avoid, at all costs, questions about whose time and dime you spent developing a cohesive plan.

Any successful and sustained effort to remove a large amount of debt absolutely requires the support of your project's financiers and sponsors. The folks who write the checks need to understand the investment you're making. This can be a challenge; you're asking people to think in long range, into the future, and to move away from the buy now, pay later mentality. The explanation "just because" simply doesn't cut it.

The problem with this is that executives will inevitably ask, "Aren't you professionals?" You might feel put against the ropes when probed along these lines. After all, weren't they paying you, the pro, to deliver a quality product on time and in budget?

This is a tough argument to counter. I say don't bother. Have the courage and honesty to present the facts as they are. This seemingly risky approach boils down to the very human issues of accountability and trust.

Couch your argument like this: you've fielded successful software in the requested amount of time for a provisioned amount of money. In order to achieve this you've had to make compromises along the way in response to business pressures. Now, to go forward at a predictable and steady rate, you need to deal with the effects of these compromises. The entire organization has bought them, and now it's time to pay back.

Your next challenge is to prove to non-technical folks where the debt is causing the most damage. In my experience, business executives respond to quantitative, data-driven arguments supported by "numbers" and "facts." I put numbers and facts in quotes because we all really know we're living in a relative world and no single number (cyclomatic complexity, efferent coupling, lines of code, test coverage, what have you) sells a change. Compounding this difficulty, you'll need to communicate the areas of biggest drain in economic terms: why is this slower than you'd like; why did this feature cost so much?

Evidence DEFEATS Doubt

When building your case, there's an immensely useful tool from the Dale Carnegie management training system embodied in a pithy phrase, "evidence defeats doubt." As is common with such management systems (and our discipline in general), the DEFEATS part is an acronym. I'll detail some of the ways in which this applies to software development. Note, however, that I've omitted the second E which stands for Exhibit because it seems to repeat the first E, which stands for Example.

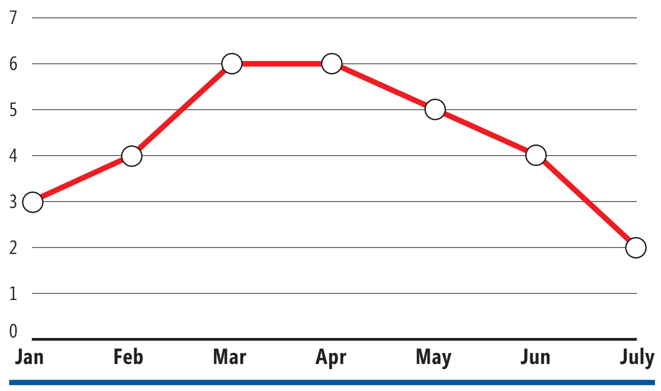


Figure 2 **Tracking Development Velocity**

D is for *Demonstration*. There's nothing better than show and tell and this is what the demonstration is all about. If you're tracking your velocity, this should be easy. Show the dip over time (see **Figure 2**) while drawing the connection to increasingly inflexible and hard-to-change code. Once you sell, you need to keep selling.

If you're using an Agile process such as Scrum or eXtreme Programming, customer feedback events are an essential practice. At the end of an iteration, demonstrate new features to your customer. While the quality and quantity of features will dip when you encounter the technical debt tar pits and while you ramp up your improvement efforts, you should be able to demonstrate gains over time. Less debt means greater output and greater output yields more stuff to demonstrate.

As the idiom goes, before you criticize someone, walk a mile in their shoes. If you have a more-technical manager, encourage her to work with developers on some of the more difficult sections of the codebase to develop empathy for the difficulty of change. Ask her to look at some code. Can she follow it? Is it readable? There's no quicker way to win your champion.

E is for *Example*. There's nothing like a concrete example. Find some stories or requirements that were either impossible to complete because of technical debt, or created significant regression. Pick a section of code that's unreadable, byzantine, riddled with side-effects. Explain how these attributes of the code led to a customer-found defect or the need for massive effort from QA.

Another powerful tool the Agile processes give you is the retrospective. Choose a story that went south in the last couple of iterations and ask the question "why?" Get to the root cause of why this particular story couldn't be completed, took twice as long as your average story, or spanned more than a single iteration. Often, inflexible software will be the culprit or perhaps you had to revert changes because regression bugs were insurmountable. If you find the last "why" ends up being a technical debt-related reason, capture the analysis in a short, direct form. It's another feather in your cap, another point in your argument.

F is for *Fact*. Facts are very easy to come by. Did you release a project on time? What was the post-release defect rate? What is the team's average velocity over time? Were customers satisfied with the software as delivered? These are the kind of facts you'll want to bring to the business table, and I believe it's these facts that speak most directly to business-minded.

Collaboration is a key element here. As a developer, you can more readily supply technical facts. Look for assistance from the people that own the budgets. Chances are they'll have a much clearer picture and easier access to the business facts that demonstrate the damage that technical debt is causing.

A is for *Analogy*. I find this especially important. Business people sometimes find software development confusing, even esoteric. If you go to your sponsors with talk of coupling and cohesion and Single Responsibility Principle, you stand a very good chance of losing them. But these are very important concepts in professional software development and, ultimately, it's how you're building a data-driven case for tackling debt. My suggestion is to avoid jargon and explain these items with an analogy.

You could describe coupling as a house of cards, for example. Tell your sponsors that the reason your velocity has dropped is because making change to the code is like adding a wall, ceiling, or story to an already established and very elaborate house of cards: a surgical operation requiring an unusually steady hand, significant time and patience, and is ultimately an uncertain and anxiety-provoking event. Sometimes the house of cards collapses.

Now that you've got a plan, it's time to communicate the value of eliminating debt to your project sponsors.

When employing metaphor and simile, it's a good idea to state you are doing so. Justify your analogy with a brief description of the more-technical concept you are trying to convey. Using the house of cards example, you might say, "this is the effect that coupling has on our ability to respond to change and add new features."

T is for *Testimonial*. Sometimes hearing the same message from a third party can have a more powerful effect. This third party may be an industry leader or a consultant. The reason their word might go farther than yours is that they're perceived as an objective expert.

If you don't have the money to hire an outside consultant, consider collecting anecdotes and insight freely available from industry thought leaders. While generic testimonials about so-called best practices are unlikely to seal the deal, they will add to the gestalt of your overall argument.

S is for *Statistics*. Numbers matter. There's a common phrase in management, "if you can't measure it, you can't manage it." I'm not sure this conventional wisdom applies wholly, but you can certainly present a case. Coupling and complexity are two metrics that can be used to show a root-cause relationship between a declining throughput (how much work is being delivered) and a codebase increasingly calcified with debt.

I find that composite statistics are usually the best bet here; it's much easier to understand the importance of code coverage if you can overlay a code coverage metric that decreases over time with a decrease in velocity, thus implying, if not showing, a relationship.

Appoint a Leader

Your efforts to procure a green light for fixing technical debt will go a lot longer with an effective leader, a champion who can communicate in business terms and who has influence with the decision makers in your organization. Often, this will be your manager, her director, the CTO, the VP of Engineering, or someone is a similar position of perceived authority.

This brings up an interesting chicken and egg problem. How do you sell this person? The process of "managing up" is a developer's responsibility, too. Your first challenge is to sell the seller. How exactly do you do that? Evidence defeats doubt!

Next Steps

So far I've covered identifying debt as a team and building a case for fixing that debt. I'll reiterate: consensus among your team and buy-in with your customers are key factors in these steps.

Make the steps small and don't invest a lot of time. The first time you identify debt, it will necessarily take longer than when you iterate over new opportunities for improvement, but when you build your case for management, only include those items you plan to work on. Keeping an eye on productivity can be a huge energy saver.

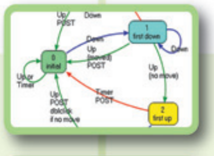
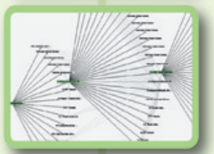
In a future issue I'll look at the rest of the workflow, including tactics for eliminating debt, and I'll cover how you can make this process iterative, and capturing the lessons learned from previous debt removal efforts. ■

DAVE LARIBEE *coaches the product development team at VersionOne. He is a frequent speaker at local and national developer events and was awarded a Microsoft Architecture MVP for 2007 and 2008. He writes on the CodeBetter blog network at thebeelog.com.*

THANKS to the following technical experts for reviewing this article: *Donald Belcham*

Build interactive diagrams easier than you ever imagined

Create custom interactive diagrams, network editors, workflows, flow-charts and design tools. For web servers or local applications. It's easy-to-use and very flexible. We're the first developer of diagramming components and still the best. Find out for yourself: download our FREE fully functional evaluation kit, with full support at www.nwoods.com.

Now supporting WPF & Silverlight

GoDiagram

Interactive diagram components

www.nwoods.com

Building a Desktop To-Do Application with NHibernate

Oren Eini

NHibernate is an Object Relational Mapper (OR/M), tasked with making it as easy to work with a database as it is to work with in-memory objects. It is one of the most popular OR/M frameworks for Microsoft .NET Framework development. But most users of NHibernate are doing so in the context of Web applications, so there is relatively little information about building NHibernate applications on the desktop.

When using NHibernate in a Web application, I tend to use the session-per-request style, which has a lot of implications that are easy to miss. I don't worry about the session maintaining a reference to the loaded entities, because I expect that the session will go away shortly. I don't have to worry about error handling (much), as I can just abort the current request and its associated session if an error occurs.

The lifetime of the session is well defined. I don't need to update other sessions about any changes that I made. I don't need to worry about holding a long transaction in the database, or even holding a connection open for a long time, because they are only alive for the duration of a single request.

As you can imagine, those are concerns in a desktop application. Just to be clear about it, I am talking about an application talking to a database directly. An application that uses some sort of remote services is using NHibernate on the remote server, under the per-request scenario, and is not the focus of this article. This article does not cover occasionally connected scenarios, although much of the discussion here would apply to those scenarios.

Building an NHibernate-based desktop application isn't much different than building a desktop application using any other persistence technology. Many of the challenges that I intend to outline in this article are shared between all data-access technologies:

- Managing the scope of units of work.
- Reducing duration of opened database connections.
- Propagating entity changes to all parts of the application.
- Supporting two-way data binding.
- Reducing startup times.
- Avoiding blocking the UI thread while accessing the database.
- Handling and resolving concurrency conflicts.

This article discusses:

- Managing sessions and connections
- Publishing events
- Using the Interceptor
- Handling concurrency issues

Technologies discussed:

NHibernate, .NET Framework

Code download available at:

code.msdn.microsoft.com/mag200912NHibernate

While the solutions that I give are dealing exclusively with NHibernate, at least the majority of them are also applicable to other data-access technologies. One such challenge, shared by all data-access technologies that I am aware of, is how to manage the scope of the application's unit of work—or, in NHibernate's terms, the session lifetime.

Managing Sessions

A common bad practice with NHibernate desktop applications is to have a single global session for the entire application. It is a problem for many reasons, but three of them are most important. Because a session keeps a reference to everything that it loaded, the user working on the application is going to load an entity, work with it a bit, and then forget about it. But because the single global session is maintaining a reference to it, the entity is never released. In essence, you have a memory leak in your application.

Then there is the problem of error handling. If you get an exception (such as StaleObjectStateException, because of concurrency conflict), your session and its loaded entities are toast, because with NHibernate, an exception thrown from a session moves that session into an undefined state. You can no longer use that session or any loaded entities. If you have only a single global session, it means that you probably need to restart the application, which is probably not a good idea.

Finally, there's the issue of transaction and connection handling. While opening a session isn't akin to opening a database connection, using a single session means that you are far more likely to hold transactions and connection for longer than you should.

Another equally bad and, unfortunately, almost as common practice with NHibernate is micromanaging the session. A typical example is code like this:

```
public void Save(ToDoAction action) {
  using(var session = sessionFactory.OpenSession())
  using(var tx = session.BeginTransaction()) {
    session.SaveOrUpdate(action);

    tx.Commit();
  }
}
```

The problem with this type of code is that it removes a lot of the advantages that you gain in using NHibernate. NHibernate is doing quite a bit of work to handle change management and transparent persistence for you. Micromanaging the session cuts off NHibernate's ability to do that and moves the onus of doing that work to you. And that is even without mentioning the problems it causes with lazy loading down the line. In just about any system where I have seen such an approach attempted, the developers had to work harder to resolve those problems.

A session should not be held open for too long, but it should also not be held open for too short a time to make use of NHibernate's capabilities. Generally, try to match the session lifetime to the actual action that is being performed by the system.

The recommended practice for desktop applications is to use a session per form, so that each form in the application has its own session. Each form usually represents a distinct piece of work that the user would like to perform, so matching session lifetime to the form lifetime works quite well in practice. The added benefit is that you no longer have a problem with memory leaks, because when you close a form in the application, you also dispose of the session. This would make all the entities that were loaded by the session eligible for reclamation by the garbage collector (GC).

There are additional reasons for preferring a single session per form. You can take advantage of NHibernate's change tracking, so it will flush all changes to the database when you commit the transaction. It also creates an isolation barrier between the different forms, so you can commit changes to a single entity without worrying about changes to other entities that are shown on other forms.

While this style of managing the session lifetime is described as a session per form, in practice you usually manage the session per presenter. The code in **Figure 1** is taken from the presenter base class.

Figure 1 **Presenter Session Management**

```
protected ISession Session {
  get {
    if (session == null)
      session = sessionFactory.OpenSession();
    return session;
  }
}

protected IStatelessSession StatelessSession {
  get {
    if (statelessSession == null)
      statelessSession = sessionFactory.OpenStatelessSession();
    return statelessSession;
  }
}

public virtual void Dispose() {
  if (session != null)
    session.Dispose();
  if (statelessSession != null)
    statelessSession.Dispose();
}
```

As you can see, I lazily open a session (or a stateless session) and keep it open until I dispose of the presenter. This style matches quite nicely to the lifetime of the form itself and allows me to associate a separate session with each presenter.

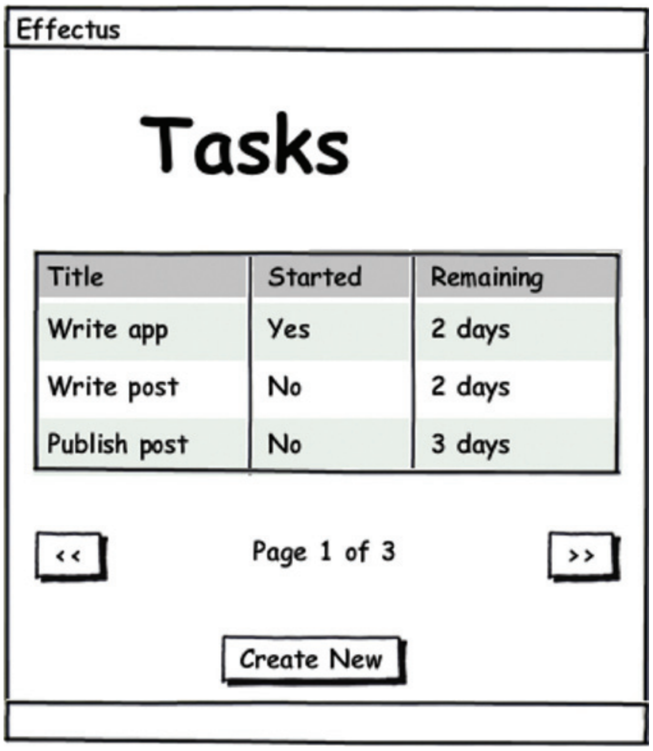
Maintaining Connections

In the presenters, you don't have to worry about opening or closing the session. The first time that you access a session, it's opened for you, and it will dispose itself properly as well. But what about the database connection associated with the session? Are you holding a database connection open for as long as the user is viewing the form?

Most databases dislike having to hold a transaction open for extended periods of time. It usually results in causing errors or deadlocks down the line. Opened connections can cause similar problems, because a database can only accept so many connections before it runs out of resources to handle the connection.

To maximize performance of your database server, you should keep transaction lifespan to a minimum and close connections as soon as possible, relying on connection pooling to ensure fast response times when you open a new connection.

With NHibernate, the situation is much the same, except that NHibernate contains several features that are there to explicitly


Effectus

Tasks

Title	Started	Remaining
Write app	Yes	2 days
Write post	No	2 days
Publish post	No	3 days

Figure 2 **A To-Do List Application**

make things easier for you. The NHibernate session doesn't have a one-to-one association with a database connection. Instead, NHibernate manages the database connection internally, opening and closing it as needed. This means you don't have to maintain some sort of state in the application to disconnect and reconnect to the database as needed. By default, NHibernate will minimize to the maximum extent the duration in which a connection is open.

You do need to worry about making transactions as small as possible. In particular, one of the things that you don't want is to hold a transaction open for the lifetime of the form. This will force NHibernate to keep the connection open for the duration of the transaction. And because the lifetime of a form is measured in human response times, it's more than likely that you would end up holding up a transaction and connection for longer periods than is really healthy.

What you will usually do is open separate transactions for each operation that you make. Let's look at a form mockup that shows a simple to-do list, my sample application of choice (see **Figure 2**). The code for handling this form is quite simple, as you can see in **Figure 3**.

I have three operations: loading the form for the first time, showing the first page, and paging back and forth through the records.

On each operation, I begin and commit a separate transaction. That way I don't consume any resources on the database and don't have to worry about long transactions. NHibernate will automatically open a connection to the database when I begin a new transaction and close it once the transaction is completed.

Stateless Sessions

There is another small subtlety that I should note: I'm not using an ISession. Rather, I'm using IStatelessSession in its place to load the data. Stateless sessions are typically used in bulk data manipulation, but in this case I'm making use of a stateless session to resolve memory consumption issues.

A stateless session is, well, stateless. Unlike a normal session, it doesn't maintain a reference to the entities that it loads. As such, it's perfectly suited to loading entities for display-only purposes. For that type of task, you generally load the entities from the database, throw them on the form and forget about them. A stateless session is just what you need in this case.

But stateless sessions come with a set of limitations. Chief among them in this case is that stateless sessions do not support lazy loading, do not involve themselves in the usual NHibernate event mode and do not make use of NHibernate's caching features.

For those reasons, I generally use them for simple queries where I just want to show the user the information without doing anything complicated. In cases where I want to show an entity for editing, I could still make use of a stateless session, but I tend to avoid that in favor of a normal session.

In the main application form, I strive to make all the data display-only, and try to make use of stateless sessions alone. The main form lives for as long as the application is open and a stateful session is going to be a problem, not only because it will maintain a reference to the entities that it loaded, but because it's likely to cause problems if the session throws an exception.

NHibernate considers a session that threw an exception to be in an undefined state (only Dispose has a defined behavior in this case). You would need to replace the session, and because entities loaded by a stateful session maintain a reference to it, you would need to

Figure 3 **Creating the To-Do Form**

```
public void OnLoaded() {
  LoadPage(0);
}

public void OnMoveNext() {
  LoadPage(CurrentPage + 1);
}

public void OnMovePrev() {
  LoadPage(CurrentPage - 1);
}

private void LoadPage(int page) {
  using (var tx = StatelessSession.BeginTransaction()) {
    var actions = StatelessSession.CreateCriteria<ToDoAction>()
      .SetFirstResult(page * PageSize)
      .SetMaxResults(PageSize)
      .List<ToDoAction>();

    var total = StatelessSession.CreateCriteria<ToDoAction>()
      .SetProjection(Projections.RowCount())
      .UniqueResult<int>();

    this.NumberOfPages.Value = total / PageSize +
               (total % PageSize == 0 ? 0 : 1);
    this.Model = new Model {
      Actions = new ObservableCollection<ToDoAction>(actions),
      NumberOfPages = NumberOfPages,
      CurrentPage = CurrentPage + 1
    };
    this.CurrentPage.Value = page;

    tx.Commit();
  }
}
```

clear all the entities that were loaded by the now-defunct session. It's so much simpler to use a stateless session in this circumstance.

Entities loaded by stateless sessions do not care for the state of the session, and recovering from an error in a stateless session is as simple as closing the current stateless session and opening a new one.

Manipulating Data

Figure 4 shows the edit screen mockup. What challenges do you face when you need to edit entities?

Well, you actually have two separate challenges here. First, you want to be able to make use of NHibernate's change tracking, so you can display an entity (or an entity object graph) and have NHibernate just persist it when you're finished. Second, once you save an entity, you want to ensure that every form that also displays this entity is updated with the new values.

The first item of business is actually fairly easy to handle. All you need to do is make use of the session associated with the form, and that's it. **Figure 5** shows the code driving this screen.

You get the entity from the database in the Initialize(id) method, and you update it in the OnSave method. Notice that you do so in two separate transactions, instead of keeping a transaction alive for a long period of time. There's also this strange EventPublisher call. What's that all about?

EventPublisher is here in order to deal with another challenge: when each form has its session, then each form has different instances of the entities you work with. On the face of it, that looks like a waste. Why should you load the same action several times?

In actuality, having this separation between the forms will simplify the application considerably. Consider what would happen if you shared the entity instances across the board. In that situation, you would find yourself with a problem in any conceivable edit scenario. Consider what would happen if you were to display an entity in two forms that allow editing that entity. That may be an editable grid and a detailed edit form, for example. If you make a change to the entity in the grid, open the detailed edit form and then save it, what would happen to the change you made on the editable grid?

If you employ a single entity instance throughout the application, then it's likely that saving the details form would also cause you to save the changes made using the grid. That's likely not something that you would want to do. Sharing an entity instance also makes it much more difficult to do things like cancel an edit form and have all the unsaved changes go away.

Those problems simply do not exist when you use an entity instance per form, which is a good thing, as this is more or less mandatory when you use a session per form approach.

Publishing Events

But I haven't fully explained the purpose of the EventPublisher yet. It's actually fairly simple. Instead of having a single instance of the entity in the application, you may have many, but the user would still like to see the entity (once properly saved) updated on all the forms that show that entity.

In my example I do so explicitly. Whenever I save an entity, I publish an event saying that I did so, and on which entity. This isn't a standard .NET event. A .NET event requires a class to subscribe to it directly. That doesn't actually work for this type of notification because it would require each form to register to events in all other forms. Just trying to manage that would be a nightmare.

The EventPublisher is a publish-subscribe mechanism that I use to decouple a publisher from its subscriber. The only commonality between them is the EventPublisher class. I use the event type (ActionUpdated in **Figure 5**) to decide who to tell about the event.

Let's look at the other side of that now. When I update a to-do action, I would like to show the updated values in the main form, which shows a grid of to-do actions. Here is the relevant code from that form presenter:

```
public Presenter() {
  EventPublisher.Register<ActionUpdated>(
    RefreshCurrentPage);
}

private void RefreshCurrentPage(
  ActionUpdated actionUpdated) {
  LoadPage(CurrentPage);
}
```

On startup, I register the method RefreshCurrentPage to the ActionUpdated event. Now, whenever that event is raised, I will simply refresh the current page by calling LoadPage, which you are already familiar with.

This is actually a fairly lazy implementation. I don't care if the current page is showing the edited entity; I just refresh it anyway. A more complex (and efficient) implementation would only refresh the grid data if the updated entity is shown on that page.

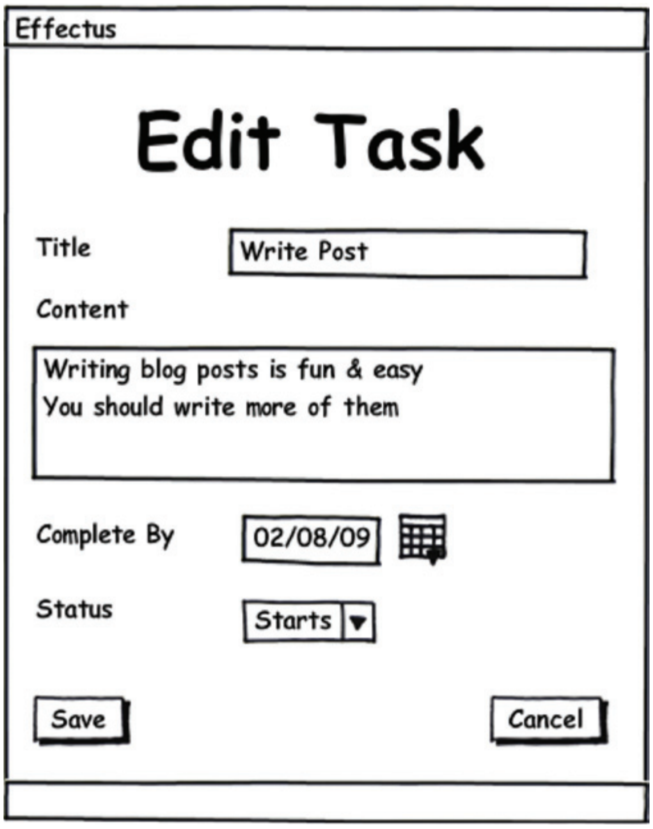


Figure 4 **Editing Entities**

Figure 5 **Editing an Entity in the Session**

```
public void Initialize(long id) {
  ToDoAction action;
  using (var tx = Session.BeginTransaction()) {
    action = Session.Get<ToDoAction>(id);
    tx.Commit();
  }

  if(action == null)
    throw new InvalidOperationException(
      "Action " + id + " does not exists");

  this.Model = new Model {
    Action = action
  };
}

public void OnSave() {
  using (var tx = Session.BeginTransaction()) {
    // this isn't strictly necessary, NHibernate will
    // automatically do it for us, but it make things
    // more explicit
    Session.Update(Model.Action);

    tx.Commit();
  }

  EventPublisher.Publish(new ActionUpdated {
    Id = Model.Action.Id
  }, this);

  View.Close();
}
```

The main advantage of using the publish-subscribe mechanism in this manner is decoupling the publisher and subscribers. I don't care in the main form that the edit form publishes the ActionUpdated event. The idea of event publishing and publish-subscribe is a cornerstone in building loosely coupled user interfaces, and is covered extensively in the Composite Application Guidance (msdn.microsoft.com/library/cc707819) from the Microsoft patterns & practices team.

There is another case worth considering: What would happen if you have two edit forms to the same entity open at the same time? How can you get the new values from the database and show them to the user?

The following code is taken from the edit form presenter:

```
public Presenter() {
  EventPublisher.Register<ActionUpdated>(RefreshAction);
}

private void RefreshAction(ActionUpdated actionUpdated) {
  if(actionUpdated.Id != Model.Action.Id)
    return;
  Session.Refresh(Model.Action);
}
```

This code registers for the ActionUpdated event, and if it's the entity that you're editing, you ask NHibernate to refresh it from the database.

This explicit model of refreshing the entity from the database also gives you the chance to make decisions about what should happen now. Should you update automatically, erasing all the user changes? Should you ask the user? Try to silently merge the changes? Those are all decisions that you now have the chance to deal with in a straightforward manner.

In most cases, however, I find that simply refreshing the entity is quite enough, because you generally do not allow updating of a single entity in parallel (at least not by a single user).

While this entity refresh code will indeed update the values of the entity instance, how are you going to make the UI respond to this change? You have data bound the entity values to the form fields, but you need some way of telling the UI that those values have changed.

The Microsoft .NET Framework provides the INotifyPropertyChanged interface, which most UI frameworks understand and know how to work with. Here's the INotifyPropertyChanged definition:

```
public delegate void PropertyChangedEventHandler(
  object sender, PropertyChangedEventArgs e);

public class PropertyChangedEventArgs : EventArgs {
  public PropertyChangedEventArgs(string propertyName);
  public virtual string PropertyName { get; }
}

public interface INotifyPropertyChanged {
  event PropertyChangedEventHandler PropertyChanged;
}
```

An object that implements this interface should raise the PropertyChanged event with the name of the property that was changed. The UI will subscribe to the PropertyChanged event and whenever a change is raised on a property that's bound, it will refresh the binding.

Implementing this is quite easy:

```
public class Action : INotifyPropertyChanged {
  private string title;
  public virtual string Title {
    get { return title; }
    set {
      title = value;
      PropertyChanged(this,
        new PropertyChangedEventArgs("Title"));
    }
  }

  public event PropertyChangedEventHandler
    PropertyChanged = delegate { };
}
```

While simple, it's fairly repetitive code, and is only required to satisfy UI infrastructure concerns.

Intercepting Entity Creation

I don't want to have to write code just to get the UI data binding working properly. And as it turns out, I don't actually need to.

One of the requirements of NHibernate is that you make all properties and methods on your classes virtual. NHibernate requires this to handle lazy loading concerns properly, but you can take advantage of this requirement for other reasons.

One thing you can do is take advantage of the virtual keyword to inject your own behavior into the mix. You do this using a technique called Aspect-Oriented Programming (AOP). In essence, you take a class and add additional behaviors to this class at runtime. The exact mechanism of how you implement this is outside the scope of the article, but it's encapsulated in the DataBindingFactory class, whose definition is:

```
public static class DataBindingFactory {
  public static T Create<T>();
  public static object Create(Type type);
}
```

The entire implementation of the class is about 40 lines, not terribly complex. What it does is take a type and produce an instance of this type that also fully implements the INotifyPropertyChanged contract. In other words, the following test will work:

```
ToDoAction action = DataBindingFactory.Create<ToDoAction>();
string changedProp = null;
((INotifyPropertyChanged)action).PropertyChanged
  += (sender, args) => changedProp = args.PropertyName;
action.Title = "new val";
Assert.Equal("Title", changedProp);
```

Given that, all you have to do now is make use of the DataBindingFactory whenever you create a new class in the presenters. The main advantage that you gain from such a system is that now, if you would like to make use of the NHibernate domain model in a non-presentation context, you can simply not make use of the DataBindingFactory, and you get a domain model completely free from presentation concerns.

There's still one problem, though. While you can create *new* instances of entities using the DataBindingFactory, a lot of the time you will have to deal with instances that were created by NHibernate. Obviously, NHibernate knows nothing about your DataBindingFactory and can't make use of it. But before you despair, you can make use of one of the most useful extension points with NHibernate, the Interceptor. NHibernate's Interceptor allows you to take over, in essence, some of the functionalities that NHibernate is performing internally.

One of the functionalities that the Interceptor allows you to take over is creating new instances of entities. **Figure 6** shows an Interceptor that creates instances of entities using the DataBindingFactory.

You override the Instantiate method and handle the case where we get an entity with a type that you recognize. You then proceed to create an instance of the class and set its identifier property. You also need to teach NHibernate how to understand what type an instance created via DataBindingFactory belongs to, which you do in the GetEntityName method of the intercepter.

The only thing left now is to set up NHibernate with the new Interceptor. The following is taken from the BootStrapper class, responsible for setting up the application:

```
public static void Initialize() {
  Configuration = LoadConfigurationFromFile();
  if(Configuration == null) {
    Configuration = new Configuration()
      .Configure("hibernate.cfg.xml");
    SaveConfigurationToFile(Configuration);
  }
  var intercepter = new DataBindingIntercepter();
  SessionFactory = Configuration
    .SetInterceptor(intercepter)
    .BuildSessionFactory();
  intercepter.SessionFactory = SessionFactory;
}
```

For now, ignore the configuration semantics—I will address that in a bit. The important point is that you create the Interceptor, set it on the configuration and build the session factory. The last step is setting the session factory on the Interceptor. It's a bit awkward, I'll admit, but that's the simplest way to get the appropriate session factory into the Interceptor.

Once the Interceptor is wired, every entity instance that NHibernate creates will now support INotifyPropertyChanged notifications without you having to do any work at all. I consider this quite an elegant solution to the problem.

There are a few who would say that choosing such a solution is a problem from a performance perspective over hard coding the implementation. In practice, that turns out to be a false assumption. The tool that I'm using (Castle Dynamic Proxy) to perform this on-the-fly extension of classes has been heavily optimized to ensure optimal performance.

Addressing Performance

Speaking of performance, an additional concern in desktop applications that you do not have in Web applications is startup time. In Web applications it is quite common to decide to favor longer startup times to increase request performance. In desktop

Generally, try to match the session lifetime to the actual action that is being performed by the system.

applications, you would like to reduce the startup time as much as possible. In fact, a common cheat with desktop application is to simply show a screen shot of the application to the user until the application finishes starting up.

Unfortunately, NHibernate startup time is somewhat long. This is mostly because NHibernate is performing a lot of initialization and checks on startup, so it can perform faster during normal operation. There are two common ways of handling this issue.

The first is to start NHibernate in a background thread. While this means the UI will show up much faster, it also creates a complication

Figure 6 **Intercepting Entity Creation**

```
public class DataBindingInterceptor : EmptyInterceptor {
  public ISessionFactory SessionFactory { set; get; }

  public override object Instantiate(string clazz,
    EntityMode entityMode, object id) {

    if(entityMode == EntityMode.Poco) {
      Type type = Type.GetType(clazz);
      if (type != null) {
        var instance = DataBindingFactory.Create(type);
        SessionFactory.GetClassMetadata(clazz)
          .SetIdentifier(instance, id, entityMode);
        return instance;
      }
    }
    return base.Instantiate(clazz, entityMode, id);
  }

  public override string GetEntityName(object entity) {
    var markerInterface = entity as
      DataBindingFactory.IMarkerInterface;
    if (markerInterface != null)
      return markerInterface.TypeName;
    return base.GetEntityName(entity);
  }
}
```

for the application itself, because you can't show the user anything from the database until you finish the session factory startup.

The other option is to serialize NHibernate's Configuration class. A large amount of the cost related to NHibernate startup is related to the cost of validating the information passed to the Configuration class. The Configuration class is a serializable class and therefore you can pay that price only once, after which you can shortcut the cost by loading an already validated instance from persistent storage.

That is the purpose of the LoadConfigurationFromFile and SaveConfigurationToFile, serializing and deserializing NHibernate's configuration. Using these you only have to create the configuration the first time you start the application. But there's a small catch you should be aware of: You should invalidate the cached configuration if the entities assembly or the NHibernate configuration file has changed.

The sample code for this article contains a full implementation that's aware of this and invalidates the cached file if the entities or the configuration have changed.

There's another performance issue that you have to deal with. Calling the database is one of the more expensive operations that the application makes. As such, it's not something that you'd want to do on the application UI thread.

Such duties are often relegated to a background thread, and you can do the same with NHibernate, but keep in mind that the NHibernate session is not thread safe. While you can make use of a session in multiple threads (it has no thread affinity), you must not use a session (or your entities) on multiple threads in parallel. In other words, it's perfectly fine to use the session in a background thread, but you must serialize access to the session and not allow concurrent access to it. Using the session from multiple threads in parallel will result in undefined behavior; in other words, it should be avoided.

Luckily, there are a few relatively simple measures that you can take in order to ensure that access to the session is serialized. The System.ComponentModel.BackgroundWorker class was explicitly designed to handle these sorts of tasks. It allows you to execute a task on a background thread and notify you when it is completed, taking care of the UI thread synchronization issue, which is so important in desktop applications.

You saw earlier how to manage editing an existing entity, which I did directly on the UI thread. Now, let's save a new entity on a background thread. The following code is the initialization of the Create New presenter:

```
private readonly BackgroundWorker saveBackgroundWorker;

public Presenter() {
  saveBackgroundWorker = new BackgroundWorker();
  saveBackgroundWorker.DoWork +=
    (sender, args) => PerformActualSave();
  saveBackgroundWorker.RunWorkerCompleted +=
    (sender, args) => CompleteSave();
  Model = new Model {
    Action = DataBindingFactory.Create<ToDoAction>(),
    AllowEditing = new Observable<bool>(true)
  };
}
```

The BackgroundWorker is used to perform the actual save process, which was split into two distinct pieces. Aside from that split, it's very similar to the way I handled it in the edit scenario. Another interesting bit that you need to pay attention to is the AllowEditing property; this property is used to lock the UI in the form when you're performing a save operation. This way, you can safely use the session in another thread, knowing that there won't be concurrent access to either the session or any of its entity by that form.

The saving process itself should be pretty familiar to you by now. Let's look at the OnSave method first:

```
public void OnSave() {
  Model.AllowEditing.Value = false;
  saveBackgroundWorker.RunWorkerAsync();
}
```

This method is responsible for disabling editing in the form, then kicking off the background process. In the background, you perform the actual save. The code shouldn't come as a surprise:

```
private void PerformActualSave() {
  using(var tx = Session.BeginTransaction()) {
    Model.Action.CreatedAt = DateTime.Now;

    Session.Save(Model.Action);
    tx.Commit();
  }
}
```

When the actual save to the database is completed, the BackgroundWorker will execute the CompleteSave part of the process in the UI thread:

```
private void CompleteSave() {
  Model.AllowEditing.Value = true;
  EventPublisher.Publish(new ActionUpdated {
    Id = Model.Action.Id
  }, this);

  View.Close();
}
```

You re-enable the form, publish a notification that an action was updated (causing the relevant screens to update as well), and finally close the form. I suppose that enabling the UI isn't strictly necessary, but I included it there for completion sake.

Using this technique, you can take advantage of background processing without violating the threading contract on the session

Figure 7 **Handling Concurrency Conflicts**

```
public void OnSave() {
  bool successfulSave;
  try {
    using (var tx = Session.BeginTransaction()) {
      Session.Update(Model.Action);

      tx.Commit();
    }
    successfulSave = true;
  }
  catch (StaleObjectStateException) {
    successfulSave = false;
    MessageBox.Show(
      @"Another user already edited the action before you had a chance to
do so. The application will now reload the new data from the database,
please retry your changes and save again.");

    ReplaceSessionAfterError();
  }

  EventPublisher.Publish(new ActionUpdated {
    Id = Model.Action.Id
  }, this);

  if (successfulSave)
    View.Close();
}
```

XCEED

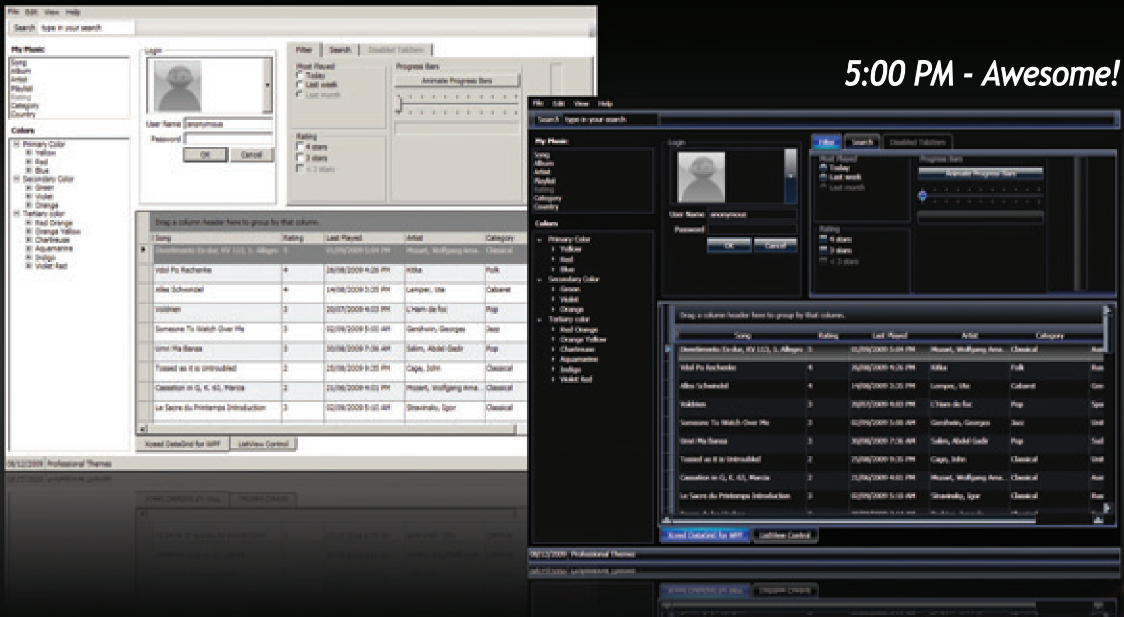
Professional Themes

for WPF

Beautifully themed WPF apps in under 2 minutes.

4:58 PM - Ewww!

5:00 PM - Awesome!



Song	Rating	Last Played	Artist	Category
Divertimento Es-dur, KV 113, 1. Allegro	5	11/09/2009 5:04 PM	Mozart, Wolfgang Ama...	Classical
Vdol Po Rechenke	4	26/08/2009 4:26 PM	Kitka	Folk
Alles Schwindel	4	14/08/2009 3:35 PM	Lemper, Ute	Cabaret
Voldmen	3	20/07/2009 4:03 PM	L'Ham de foc	Pop
Someone To Watch Over Me	3	02/09/2009 5:00 AM	Gershwin, Georges	Jazz
Umri Ma Banaa	3	30/08/2009 7:36 AM	Salim, Abdel Gadir	Pop
Tossed as it is Untroubled	2	25/08/2009 9:35 PM	Cage, John	Classical
Cassation in G, K. 63, Marcia	2	21/06/2009 4:01 PM	Mozart, Wolfgang Ama...	Classical
Le Sacre du Printemps Introduction	3	02/09/2009 5:10 AM	Stravinsky, Igor	Classical

Flawless WPF themes for styling all official Microsoft WPF controls. Requires as few as a couple of lines of XAML code to quickly apply beautiful, uniform styles to your entire WPF application.

Try it live on xceed.com

XCEED
MULTI-TALENTED COMPONENTS

Figure 8 **Updating Sessions and Entities**

```
protected void ReplaceSessionAfterError() {
  if(session!=null) {
    session.Dispose();
    session = sessionFactory.OpenSession();
    ReplaceEntitiesLoadedByFaultedSession();
  }
  if(statelessSession!=null) {
    statelessSession.Dispose();
    statelessSession = sessionFactory.OpenStatelessSession();
  }
}

protected override void
  ReplaceEntitiesLoadedByFaultedSession() {
  Initialize(Model.Action.Id);
}
```

instances. As always, threading is a great way to create a more responsive application, but multi-threaded programming is not a task to be approached lightly, so use this technique with care.

Dealing with Concurrency

Concurrency is a complex topic at the best of times, and it isn't limited to threading alone. Consider the case where you have two users that edit the same entity at the same time. One of them is going to hit the submit button first, saving the changes to the database. The question is, what should happen when the second user hits the save button?

This is called a concurrency conflict, and NHibernate has quite a few ways of detecting such a conflict. The ToDoAction entity has a <version/> field that tells NHibernate that it needs to explicitly perform optimistic concurrency checks. For a full discussion of the concurrency options that NHibernate offers, see my blog post at ayende.com/Blog/archive/2009/04/15/nhibernate-mapping-concurrency.aspx.

Essentially, the concurrency solutions fall into two broad categories:

- Pessimistic concurrency control, which requires you to hold locks on the database and keep the transaction open for an extended period of time. As I discussed earlier, this is not a good idea in a desktop application.
- Optimistic concurrency control, which means you can close the database connection during the user's "think time." Most of the options NHibernate has to offer are on the optimistic side, allowing several strategies to detect conflicts.

Because pessimistic concurrency control incurs such a heavy performance cost, it's generally not acceptable. This, in turn, means that you should favor optimistic concurrency control. With optimistic concurrency, you try to save the data normally, but you're prepared to handle the case where the data was changed by another user.

NHibernate will manifest this situation as a StaleObjectStateException during the save or commit processes. Your applications should catch that exception and behave accordingly. Usually, it means that you need to show some sort of a message to the user, explaining that the entity was edited by another user, and that the user needs to redo their changes. Occasionally, you need to perform more complex operations, such as offering to merge the information, or allowing the user to decide which version to keep.

Because the first option—displaying a message and having the user redo any changes—is much more common, I'll show how to implement that with NHibernate, and then discuss briefly how you can implement the other solutions.

One interesting problem that you face right away is that an exception raised from the session means the session is no longer usable. Any concurrency conflict shows up in NHibernate as an exception. The only thing that you may do to a session after it has thrown an exception is to call Dispose on it; any other operation will result in undefined behavior.

I'm going to go back to the edit screen and implement concurrency handling there as an example. I will add a Create Concurrency Conflict button to the edit screen, which will execute the following:

```
public void OnCreateConcurrencyConflict() {
  using(var session = SessionFactory.OpenSession())
  using(var tx = session.BeginTransaction()) {
    var anotherActionInstance =
      session.Get<ToDoAction>(Model.Action.Id);
    anotherActionInstance.Title =
      anotherActionInstance.Title + " -";
    tx.Commit();
  }
MessageBox.Show("Concurrency conflict created");
}
```

This creates a new session and modifies the title property. This will trigger a concurrency conflict when I try to save the entity in the form, because the session on the form is not aware of those changes. **Figure 7** shows how I'm handling that.

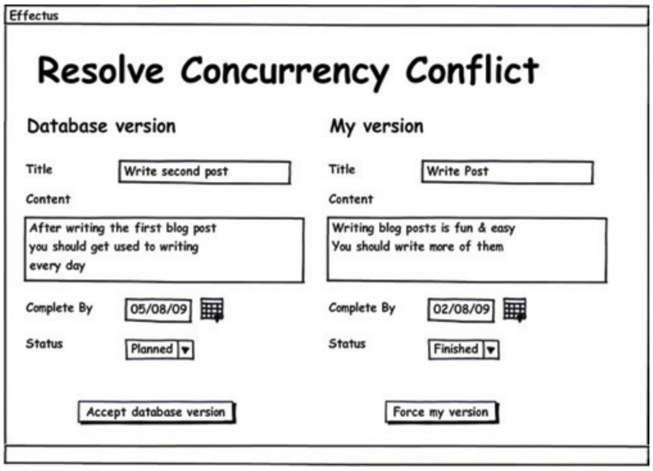
Despite the number of challenges outlined in this article, building an NHibernate desktop application isn't any harder than building a NHibernate Web application.

I simply wrapped the code that saves to the database in a try catch block and handled the stale state exception by informing the user that I detected a concurrency conflict. I then replace the session.

Notice that I am *always* calling ActionUpdated, even if I received a concurrency conflict. Here's why: Even if I got a concurrency conflict, the rest of the application probably doesn't know about it, and the entity was changed in the database, so I might as well give the rest of the application the chance to show the user the new values as well.

Finally, I only close the form if I've been successful in saving to the database. So far, there's nothing much, but there is the session and entity replacement that I still need to consider (see **Figure 8**).

As you can see, I replace the session or the stateless session by disposing them and opening new ones. In the case of a session, I also ask the presenter to replace all the entities that were loaded


Effectus

Resolve Concurrency Conflict

Database version		My version	
Title	Write second post	Title	Write Post
Content	After writing the first blog post you should get used to writing every day	Content	Writing blog posts is fun & easy You should write more of them
Complete By	05/08/09	Complete By	02/08/09
Status	Planned	Status	Finished

Figure 9 UI for Managing Change Conflicts

by the faulted session. NHibernate entities are closely associated to their session, and when the session become unusable, it's generally best to replace the entities as well. It is not *required*—the entities aren't going to suddenly stop working—but things like lazy loading will no longer work. I'd rather pay the cost of replacing the entities than try to figure out if I can or cannot traverse the object graph in certain cases.

The implementation of the entity replacement is done by just calling the Initialize method in this case. This is the same Initialize method that I discussed in the edit form case. This method just gets the entity from the database and sets it into the model property—nothing exciting. In more complex scenarios, it may replace several entity instances that are used in a single form.

For that matter, the same approach holds not only for concurrency conflicts, but for any error that you might get from NHibernate's sessions. Once you get an error, you must replace the session. And when you replace the session, you probably ought to reload any entities that you loaded using the old session in the new one, just to be on the safe side.

Conflict Management

The last topic that I want to touch upon in this article is the more complex concurrency conflict management techniques. There's only one option, basically: allow the user to make a decision between the version in the database and the version that the user just modified.

Figure 9 shows the merge screen mockup. As you can see, here you are simply showing the user both options and asking to choose which one they will accept. Any concurrency conflict resolution is somehow based on this idea. You may want to present it in a different way, but that's how it works, and you can extrapolate from here. In the edit screen, change the conflict resolution to this:

```
catch (StaleObjectStateException) {
  var mergeResult =
    Presenters.ShowDialog<MergeResult?>(
    "Merge", Model.Action);
  successfulSave = mergeResult != null;

  ReplaceSessionAfterError();
}
```

I show the merge dialog, and if the user has made a decision about the merge, I decide that it was a successful save (which would close the edit form). Note that I pass the currently edited action to the merge dialog, so it knows the current state of the entity.

The merge dialog presenter is straightforward:

```
public void Initialize(ToDoAction userVersion) {
  using(var tx = Session.BeginTransaction()) {
    Model = new Model {
      UserVersion = userVersion,
      DatabaseVersion =
        Session.Get<ToDoAction>(userVersion.Id),
        AllowEditing = new Observable<bool>(false)
    };

    tx.Commit();
  }
}
```

On Startup, I get the current version from the database and show both it and the version that the user changed. If the user accepts the database version, I don't have much to do, so I simply close the form:

```
public void OnAcceptDatabaseVersion() {
  // nothing to do
  Result = MergeResult.AcceptDatabaseVersion;
  View.Close();
}
```

If the user wants to force their own version, it's only slightly more complicated:

```
public void OnForceUserVersion() {
  using(var tx = Session.BeginTransaction()) {
    //updating the object version to the current one
    Model.UserVersion.Version =
      Model.DatabaseVersion.Version;
    Session.Merge(Model.UserVersion);
    tx.Commit();
  }
  Result = MergeResult.ForceDatabaseVersion;
  View.Close();
}
```

I use NHibernate's Merge capabilities to take all the persistent values in the user's version and copy them to the entity instance inside the current session. In effect, it merges the two instances, forcing the user values on top of the database value.

This is actually safe to do even with the other session dead and gone, because the Merge method contract ensures that it doesn't try to traverse lazy loaded associations.

Note that before I attempt the merge, I set the user's version property to the database's version property. This is done because in this case I want to explicitly overwrite that version.

This code doesn't attempt to handle recursive concurrency conflicts (that is, getting a concurrency conflict as a result of resolving the first one). That's left as an exercise for you.

Despite the number of challenges outlined in this article, building an NHibernate desktop application isn't any more difficult than building a NHibernate Web application. And in both scenarios, I believe that using NHibernate will make your life easier, the application more robust and the overall system easier to change and work with. ■

OREN EINI *(who works under the pseudonym Ayende Rahien) is an active member of several open source projects (NHibernate and Castle among them) and is the founder of many others (Rhino Mocks, NHibernate Query Analyzer and Rhino Commons among them). Eini is also responsible for the NHibernate Profiler (nhprof.com), a visual debugger for NHibernate. You can follow Eini's work at ayende.com/Blog.*

THANKS to the following technical experts for reviewing this article: *Howard Dierking*

Building a Visual Studio Team Explorer Extension

Brian A. Randell and Marcel de Vries

The main user interface you use to interact with Team Foundation Server (TFS) is the Team Explorer client. Team Explorer provides a way to access and view information on one or more TFS servers. By default, Team Explorer provides access to TFS features like work items, SharePoint document libraries, reports, builds, and source control.

Marcel's development teams use a number of separate Visual Studio solutions, each consisting of five to ten projects. In each of these solutions, there is a single point of deployment, maintenance, and design. Sometimes there are cross-solution dependencies. The team manages those dependencies via a special folder inside the TFS version control repository. Each solution writes its output to a well-known location and the team checks in the output after each build. A solution that depends on the output of another solution can reference the assembly when it is retrieved from TFS.

Unfortunately, Marcel's team faced a problem: Visual Studio stores file references using a relative path to the current solution, but developers want the flexibility of using their own directory structures and workspace mappings. This desirable flexibility results in frustration when Visual Studio can't build solutions because it can't find referenced files.

This article discusses:

- Building a VSPackage
- Defining context menus
- Handling commands
- VSPackage Registration

Technologies discussed:

Visual Studio Team System and Team Foundation Server 2008

Code download available at:

code.msdn.microsoft.com/mag200912VSTS2008Ext

One way to fix this problem is to map the location that contains the binary references as a substituted drive using the subst.exe command. By referencing the assemblies on the substituted drive, Visual Studio is able to find the files in a consistent location. Developers can put the files in the locations they prefer, then use subst.exe to map their locations to the standard mapping. Because the files reside on a different drive, Visual Studio stores a full path rather than a relative path. An additional benefit is that a developer can test a different version by simply changing the mapping.

While this technique works, even better would be a Team Explorer extension that allows a developer to define a mapping between the version control location and a mapped drive. Marcel's team implemented this functionality in a Team Explorer extension called Subst Explorer. You can see the menu for the Subst Explorer extension in see **Figure 1**.

Getting Started

To build your own Team Explorer plug-in, you'll need Visual Studio 2008 Standard Edition or higher, Team Explorer, and the Visual Studio 2008 SDK, which you can get from the Visual Studio Extensibility Developer Center (msdn.microsoft.com/vsx).

Before creating a Team Explorer plug-in, you need to create a VSPackage. The package supplies a class that implements the Microsoft.TeamFoundation.Common.ITeamExplorerPlugin interface defined in Microsoft.VisualStudio.TeamFoundation.Client.dll.

In Visual Studio parlance, your plug-in becomes a part of the hierarchy. One feature that differentiates the Team Explorer hierarchy from other hierarchy implementations in Visual Studio is that Team Explorer supports asynchronous loading of the hierarchy, essentially because loading the hierarchy for things like work items and documents often requires remote queries to TFS. These calls would otherwise block Visual Studio during those queries, resulting in a poor user experience. The ITeamExplorerPlugin interface implemented by your VSPackage provides the mechanism that Team Explorer uses to load the contents of each node asynchronously.

Create a VSPackage by selecting File | New | Project. In the New Project dialog, expand the Other Project Types node and select the Extensibility node. Over in the Templates pane, select the Visual Studio Integration Package template.

After you fill out the New Project dialog and click OK, Visual Studio launches the Visual Studio Integration Package wizard. First, choose your preferred language—C++, C#, or Visual Basic (C# is used in this article). When choosing Visual Basic or C#, you need to select the option to generate a new strong name key file or specify an existing file. On the next page, fill in information for your company and some details about the package, including the Package Name, icon, and so on. Much of this information is shown in the Help | About dialog in Visual Studio.

On the next page, select how you want Visual Studio to expose your package. For this particular example, you want to have a MenuCommand only. The plug-in will use this to handle the context menus. On the next page, you provide a Command Name and Command ID. Just accept the defaults since you'll change them later. On the next page you can add a support for test projects. We won't be covering them in this article, so feel free to deselect them and then finish the wizard. The wizard will generate the basic classes and resources you need for implementing the VSPackage.

Next, you need to add the following references, which provide access to the Team Explorer base classes used to create the Team Explorer plug-in:

```
Microsoft.VisualStudio.TeamFoundation.Client.(9.0.0)
Microsoft.VisualStudio.TeamFoundation (9.0.0)
Microsoft.VisualStudio.Shell (2.0.0)
```

You also need to remove the default reference to Microsoft.VisualStudio.Shell.9.0, since Microsoft built the Team Foundation assemblies against the 2.0 version of the assemblies instead of the 9.0 version. In addition, as generated, the project assumes it can use the regpkg.exe tool to register the package in the registry after compile. However, regpkg.exe depends on the Shell.9.0 assembly. To make the project build in Visual Studio 2008, you must change the project's .proj file. You need to unload the project file, and then add the following properties to the file under the RegisterOutputPackage property:

```
<!-- We are 2005 compatible, and don't rely on RegPkg.exe
of VS2008 which uses Microsoft.VisualStudio.Shell.9.0 -->
<UseVS2005MPF>true</UseVS2005MPF>
<!-- Don't try to run as a normal user (RANA),
create experimental hive in HKEY_LOCAL_MACHINE -->
<RegisterWithRanu>false</RegisterWithRanu>.
```

The Microsoft.VisualStudio.TeamFoundation.Client assembly provides a Microsoft.TeamFoundation.Common namespace that contains a base class called PluginHostPackage. Use this as the base class for your package. It also contains a base class called BasicAsyncPlugin that implements the required ITeamExplorerPlugin interface. You'll need to delete the default implementation of the generated Package class, then inherit from PluginHostPackage instead of the default Package class.

Because the class now inherits from PluginHostPackage, you only need to override the method OnCreateService. This method returns a new instance of a BasicAsyncPlugin derived class that manages the actual plug-in implementation. You can see the implementation of the HostPackage for the Subst Explorer in **Figure 2**. You'll also need to register your Team Explorer plug-in by hand, a task we'll return to later in the article.

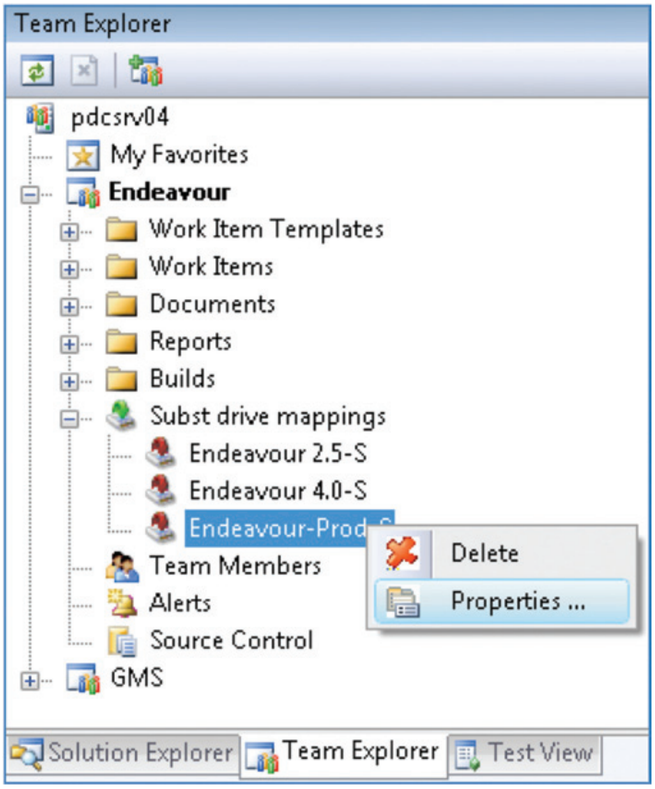


Figure 1 **The Subst Explorer Loaded in the Team Explorer Window**

In **Figure 2**, there are two attributes that are of special interest for the Team Explorer plug-in. ProvideService indicates this package provides a service, and the ServiceType is SubstExplorer. PluginRegistration

Figure 2 SubstExplorerPackage Implementation

```
...

[ProvideService(typeof(SubstExplorer))]
[PluginRegistration(Catalogs.TeamProject, "Subst explorer",
typeof(SubstExplorer))]
public sealed class SubstExplorerPackage: PluginHostPackage,
IVsInstalledProduct {
  private static SubstExplorerPackage _instance;
  public static SubstExplorerPackage Instance {
    get { return _instance; }
  }

  public SubstExplorerPackage () : base() {
    _instance = this;
  }

  protected override object OnCreateService(
    IServiceContainer container, Type serviceType) {

    if (serviceType == typeof(SubstExplorer)) {
      return new SubstExplorer();
    }
    throw new ArgumentException(serviceType.ToString());
  }
}
```

indicates that the package provides a Team Explorer plug-in and that additional registration is required. This attribute derives from RegistrationAttribute and regpkg.exe normally processes it.

Nodes and Hierarchies

As you can see in **Figure 2**, the implementation of OnCreateService is straightforward. It returns a new instance of the SubstExplorer class that provides the implementation of the BasicAsyncPlugin class. The SubstExplorer class is responsible for managing a part of the Team Explorer hierarchy. A hierarchy in Visual Studio is a tree of nodes where each node has a set of associated properties. Examples of other hierarchies in Visual Studio are the Solution Explorer, the Server Explorer, and the Performance Explorer.

The SubstExplorer manages the plug-in hierarchy by overriding two methods called CreateNewTree and GetNewUIHierarchy**.** In **Figure 3**, you can see the implementation of the SubstExplorer class that derives from BasicAsyncPlugin.

The SubstExplorer class manages the creation of a set of hierarchy nodes. For the SubstExplorer package, these nodes represent virtual folder locations that the plug-in can map as a drive. Each node contains the properties needed to map a drive using the subst.exe command. The package will track Name, Drive Letter, and Location (in the version control repository).

The package creates the tree in two steps. First, it creates the command handler class of all hierarchy nodes, better known as a UIHierarchy. The GetNewUIHierarchy method initiates this step. Second, the CreateNewTree method handles the creation of the tree of nodes that represent virtual drive mappings.

GetNewUIHierarchy is called from the UI thread and returns an instance of a class that derives from the base class BaseUIHierarchy. You'll find the package's implementation in the SubstExplorerUIHierarchy class. SubstExplorerUIHierarchy needs to handle all the Add, Delete, and Edit commands executed from any of the nodes the package adds to Team Explorer. The ExecCommand method class handles these commands. But first you need to create the menus and commands in Visual Studio.

In the SubstExplorer class, you override the CreateNewTree method that is called from a non-UI thread and returns the tree of nodes that represent all the drive substitutions configured for a team project. The tree always starts with a root node, derived from the RootNode class. For each definition, you'll add a child node to the root. The leaf node contains the properties you need to map a drive.

Commands and Properties

Now that you've seen the basic requirements to set up a Team Explorer plug-in, you need to add some functionality to it. The SubstExplorerRoot class derives from the RootNode class found in the Microsoft.TeamFoundation.Common assembly. Here you override the Icons, PropertiesClassName, and ContexMenu properties.

The Icons property returns an ImageList that contains the icons you want to use for displaying the nodes. In the constructor of the RootNode, you need to set the ImageIndex so that it points to the right image in the ImageList.

The PropertiesClassName returns a string that represents the name that Visual Studio displays in the properties grid window when you select a node. Any string you think is appropriate will suffice here.

Figure 3 **SubstExplorer implementation**

```
[Guid("97CE787C-DE2D-4b5c-AF6D-79E254D83111")]
public class SubstExplorer : BasicAsyncPlugin {
  public SubstExplorer() :
    base(MSDNMagazine.TFSPlugins.SubstExplorerHostPackage.Instance) {}

  public override String Name
  { get { return "Subst drive mappings"; } }

  public override int DisplayPriority {
    get {
      // After team explorer build, but before any installed power tools
      // power tools start at 450
      return 400;
    }
  }

  public override IntPtr OpenFolderIconHandle
  { get { return IconHandle; }}

  public override IntPtr IconHandle
  { get { return new Bitmap(
    SubstConfigurationFile.GetCommandImages().Images[2]).GetHicon(); } }

  protected override BaseUIHierarchy GetNewUIHierarchy(
    IVsUIHierarchy parentHierarchy, uint itemId) {

    SubstExplorerUIHierarchy uiHierarchy =
      new SubstExplorerUIHierarchy(parentHierarchy, itemId, this);
    return uiHierarchy;
  }

  protected override BaseHierarchyNode CreateNewTree(
    BaseUIHierarchy hierarchy) {
```

```
    SubstExplorerRoot root =
      new SubstExplorerRoot(hierarchy.ProjectName +
      '/' + "SubstExplorerRoot");
    PopulateTree(root);
    // add the tree to the UIHierarchy so it can handle the commands
    if (hierarchy.HierarchyNode == null)
      { hierarchy.AddTreeToHierarchy(root, true); }
    return root;
  }

  public static void PopulateTree(BaseHierarchyNode teNode) {
    string projectName =
      teNode.CanonicalName.Substring(0,
      teNode.CanonicalName.IndexOf("/"));
    var substNodes =
      SubstConfigurationFile.GetMappingsForProject(projectName);
    if (substNodes != null) {
      foreach (var substNode in substNodes) {
        SubstExplorerLeaf leafNode =
          new SubstExplorerLeaf(substNode.name, substNode.drive,
          substNode.versionControlPath);
        teNode.AddChild(leafNode);
      }
      // (bug workaround) force refresh of icon that changed
      // during add, to force icon refresh
      if (teNode.IsExpanded) {
        teNode.Expand(false);
        teNode.Expand(true);
      }
    }
  }
}
```

LEADTOOLS®

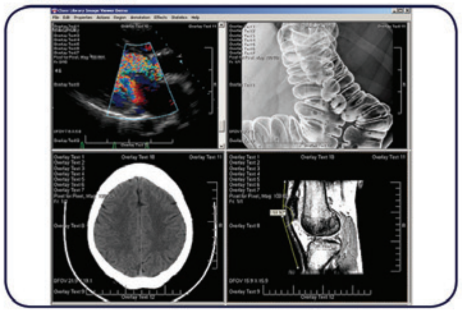
The World Leader in Imaging Development SDKs

Develop your application with the same robust imaging technologies used by **Microsoft, HP, Sony, Canon, Kodak, GE, Siemens,** the **US Air Force** and **Veterans Affairs Hospitals.**

LEADTOOLS provides developers easy access to decades of expertise in color, grayscale, document, medical, vector and multimedia imaging development. Install LEADTOOLS to eliminate months of research and programming time while maintaining high levels of quality, performance and functionality.

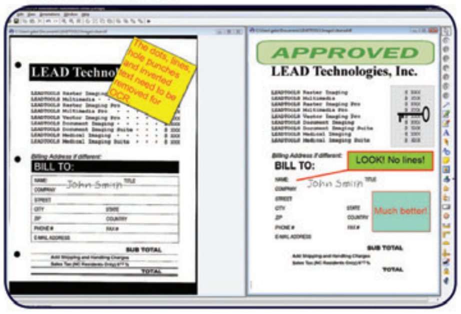
.NET, WPF, WCF, WF, C API, C++ Class Lib, COM & more!

Medical

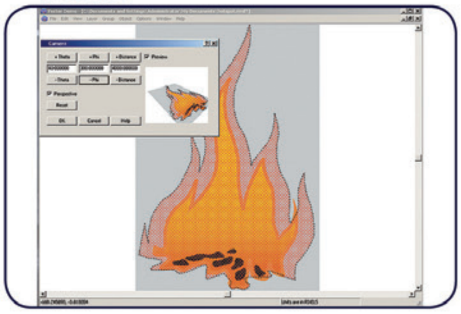


Multimedia

Document


LEAD Technologies, Inc.
The dots, lines, hole punches and inverted text need to be removed for OCR.
APPROVED
BILL TO:
LOOK! No lines!
Much better!

Vector



- •**Image Formats & Compression:** Supports 150+ image formats and compressions including TIFF, EXIF, PDF, JPEG2000, JBIG and CCITT.
- •**Display Controls:** ActiveX, COM, Win Forms, Web Forms, WPF and Silverlight.
- •**Image Processing:** 200+ filters, transforms, color conversion and drawing functions supporting region of interest and extended grayscale data.
- •**OCR/ICR/OMR:** Full page or zonal recognition for multithreaded 32 and 64 bit development.
- •**Forms Recognition and Processing:** Automatically identify forms and extract user filled data.
- •**Barcode:** Detect, read and write 1D and 2D barcodes for multithreaded 32 and 64 bit development.
- •**Document Cleanup/Preprocessing:** Deskew, despeckle, hole punch, line and border removal, inverted text correction and more.
- •**PDF and PDF/A:** Read and write searchable PDF with text, images and annotations.
- •**Annotations:** Interactive UI for document mark-up, redaction and image measurement (including support for DICOM annotations).
- •**Medical Web Viewer Framework:** Plug-in enabled framework to quickly build high-quality, full-featured, web-based medical image delivery and viewer applications.
- •**Medical Image Viewer:** High level display control with built-in tools for image mark-up, window level, measurement, zoom/pan, cine, and LUT manipulation.
- •**DICOM:** Full support for all IOD classes and modalities defined in the 2008 DICOM standard (including Encapsulated PDF/CDA and Raw Data).
- •**PACS Communications:** Full support for DICOM messaging and secure communication enabling quick implementation of any DICOM SCU and SCP services.
- •**JPIP:** Client and Server components for interactive streaming of large images and associated image data using the minimum possible bandwidth.
- •**Scanning:** TWAIN 2.0 and WIA (32 and 64-bit), autodetect optimum driver settings for high speed scanning.
- •**DVD:** Play, create, convert and burn DVD images.
- •**DVR:** Pause, rewind and fast-forward live capture and UDP or TCP/IP streams.
- •**Multimedia:** Capture, play, stream and convert MPEG, AVI, WMV, MP4, MP3, OGG, ISO, DVD and more.
- •**Enterprise Development:** Includes WCF services and WF activities to create scalable, robust enterprise applications.

Version 16.5 is a free upgrade for qualifying customers. To see if you qualify, or for more information on what is new in version 16.5, contact: **sales@LEADTOOLS.com**

Free 60 Day Evaluation! **800 637-1840** **www.leadtools.com/msdn**

LEAD TECHNOLOGIES INCORPORATED

High Level Design • Low Level Control

The LEADer in Intelligent Imaging

Figure 4 **SubstExplorerRoot**

```
public class SubstExplorerRoot : RootNode {
  static private readonly CommandID command =
    new CommandID(GuidList.guidPackageCmdSet,
    CommandList.mnuAdd);

  public SubstExplorerRoot(string path) : base(path) {
    this.ImageIndex = 2;
    NodePriority  = (int)TeamExplorerNodePriority.Folder;
  }

  public override System.Windows.Forms.ImageList Icons
  { get { return SubstConfigurationFile.GetCommandImages();  } }

  public override string PropertiesClassName {
    //Name of the node to show in the properties window
    get { return "Subst Explorer Root"; }
  }

  public override
    System.ComponentModel.Design.CommandID ContextMenu
  { get { return command; } }
}
```

The ContextMenu property returns a CommandID that represents the context menu you want to show. For the root node, you need a Context Menu with one option, called Add. **Figure 4** shows the implementation of the SubstExplorerRoot.

The leaf node class SubstExplorerLeaf (see **Figure 5**) derives from BaseHierarchyNode, and here you need to override the properties ContextMenu, PropertiesClassName and PropertiesObject. Youl also need to provide a custom implementation of DoDefaultAction. Visual Studio calls this method when you double-click a leaf node. DoDefaultAction executes the code that performs the Subst command. If you've previously executed the Subst command, it removes the mapping.

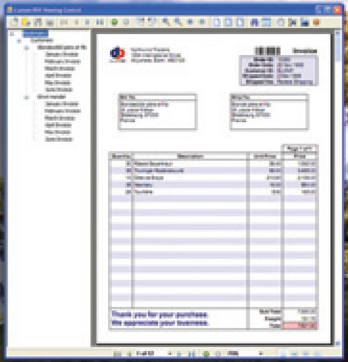
The ContextMenu property represents the context menu you want to show at the leaf node. The context menu exposes two commands: Properties and Delete. In the class, the PropertiesClassName has the same purpose as in the root node. You use the PropertiesObject property to get back an object that you can use to display the properties of the selected node in the properties window. For the leaf node, the properties exposed will be Name, DriveLetter, and VersionControlPath.

You return a new instance of the type SubstExplorerProperties (see **Figure 6**). You use this object to display the properties of the leaf node. SubstExplorerProperties provides an implementation of the ICustomTypeDescriptor interface that returns information on which properties you want to show and how you want to show

Figure 5 **SubstExplorerLeaf**

```
public class SubstExplorerLeaf : BaseHierarchyNode {
  private enum SubstIconId {
    unsubsted = 1,
    substed = 2
  }

  CommandID command =
    new CommandID(GuidList.guidPackageCmdSet,
    CommandList.mnuDelete);
  bool IsDriveSubsted { get; set; }

  public string VersionControlPath { get; set; }
  public string SubstDriveLetter { get; set; }

  public SubstExplorerLeaf(string path,
    string substDriveLetter, string versionControlPath)
    : base(path, path + " (" + substDriveLetter + ":)") {

    this.ImageIndex = (int)SubstIconId.unsubsted;
    this.NodePriority = (int)TeamExplorerNodePriority.Leaf;

    this.VersionControlPath = versionControlPath;
    this.SubstDriveLetter = substDriveLetter;
    this.IsDriveSubsted = false;
  }

  public override void DoDefaultAction() {
    if (!IsDriveSubsted) {
      SubstDrive();
    }
    else {
      UnsubstDrive(SubstDriveLetter);
    }
  }

  public override CommandID ContextMenu
  { get { return command;  } }

  public override string PropertiesClassName
  { get { return "Subst Leaf Node"; }}

  public override ICustomTypeDescriptor PropertiesObject {
    get {
      return new SubstExplorerProperties(this);
    }
  }

  private void SubstDrive() {
    if (IsDriveAlreadySubsted(SubstDriveLetter)) {
      UnsubstDrive(SubstDriveLetter);
    }
    string substresponse =
      SubstHelper.Subst(SubstDriveLetter, GetLocalFolder());

    if (string.IsNullOrEmpty(substresponse)) {
      IsDriveSubsted = true;
      this.ImageIndex = (int)SubstIconId.substed;
    }
    else {
      MessageBox.Show(string.Format(
        "Unable to make subst mapping. Message:\n {0}",
        substresponse));
    }
  }

  private bool IsDriveAlreadySubsted(string driveLetter) {
    bool IsdrivePhysicalyMaped =
      SubstHelper.SubstedDrives().Where(
      d => d.Contains(driveLetter + ":\\")).Count() != 0;
    bool IsdriveKnownToBeMaped =
      (from substedNode in _substedNodes
      where substedNode.SubstDriveLetter == driveLetter
      select substedNode).ToArray<SubstExplorerLeaf>().Length > 0;
    return IsdriveKnownToBeMaped || IsdrivePhysicalyMaped;
  }

  public void UnsubstDrive(string substDriveLetter) {
    string substResponse = SubstHelper.DeleteSubst(substDriveLetter);
    IsDriveSubsted = false;
    this.ImageIndex = (int)SubstIconId.unsubsted;
  }

  public string localPath {
    get { return VersionControlPath; }
  }
}
```

Dependable Developer Components



DynamicPDF Viewer

Our new, customizable DynamicPDF Viewer allows you to display PDF documents within any WinForm application. No longer rely on an external viewer for displaying your PDF documents. DynamicPDF Viewer utilizes the proven reliable and efficient Foxit PDF viewing engine and maximizes performance and compatibility with our other DynamicPDF products.

DynamicPDF Converter

Our DynamicPDF Converter library can efficiently convert over 30 document types (including HTML and all common Office file formats) to PDF. Events can be used to manage the action taken on a successful or failed conversion. It is highly intuitive and flexible and integrates well with our other DynamicPDF products.

DynamicPDF Rasterizer

Our DynamicPDF Rasterizer library can quickly convert PDF documents to over 10 common image formats including multi-page TIFF. Rasterizing form field values as well as annotations is fully supported. PDFs can also be rasterized to a System.Drawing.Bitmap class for further manipulation.

Try our three new products FREE today!

Fully functional and never expiring evaluation editions available at www.cete.com/download

To learn more about these or any of our other popular tools: DynamicPDF Generator, DynamicPDF Merger, DynamicPDF ReportWriter, DynamicPDF Suite, DynamicPDF WebCache or Firemail, visit us online.

ceTesoftware
INFINITE POSSIBILITIES

ceTe Software has been delivering quality software applications and components to our customers for over 10 years. Our DynamicPDF product line has proven our commitment to delivering innovative software components and our ability to respond to the changing needs of software developers. We back our products with a first class support team trained to provide timely, accurate and thorough responses to any support needs.

www.cete.com
info@cete.com

800.631.5006
+1 410.772.8620

them. BaseHierarchyNode has a default properties object that shows things like the URL, ServerName, and ProjectName, but that did not seem useful for our leaf node.

Commands and Menus

If you examine the root and leaf node implementations, you see that both need to display a context menu. The root node needs to have an Add menu item. A leaf node needs Delete and Properties menu items. Both node implementations return a CommandID instance as the implementation of their respective ContextMenu properties. In order for the CommandID class to function correctly, you need to define the menus and commands in the solution.

To add a menu and command to Visual Studio, you need to define the commands in a command table. You add command tables to the assembly as an embedded resource. In addition, you need to register the command table and the system registry during package registration. When you run devenv /setup, Visual Studio gathers all command resources from all registered packages and builds an internal representation of all commands in the development environment.

Starting with Visual Studio 2005, you could define command tables in an XML file with the extension .vsct. In this file, you define the menus, the command groups, and the buttons you want to show in the menu. A Visual Studio command is part of a command group. You place command groups on menus.

For the root node, you need an Add command, placed in a group contained by a menu. The leaf node needs Delete and Properties commands. You need to define a second menu that contains a different group that contains these two commands. (See the download accompanying this article for an example .vsct file.)

The .vsct file needs special treatment in the Visual Studio project. You must compile it into a resource and then embed the resource in the assembly. After you install the Visual Studio SDK, you can select a special build action for your command file called VSCTCompile. This action takes care of compiling and embedding the resource in the assembly.

In the command table XML, some symbols are used in the definition of the menus and commands. You add all menus, commands, and buttons to the same commandSet called GuidPackageCmdSet:

```
<Symbols>
  <!-- This is the package guid. -->
  <GuidSymbol name="GuidPackage" value=
"{9B024C14-2F6F-4e38-AA67-3791524A807E}"/>
  <GuidSymbol name="GuidPackageCmdSet" value=
"{D0C59149-AC1D-4257-A68E-789592381830}"/>
    <IDSymbol name="mnuAdd" value="0x1001" />
    <IDSymbol name="mnuDelete" value="0x1002" />
```

Everywhere you need to provide context menu information, you refer back to this symbol as the container of the menu. Thus, in the SubstExplorerRootNode and SubstExplorerLeafNode implementations, you create an instance of the CommandID type and use GuidPackageCommandSet as the first argument and the actual menu you want to display as the second argument:

```
CommandID command = new CommandID(
  GuidList.guidPackageCmdSet,
  CommandList.mnuDelete);
```

In the .vsct file, there are three commands that the UIHierarchy needs to respond to. The ExecCommand method is called when you click one of the menu items. The method needs to select the action to execute based on the nCmdId passed to it. The basic implementation of the SubstExplorerUIHierarchy is shown in **Figure 7**.

Add, Edit and Delete

Now you need to provide a way for the user to add, delete, or edit mappings on the root or leaf nodes. The code is in place to handle calls for Add on the root node and for the Edit and Delete commands on the leaf nodes. Adding a new mapping requires input from the user and

Figure 6 **SubstExplorerProperties**

```
public class SubstExplorerProperties
  : ICustomTypeDescriptor, IVsProvideUserContext {

  private BaseHierarchyNode m_node = null;
  public SubstExplorerProperties(BaseHierarchyNode node)
    { m_node = node; }

  public string GetClassName()
    { return m_node.PropertiesClassName;}

  public string GetComponentName()
    { return m_node.Name; }
  public PropertyDescriptorCollection
    GetProperties(Attribute[] attributes) {

    // create for each of our properties the
    // appropriate PropertyDescriptor
    List<PropertyDescriptor> list = new List<PropertyDescriptor>();
    PropertyDescriptorCollection descriptors =
      TypeDescriptor.GetProperties(this, attributes, true);

    for (int i = 0; i < descriptors.Count; i++) {
      list.Add(new DesignPropertyDescriptor(descriptors[i]));
    }
    return new PropertyDescriptorCollection(list.ToArray());
  }

  public object GetPropertyOwner(PropertyDescriptor pd) {
    // return the object implementing the properties
    return this;
  }

  // rest of ICustomTypeDescriptor methods are not
  // shown since they are returning defaults
  // actual properties start here
  [Category("Drive mapping")]
  [Description("...")]
  [DisplayName("Version Control Path")]
  public string VersionControlPath
    {  get { return ((SubstExplorerLeaf)m_node).VersionControlPath; } }

  [Category("Drive mapping")]
  [Description("...")]
  [DisplayName("Subst drive letter")]
  public SubstDriveEnum SubstDriveLetter {
    get { return
      (SubstDriveEnum)Enum.Parse(typeof(SubstDriveEnum),
      ((SubstExplorerLeaf)m_node).SubstDriveLetter); }
  }

  [Category("Drive mapping")]
  [Description("...")]
  [DisplayName("Mapping name")]
  public string MappingName
    {  get { return ((SubstExplorerLeaf)m_node).Name; } }
}
```

Figure 7 **SubstExplorerUIHierarchy**

```
public class SubstExplorerUIHierarchy : BaseUIHierarchy,
  IVsHierarchyDeleteHandler, IVsHierarchyDeleteHandler2 {

  public SubstExplorerUIHierarchy(IVsUIHierarchy parentHierarchy,
    uint itemId, BasicAsyncPlugin plugin)
    : base(parentHierarchy, itemId, plugin,
    MSDNMagazine.TFSPlugins.SubstExplorerHostPackage.Instance) {
  }

  public override int ExecCommand(uint itemId,
    ref Guid guidCmdGroup, uint nCmdId,
    uint nCmdExecOpt, IntPtr pvain, IntPtr p) {

    if (guidCmdGroup == GuidList.guidPackageCmdSet) {
      switch (nCmdId) {
        case (uint)CommandList.cmdAdd:
             AddNewDefinition(this.ProjectName);
             return VSConstants.S_OK;
        case (uint)CommandList.cmdDelete:
             RemoveDefinition(itemId);
             return VSConstants.S_OK;
        case (uint)CommandList.cmdEdit:
             EditDefinition(itemId);
             return VSConstants.S_OK;
        default: return VSConstants.E_FAIL;
      }
    }

    return base.ExecCommand(itemId, ref guidCmdGroup, nCmdId,
nCmdExecOpt, pvain, p);
  }
  ...
}
```

you need to store the mapping data in a well-known location. This location is preferably in the user's roaming profile. So let's take a closer look on how you can respond to the Add command.

The AddNewDefinition method in the SubstExplorerUIHierarchy class handles the Add command. AddNewDefinition shows a dialog allowing users to specify the mapping they want to create. A mapping needs to have a name and a drive letter for the Subst command. In addition, the mapping needs to point to a path in the version control repository. You want to allow the user to pick the location from version control rather than having to enter a complex path manually. You can enable this by using the TFS object model, specifically the GetServer method from the TeamFoundationServerFactory class. GetServer accepts a URL representing the server you want to use and a credentialsProvider in case the user is not in the same domain as the server and the server connection requires new authentication. After you have access to a valid TeamFoundationServer instance, you have access to the various services provided by TFS.

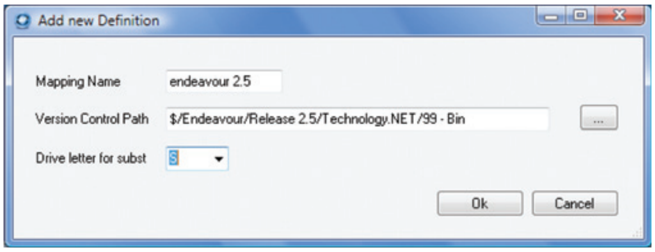
You need the VersionControlServer service to get information about the folder structure inside the current team project. In Brian's January 2007 Team System column (msdn.microsoft.com/magazine/cc163498), he showed how you could use this service to create your own version control folder browser dialog. We've reused the dialog described in that article here (see **Figure 8**). The dialog returns the folder selected by the user in the Version Control repository as shown in **Figure 9**. You store the path returned in a configuration file.

When the user clicks OK, you can add a new node to the configuration file and a new child node to the hierarchy. You add a new node by calling the AddChild method on the Hierarchy Node instance.

Executing the Default Command

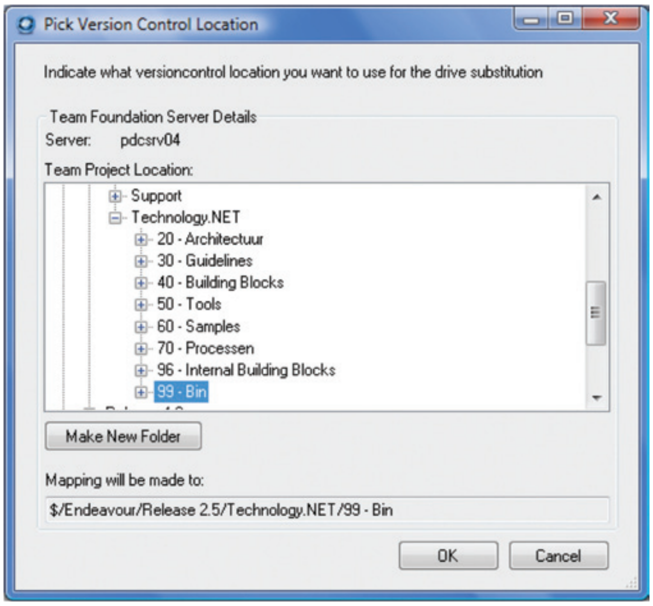
The SubstExplorerUIHierarchy class is responsible for handling all commands fired by the menu options offered by the plug-in. One of the other commands you need to handle is when a user double-clicks on a node. The DoDefaultAction method processes this event. For the root node, the default action of either collapsing or expanding the nodes in the hierarchy is acceptable. However, for leaf nodes, you'll provide a custom implementation.

You want to substitute the drive based on the properties set for that node. To subst a drive, you can issue a command-line action and provide the required parameters. For that purpose, we created a SubstHelperClass that calls into the System.Diagnostics namespace to create a new process called subst.exe and provide it with the required parameters. The parameters needed are the drive letter and the local folder you want to map as the drive. You have the drive letter available. However, you need to map the version control path to local folder. Once again, you'll use the TFS object model and get a reference to the VersionControlServer object. You can query this object for all available workspaces and try to get a mapping to a local folder based on the version control path you have. **Figure 10** provides an implementation.



Mapping Name: endeavour 2.5
Version Control Path: $/Endeavour/Release 2.5/Technology.NET/99 - Bin
Drive letter for subst: S

Figure 8 **Adding a New Mapping Definition**



Indicate what versioncontrol location you want to use for the drive substitution

Server: pdcsrv04

Mapping will be made to:
$/Endeavour/Release 2.5/Technology.NET/99 - Bin

Figure 9 **Choosing a Location in Version Control**

Finishing Touches

Now you have all the logic in place to show a tree of nodes and handle drive mapping. However, you want your Team Explorer plug-in to stand out. You might want to add some additional features in terms of delete node handling and other professional touches, such as adding an icon in the splash screen of Visual Studio.

Adding the delete functionality requires you to implement an additional interface in the SubstExplorerUIHierarchy class. Visual Studio has a specific interface called IVsHierarchyDeleteHandler that you implement to show a default dialog when you press the delete key. For this plug-in, you'll want to provide a custom dialog asking the user to confirm the deletion of the node that is selected. To make that work, you also need to implement the IVsHierarchyDeleteHandler2 interface for delete handling to work from the keyboard. Since you've already implemented the actual delete functionality, you need only to implement this interface and call the existing functions. In **Figure 11** you can see the implementation of the interfaces.

It's important to note that the plug-in does not support multiple selected nodes being deleted at once, hence pfCancelOperation is set to true in the ShowMultiSelDeleteOrRemoveMessage method. In the ShowSpecificDeleteRemoveMessage method implementation, you need to return the correct value of what you want to delete. You return a value of 1 to indicate you have removed it from storage. These flags are normally used in the Visual Studio project system and only a value of 1 produces the correct results.

You might also want to add support for splash screen integration. By default, each time you start Visual Studio, you'll see a splash screen listing the products registered. You accomplish this by implementing the IVsInstalledProduct interface in the SubstExplorerHostPackage implementation class. The methods there require you register the resource IDs for the icon to use in the splash screen and the icon to use in the About box.

The implementation is nothing more than setting the out parameter to the correct integer value and embedding a 32x32 pixel icon as a resource in the assembly. In order to embed the resource correctly in your assembly, you need to open up the resources.resx file in the XML editor and add the following lines to the resource file:

```
<data name="500"
  type="System.Resources.ResXFileRef, System.Windows.Forms">
  <value>..\Resources\SplashIcon.bmp;System.Drawing.Bitmap,
    System.Drawing, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a</value>
</data>
```

This adds the resource bitmap located in the Resources folder in the project to the resource and embeds it at the reference 500. In the method IdBmpSplash, you can now set pIdBmp to 500 and return the S_OK constant as a return value. To get the icon in the splash screen, you need to build the assembly and then run devenv /setup from the command line. This will get the information from the package you've created and it will cache the data. This ensures the package does not need to be loaded when Visual Studio shows the splash screen. You do this for the same reasons you needed to do so for the menu options you added: to speed up the load time of Visual Studio.

Figure 10 **Mapping a Version Control Path to a Location on Disk**

```
private string GetLocalFolder() {
  VersionControlServer vcs =
    (VersionControlServer)((
    SubstExplorerUIHierarchy)ParentHierarchy).
    tfs.GetService(typeof(VersionControlServer));
  Workspace[] workspaces =
    vcs.QueryWorkspaces(null, vcs.AuthenticatedUser,
    Environment.MachineName);
  foreach (Workspace ws in workspaces) {
    WorkingFolder wf =
      ws.TryGetWorkingFolderForServerItem(VersionControlPath);
    if (wf != null) {
      // We found a workspace that contains this versioncontrolled item
      // get the local location to map the drive to this location....
      return wf.LocalItem;
    }
  }
  return null;
}
```

Figure 11 **IVsHierarchyDelete Handler implementation**

```
#region IVsHierarchyDeleteHandler2 Members

public int ShowMultiSelDeleteOrRemoveMessage(
  uint dwDelItemOp, uint cDelItems,
  uint[] rgDelItems, out int pfCancelOperation) {

  pfCancelOperation = Convert.ToInt32(true);
  return VSConstants.S_OK;
}

public int ShowSpecificDeleteRemoveMessage(
  uint dwDelItemOps, uint cDelItems, uint[] rgDelItems,
  out int pfShowStandardMessage, out uint pdwDelItemOp) {

  SubstExplorerLeaf nodeToDelete =
    NodeFromItemId(rgDelItems[0]) as SubstExplorerLeaf;
  if (AreYouSureToDelete(nodeToDelete.Name)) {
    pdwDelItemOp = 1; // == DELITEMOP_DeleteFromStorage;
                      // DELITEMOP_RemoveFromProject==2;
  }
  else {
    pdwDelItemOp = 0; // NO delete, user selected NO option }

  pfShowStandardMessage = Convert.ToInt32(false);
  return VSConstants.S_OK;
}

#endregion
#region IVsHierarchyDeleteHandler Members

public int DeleteItem(uint dwDelItemOp, uint itemid) {
  SubstExplorerLeaf nodeToDelete =
    NodeFromItemId(itemid) as SubstExplorerLeaf;
  if (nodeToDelete != null) {
    // remove from storage
    RemoveDefinitionFromFile(nodeToDelete);
    // remove from UI
    nodeToDelete.Remove();
  }
  return VSConstants.S_OK;
}

public int QueryDeleteItem(uint dwDelItemOp, uint itemid,
  out int pfCanDelete) {

  pfCanDelete = Convert.ToInt32(NodeFromItemId(itemid) is
SubstExplorerLeaf);
  return VSConstants.S_OK;
}
#endregion
```

Package Registration

Now that you've finished the Team Explorer extension, it's time to package the product and get it running on other developer's systems. So, how can you distribute the results?

First, Visual Studio behaves differently when you've installed the SDK. By default it will accept or load any VSPackage. This will not be the case on machines where you haven't installed the SDK.

For a package to load correctly, you need to embed a package load key, which is obtained from Microsoft.

For a package to load correctly, you need to embed a package load key, which you obtain from Microsoft (see msdn.microsoft.com/vsx/cc655795). The most important part of this process is to ensure that you provide the exact same information when registering for your key as the information you provided in the attributes for the hostPackage class (in this case the SubstExplorerHostPackage class). Also, when the Web site asks you to enter the package name, you must provide the product name you used in the ProvideLoadKey attribute.

Once you get your load key, you can paste it into the resource file with the resource identifier you provided as last argument of the ProvideLoadKey attribute. Make sure you remove the carriage return/line feeds from the string when you copy it from the site so it is one consecutive string before you paste it in the resource file.

Now you can test if your plug-in works by specifying an additional debug parameter: /NoVsip. This parameter ensures that Visual Studio uses the normal loading behavior. If the key is not accepted, Visual Studio will display a load failure dialog. With the SDK installed, you'll find under the Visual Studio Tools menu the Package Load Analyzer. You can point this at your assembly to help debug what is wrong. If it is only the package load key, then ensure you have typed exactly the same parameters at the Web site as in your attribute.

The last step that remains is the package registration for a production machine. Unfortunately, because the Team System assemblies use a different version of the shell assemblies, you cannot use regpkg.exe to register your package. Instead, you need to do it by hand using a registry file. In this file, you need to publish the package in the correct registry location. The registration script required is shown in **Figure 12**.

In the registration script, you'll see a number of entries. The first entry registers a new Team Explorer extension that Team Explorer should load as soon as it loads. Here you provide a registry value that refers to the service ID that provides an implementation of ITeamExplorerPlugin. The next entry provides the service registration where you see the previously referred-to service ID, as well as a registry value that points to the package that provides the plug-in.

The next entry is the package registration itself. There you use the package ID as a new key and provide the information where the assembly can

Figure 12 **Package Registration Script**

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\9.0\
TeamSystemPlugins\Team Explorer Project Plugins\SubstExplorer]
@="97CE787C-DE2D-4b5c-AF6D-79E254D83111"
"Enabled"=dword:00000001

[HKEY_LOCAL_MACHINE\Software\Microsoft\VisualStudio\9.0\Services\
{97ce787c-de2d-4b5c-af6d-79e254d83111}]
@="{9b024c14-2f6f-4e38-aa67-3791524a807e}"
"Name"="SubstExplorer"


[HKEY_LOCAL_MACHINE\Software\Microsoft\VisualStudio\9.0\Packages\
{9b024c14-2f6f-4e38-aa67-3791524a807e}]
@="MSDNMagazine.TFSPlugins.SubstExplorerHostPackage, TFSSubstExplorer,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=324c86b3b5813447"
"InprocServer32"="C:\\Windows\\system32\\mscoree.dll"
"Class"="MSDNMagazine.TFSPlugins.SubstExplorerHostPackage"
"CodeBase"="c:\\program files\\msdnsamples\\TFSSubstExplorer.dll"
"ID"=dword:00000065
"MinEdition"="Professional"
"ProductVersion"="1.0"
"ProductName"="SubstExplorer"
"CompanyName"="vriesmarcel@hotmail.com"

[HKEY_LOCAL_MACHINE\Software\Microsoft\VisualStudio\9.0\
InstalledProducts\SubstExplorerHostPackage]
"Package"="{9b024c14-2f6f-4e38-aa67-3791524a807e}"
"UseInterface"=dword:00000001

[HKEY_LOCAL_MACHINE\Software\Microsoft\VisualStudio\9.0\Menus]
"{9b024c14-2f6f-4e38-aa67-3791524a807e}"=", 1000, 1"
```

be found, how it can be loaded using the COM infrastructure, and what Visual Studio version the package supports. The last two entries are the registration of the installed products, used for the splash screen. Here the UseInterface key indicates Visual Studio must call the IVsInstalledProduct interface instead of relying on the InstalledProductRegistration attribute to provide an icon and product description that needs to be shown at startup.

The last entry is the registration of the context menus. Here you refer back to your package, but you also provide information about where you've embedded the resources in the assembly. These are the embedded resources you created before using the .vsct files and the custom build action on that file. With this script and the assembly you've built, you can deploy it on other machines. Just place the assembly on the file system, tweak the registry script to reflect the correct assembly location, and merge it into the registry. Then, the final step is to run devenv /setup on that machine. When you start Visual Studio, you will see the icon in the splash screen and when you load the Team Explorer, you will see the root node of the plug-in you've created. ■

BRIAN A. RANDELL *is a senior consultant with MCW Technologies LLC. Brian spends his time speaking, teaching, and writing about Microsoft technologies. He is the author of Pluralsight's Applied Team System course and is a Microsoft MVP. You can contact Brian via his blog at mcwtech.com/cs/blogs/brianr.*

MARCEL DE VRIES *is an IT architect at Info Support in the Netherlands, where he creates solutions for large bank and insurance companies across the country and teaches the Team System and Windows Workflow courses. Marcel is a frequent speaker at developer conferences in Europe, a Team System MVP since 2006, and a Microsoft regional director since January 2009.*

THANKS to the following technical experts for reviewing this article: *Dennis Habib and Buck Hodges*

K. SCOTT ALLEN

Looking Ahead to ASP.NET 4.0

When Visual Studio 2010 and .NET 4.0 arrive next year, we ASP.NET developers will have two mature frameworks for building Web applications: the ASP.NET Web Forms framework and the ASP.NET MVC framework. Both build on top of the core ASP.NET runtime, and both are getting some new features to start the next decade.

I don't have the space to cover every addition to ASP.NET in one article, as there are numerous improvements to both frameworks and the underlying runtime. Instead, I'll highlight what I think are the important new features for Web Forms and MVC.

New For ASP.NET Web Forms

ASP.NET Web Forms will be over eight years old by the time Microsoft releases Version 4, and the team continues to refine the framework and make improvements. In my last column, I touched on a few of these improvements, like the new classes that make it easy to use the URL routing features now included in the core services of ASP.NET, and the new MetaKeywords and MetaDescription properties on the Page base class that make it simple to control the content of metatags on a form. These changes are relatively minor, however.

The key changes in Web Forms address some of the chief criticisms about the framework. Many developers have wanted more control over the HTML a Web form and its controls produce, including the client-side identifiers emitted inside the HTML. In 4.0, many of ASP.NET's server-side controls were reworked to produce HTML that is easier to style with CSS and conforms to conventional Web practices. Also, new properties have been added to base classes that will give developers more control over the client-side identifiers generated by the framework. I'll highlight these changes in the following sections.

CSS-Friendly HTML

One example of a server control that is notoriously difficult to style with CSS is the ASP.NET menu control. When the menu renders, it emits nested table tags that include cellpadding, cellspacing and border attributes. To make matters worse, the menu control embeds style information inside the cells of the nested tables and injects an in-line style block at the top of the page. As an example, look at the following definition of a simple menu:

```
<asp:Menu runat="server" ID="_menu">
    <Items>
        <asp:MenuItem Text="Home" NavigateUrl="~/Default.aspx" />
        <asp:MenuItem Text="Shop" NavigateUrl="~/Shop.aspx" />
    </Items>
</asp:Menu>
```

In ASP.NET 3.5, the simple menu produces the following HTML (with some attributes omitted or shortened for clarity):

```
<table class="..." cellpadding="0" cellspacing="0" border="0">
    <tr id="_menun0">
        <td>
            <table cellpadding="0" cellspacing="0"
                border="0" width="100%">
                <tr>
                    <td style="...">
                        <a class="..." href="Default.aspx">Home</a>
                    </td>
                </tr>
            </table>
        </td>
    </tr>
</table>
```

In ASP.NET 4.0, Microsoft revised the menu control to produce semantic markup. The same menu control in ASP.NET 4.0 will produce the following HTML:

```
<div id="_menu">
    <ul class="level1">
        <li><a class="level1" href="Default.aspx" target="">Home</a></li>
    </ul>
</div>
```

This type of CSS-friendly markup was achievable in previous versions of ASP.NET if you used a control adapter to provide alternate rendering logic for a control, but now the markup is CSS-friendly by default. If you already have style sheets and client script written against the HTML produced by ASP.NET 3.5, you can set the controlRenderingCompatibilityVersion attribute of the pages section in web.config to the value "3.5", and the control will produce the nested table markup we saw earlier. The default value for this attribute is 4.0. Note that the 4.0 menu control still produces a style block at the top of the page, but you turn this off by setting the IncludeStyleBlock property of the control to false.

Many other controls in 4.0 are CSS-friendly as well. For example, validation controls like the RangeValidator and RequiredFieldValidator will no longer render inline styles, and template controls like the FormView, Login, and Wizard control will no longer render themselves inside of a table tag (but only if you set the RenderOuterTable property on these controls to false). Other

This article is based on a prerelease version of ASP.NET 4.0. Details are subject to change.

Send your questions and comments to xtrmasp@microsoft.com.

controls have changed, too. As just one example, you can force the RadioButtonList and CheckBoxList controls to render their inputs inside of list elements by setting the RepeatLayout property to the value OrderedList or UnorderedList, which forces the controls to render using ol and li elements, respectively.

Generating Client IDs

If you have ever written client-side script to manipulate the DOM, then you are probably aware of ASP.NET's affinity for changing client-side ID attributes. In an effort to ensure that all ID attributes are unique on a page, ASP.NET will generate a client ID by concatenating a control's ID property with additional information. On the server, you can access the generated value using the ClientID property of a control.

As an example, if a control is inside a naming container (a control that implements the INamingContainer interface, as user controls and master pages do), then ASP.NET produces the ClientID value by prefixing the naming container's ID to the control's ID. For data-bound controls that render repeating blocks of HTML, ASP.NET will add a prefix that includes sequential numbers. If you view the source of any ASP.NET page, you'll probably encounter id values like "ctl00_content_ctl20_ctl00_loginlink". These generated values add an extra level of difficulty when writing client script for a Web Forms page.

In Web Forms 4.0, a new ClientIDMode property is on every control. You can use this property to influence the algorithm ASP.NET will use for generating the control's ClientID value. Setting the value to Static tells ASP.NET to use the control's ID as its ClientID, with no concatenation or prefixing. For example, the CheckBoxList in the following code will generate an <ol> tag with a client id of "checklist", regardless of where the control exists on the page:

```
<asp:CheckBoxList runat="server" RepeatLayout="OrderedList"
                  ID="checklist" ClientIDMode="Static">
    <asp:ListItem>Candy</asp:ListItem>
    <asp:ListItem>Flowers</asp:ListItem>
</asp:CheckBoxList>
```

When using a ClientIDMode of Static, you'll need to ensure the client identifiers are unique. If duplicated id values exist on a page, you'll effectively break any scripts that are searching for DOM elements by their ID value.

There are three additional values available for the ClientIDMode property. The value Predictable is useful for controls implementing IDataBoundListControl, like the GridView and ListView. Use the Predictable value in conju ˆ nction with the ClientIDRowSuffix property of these controls to generate client IDs with specific values suffixed to the end of the ID. For example, the following ListView will bind to a list of Employee objects. Each object has EmployeeID and IsSalaried properties. The combination of the ClientIDMode and ClientIDRowSuffix properties tell the CheckBox to generate a client ID like employeeList_IsSalaried_10, where 10 represents the associated employee's ID.

```
<asp:ListView runat="server" ID="employeeList"
              ClientIDMode="Predictable"
              ClientIDRowSuffix="EmployeeID">
    <ItemTemplate>
        <asp:CheckBox runat="server" ID="IsSalaried"
                      Checked=<%# Eval("IsSalaried") %> />
    </ItemTemplate>
</asp:ListView>
```

Another possible value for ClientIDMode is Inherit. All controls on a page use a ClientIDMode of Inherit by default. Inherit means the control will use the same ClientIDMode as its parent. In the previous code sample, the CheckBox inherits its ClientIDMode value from the ListView, which holds the value Predictable. The final possible value for ClientIDMode is AutoID. AutoID tells ASP.NET to use the same algorithm for generating the ClientID property as it does in Version 3.5. The default value for a page's ClientIDMode property is AutoID. Since all controls on a page default to using a ClientIDMode of Inherit, moving an existing ASP.NET application to 4.0 will not change the algorithm the runtime uses to generate client ID values until you make a change to a ClientIDMode property. This property can also be set in the pages section of web.config to provide a different default for all pages in an application.

New Project Template

The Web application and Web site project templates in Visual Studio 2008 provide a Default.aspx page, a web.config file and an App_Data folder. These starting templates are simple and require some additional work before you can get started on a real application. The same templates in Visual Studio 2010 provide more of the infrastructure you need to build an application using contemporary practices. A screen capture of a brand new application produced by these templates is shown in **Figure 1**.

Notice how the new application includes a master page by default (Site.master). All of the .aspx files you find inside the new project will be content pages using ContentPlaceholder controls to plug content into the structure defined by the master page. Notice the new project also includes a style sheet in the Content directory (Site.

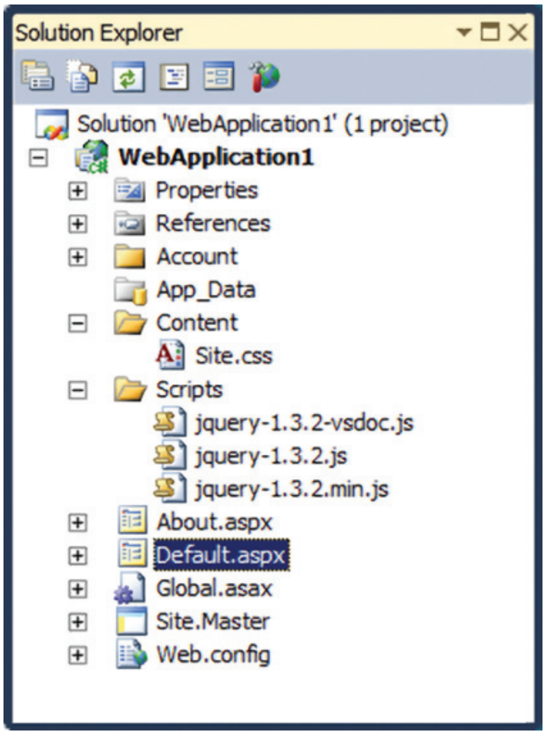
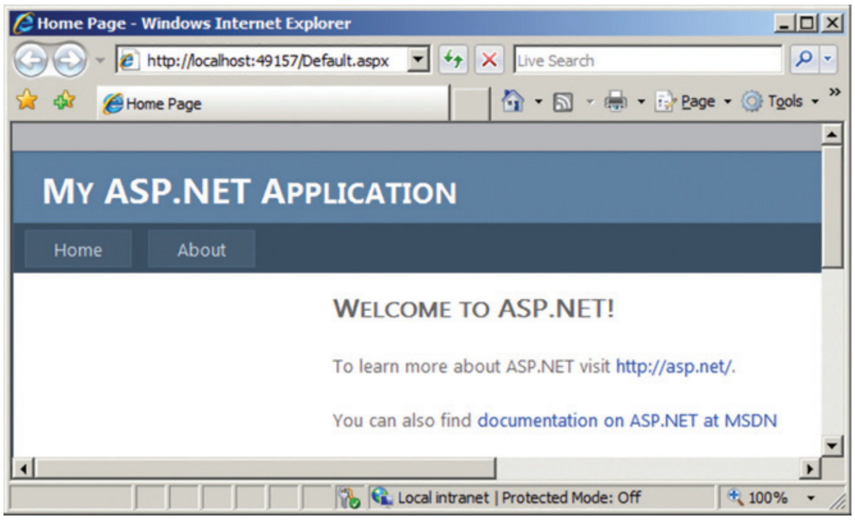


Figure 1 **New Web Application in Visual Studio 2010**

css). The master page includes this style sheet using a link tag, and inside the style sheet you'll find a number of styles defined to control the appearance of the page body, headings, primary layout and more. The new project also includes a Scripts directory with the latest version of the jQuery library, an open source JavaScript framework officially supported by Microsoft and included with Visual Studio 2010 as part of the install.

The new project template, with its use of master pages and style sheets, will help developers get started in the right direction when using Web Forms. A running version of the new application is shown in **Figure 2**. Visual Studio 2010 will also include "Empty" templates for both Web sites and Web applications. These empty templates will not include files or directories when you use them, so you'll be starting your application from scratch.

Another bit of good news about new projects in ASP.NET 4.0 is that the web.config file starts off nearly empty. Most of the configuration we became accustomed to seeing in ASP.NET 3.5 web.config files is now in the machine.config file that lives underneath the 4.0 framework's installation directory. This includes the configuration of controls from the System.Web.Extensions directory, the HTTP handlers and modules configured to support JavaScript proxies for Web services, and the system.webserver section for Web sites running under IIS 7.


Home Page - Windows Internet Explorer
http://localhost:49157/Default.aspx
Live Search
Home Page
Page
Tools
Home
About
Local intranet | Protected Mode: Off
100%

MY ASP.NET APPLICATION

WELCOME TO ASP.NET!

To learn more about ASP.NET visit http://asp.net/.

You can also find documentation on ASP.NET at MSDN

Figure 2 **Running the new ASP.NET Application**

New for ASP.NET MVC

Visual Studio 2010 should bring us the second release of the ASP.NET MVC framework. While still young, the framework has attracted many Web developers who wanted a framework designed for testability. The second release of ASP.NET MVC is going to focus on better developer productivity and adding the infrastructure to handle large, enterprise-scale projects.

Areas

One approach to building an extremely large ASP.NET Web Forms application is to break apart the application into multiple sub-projects (an approach promoted by the P&P Web Client Composite Library). This approach is difficult to undertake with ASP.NET MVC 1.0 because it works against a number of the MVC conventions. MVC 2.0 will officially support this scenario using the concept of an *area*. An *area* allows you to partition an MVC application across Web application projects, or across directories inside of a single project. Areas help to separate logically different pieces of the same application for better maintainability.

The parent area of an MVC Web application is an MVC project that will include a global.asax and root level web.config file for the application. The parent area can also include common pieces of content, like application-wide style sheets, JavaScript libraries, and master pages. Child areas are also MVC Web application projects, but since these projects physically exist underneath the parent area project at runtime, the parent and its children will appear as a single application.

As an example, imagine a large inventory application. In addition to the parent area, the inventory application might be broken into ordering, distributing, reporting and administrative areas. Each area can live in a separate MVC web project, and each project will need to register its routes by including a class that derives from the abstract base class AreaRegistration. In the code that follows, we override the AreaName property to return the friendly name of the reporting area, and override the RegisterArea method to define the routes available in the reporting area:

```
public class ReportingAreaRegistration : AreaRegistration
{
    public override string AreaName
    {
        get { return "Reporting"; }
    }

    public override void RegisterArea(AreaRegistrationContext context)
    {
        context.MapRoute(
            // route name
            "ReportingDefault",
            // url pattern
            "reporting/{controller}/{action}",
            // route defaults
            new { controller = "Home", action = "Index" },
            // namespaces
            new string[] { "Reporting.Controllers" });
    }
}
```

Notice that we include a string array of namespaces to search when locating the controller for the reporting area. Constraining the namespaces to search allows different areas to have controllers with the same name (multiple HomeController classes can exist in the application, for example).

DataAnnotations for Easy Validation

The DefaultModelBinder in ASP.NET MVC is responsible for moving data from the request environment into model proper-

Get the most comprehensive file coverage from any of Aspose's 4 Award Winning Suites

Aspose provides extensive file format processing capabilities for some of the most popular file formats including:

•DOC	•DOCX	•WordML	•OOXML	•ODF	•RTF	•SpreadsheetML	•XLS
•XLSX	•CSV	•TAB	•PDF	•FO	•PPT	•POT	•SWF

•& More...

The ever expanding list of features now includes the ability to Read, Write, Convert, Print, View, and Render for a variety of platforms including .NET, Java, SQL Server and JasperReports. Extensitve File Format Processing coupled with unmatched performance, reliability and an award winning customer support makes sure you perform... beyond the rest!

Logon to www.aspose.com
and get your free evaluation copy right now!

ASPOSE®
The .NET & Java Component Publisher™

ties. For example, when the model binder sees a model object with a property named Title, it will look through the form, query string and server variables to find a variable with a matching name (Title). However, the model binder doesn't perform any validation checks beyond simple type conversions. If you want the Title property of your model object to contain only strings with 50 characters or less, you have to perform this validation check during the execution of your controller action, implement a custom model binder or implement the IDataErrorInfo interface on your model.

In ASP.NET MVC 2.0, the DefaultModelBinder will look at DataAnnotation attributes on model objects. These DataAnnotation attributes allow you to provide validation constraints on your model. As an example, consider the following Movie class:

```
public class Movie
{
    [Required(ErrorMessage="The movie must have a title.")]
    [StringLength(50, ErrorMessage="The movie title is too long.")]
    public string Title { get; set; }
}
```

The attributes on the Title property tell the model binder that the Title is a required field, and the maximum length of the string is 50 characters. The MVC framework can automatically display the ErrorMessage text in the browser when validation fails. Additional built-in validation attributes include an attribute to check a range and an attribute to match a regular expression.

At the time of this writing, the MVC runtime uses only the validation attributes for server-side validation checks. The MVC team expects to generate client-side validation logic from the validation attributes by the time it releases MVC 2.0. Driving both the server- and client-side validation using these attributes will be a boon for the maintainability of an application.

Templated Helpers

Templated helpers in ASP.NET MVC 2.0 also consume DataAnnotation attributes. But instead of using the attributes to drive validation logic, template helpers use the attributes to drive the UI display of a model. The template helpers begin with the new HTML helper methods DisplayFor and EditorFor. These helper methods will locate the templates for a given model based on the model's type. For example, let's use the Movie class we looked at before, but with an additional property:

```
public class Movie
{
    // ...

    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
}
```

In this scenario, every movie carries its release date, but no one ever cares what time of day a movie is released. We only want to display the date information when displaying this property, and not the time information. Notice the property is decorated with a DataType attribute that advertises our intention.

To properly display the release date, we need a display template. A display template is just a partial view with an .ascx extension that lives inside a DisplayTemplates folder. The DisplayTemplates folder itself can live underneath a controller's view folder (in which case the template applies only to the views for that one controller), or in the shared views folder (in which case the template is available everywhere). In this case, the template needs the name Date.ascx and looks like the following:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>
<%= Html.Encode(String.Format("{0:d}", Model)) %>
```

In order for the MVC framework to use this template, we need to use the DisplayFor helper method when rendering the ReleaseDate property. The code shown in **Figure 3** is from another template, the Movie.ascx display template.

Figure 3 **Movie.ascx Display Template**

```
<%@ Control Language="C#"
    Inherits="System.Web.Mvc.ViewUserControl<Movie>" %>

    <fieldset>
        <legend>Fields</legend>
        <p>
            Title:
            <%= Html.LabelFor(m => m.Title) %>
            <%= Html.DisplayFor(m => m.Title) %>
        </p>
        <p>
            <%= Html.LabelFor(m => m.ReleaseDate) %>
            <%= Html.DisplayFor(m => m.ReleaseDate) %>
        </p>
    </fieldset>
```

Notice how the LabelFor and DisplayFor helper methods are strongly typed, which can help you propagate changes if a model is refactored. To use the Movie.ascx template to display a movie anywhere in an application, we just need to use the DisplayFor helper again. The following code is from a view that is strongly typed against the Movie class:

```
<asp:Content ID="detailContent"
             ContentPlaceHolderID="MainContent"
             runat="server">
        Movie:
        <%= Html.DisplayFor(m => m) %>

    </p>
</asp:Content>
```

The DisplayFor method is strongly typed to use the same model as the view page, so the m parameter in the DisplayFor lambda expression is of type Movie. DisplayFor will automatically use the Movie.ascx template when displaying the movie (which in turn uses a DisplayFor to find the Date.ascx template). If we did not use the DataType attribute on the ReleaseDate property of a movie, DisplayFor would not use the Date.ascx template and would display the date and the time portions of the ReleaseDate, but the DataType attribute helps guide the framework to the correct template. This concept of strongly typed, nested templates and data type annotations is powerful and will prove to be a productivity boost. ■

K. SCOTT ALLEN *is a member of the Pluralsight technical staff and founder of OdeToCode. You can reach Scott at scott@OdeToCode.com or read his blog at odetocode.com/blogs/scott.*

THANKS to the following technical experts for reviewing this article: *Phil Haack and Matthew Osborn*

Layered Windows with Direct2D

In my third installment on Direct2D, I'm going to show off some of its unmatched power when it comes to interoperability. Instead of exhaustively detailing all the various interoperability options that Direct2D provides, I'm going to walk you through a practical application: layered windows. Layered windows are one of those Windows features that have been around for a long time but haven't evolved much and thus require special care to use effectively with modern graphics technologies.

In this article I'm going to assume you have a basic familiarity with Direct2D programming. If not, I recommend you read my previous articles from the June (msdn.microsoft.com/magazine/dd861344) and September (msdn.microsoft.com/magazine/ee413543) issues that introduced the fundamentals of programming and drawing with Direct2D.

Originally, layered windows served a few different purposes. In particular, they could be used to easily and efficiently produce visual effects and flicker-free rendering. In the days when GDI was the predominant method for producing graphics, this was a real bonus. In today's hardware-accelerated world, however, it is no longer compelling because layered windows still belong to the world of User32/GDI and have not been updated in any significant way to support DirectX, the Microsoft platform for high-performance and high-quality graphics.

Layered windows do provide the unique ability to compose a window on the desktop using per-pixel alpha blending, which cannot be achieved in any other way with the Windows SDK.

I should mention that there are really two types of layered window. The distinction comes down to whether you need per-pixel opacity control or you just need to control the opacity of the window as a whole. This article is about the former, but if you really just need to control the opacity of a window, you can do so by simply calling the SetLayeredWindowAttributes function after creating the window to set the alpha value.

```
Verify(SetLayeredWindowAttributes(
  windowHandle,
  0, // no color key
  180, // alpha value
  LWA_ALPHA));
```

This assumes you've created the window with the WS_EX_LAYERED extended style or applied it after the fact using the SetWindowLong function. **Figure 1** provides an example of such a window. The benefit should be obvious: you don't need to change anything about the way your application paints the window as the Desktop Window Manager (DWM) will automatically blend the window appropriately. On the flip side, you need to draw absolutely everything yourself. Of course, if you're using a brand-new rendering technology such as Direct2D, that's not a problem!

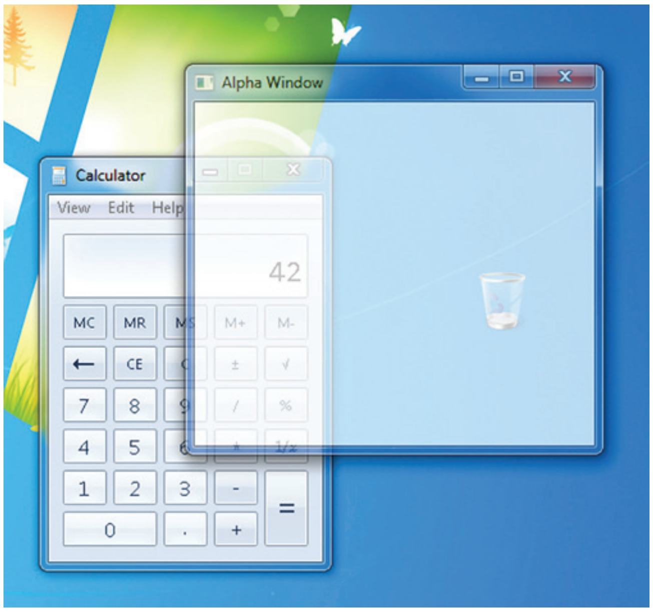


Figure 1 **Window with Alpha Value**

So what's involved? Well, at a fundamental level it is straightforward. First, you need to fill in an UPDATELAYEREDWINDOWINFO structure. This structure provides the position and size of a layered window as well as a GDI device context (DC) that defines the surface of the window—and therein lies the problem. DCs belong to the old world of GDI and are far from the world of DirectX and hardware acceleration. More on that in a moment.

Besides being full of pointers to structures that you need to allocate yourself, the UPDATELAYEREDWINDOWINFO structure isn't fully documented in the Windows SDK, making it less than obvious to use. In all, you need to allocate five structures. There's the source position identifying the location of the bitmap to copy from the DC. There's the window position identifying where the window will be positioned on the desktop once

Send your questions and comments for Kerr to mmwincpp@microsoft.com.

Figure 2 **LayeredWindowInfo Wrapper Class**

```
class LayeredWindowInfo {
  const POINT m_sourcePosition;
  POINT m_windowPosition;
  CSize m_size;
  BLENDFUNCTION m_blend;
  UPDATELAYEREDWINDOWINFO m_info;

public:

  LayeredWindowInfo(
    __in UINT width,
    __in UINT height) :
    m_sourcePosition(),
    m_windowPosition(),
    m_size(width, height),
    m_blend(),
    m_info() {

      m_info.cbSize = sizeof(UPDATELAYEREDWINDOWINFO);
      m_info.pptSrc = &m_sourcePosition;
      m_info.pptDst = &m_windowPosition;
      m_info.psize = &m_size;
      m_info.pblend = &m_blend;
      m_info.dwFlags = ULW_ALPHA;

      m_blend.SourceConstantAlpha = 255;
      m_blend.AlphaFormat = AC_SRC_ALPHA;
    }

  void Update(
    __in HWND window,
    __in HDC source) {

    m_info.hdcSrc = source;

    Verify(UpdateLayeredWindowIndirect(window, &m_info));
  }

  UINT GetWidth() const { return m_size.cx; }

  UINT GetHeight() const { return m_size.cy; }
};
```

updated. There's the size of the bitmap to copy, which also defines the size of the window:

```
POINT sourcePosition = {};
POINT windowPosition = {};
SIZE size = { 600, 400 };
```

Then there's the BLENDFUNCTION structure that defines how the layered window will be blended with the desktop. This is a surprisingly versatile structure that is often overlooked, but can be quite helpful. Normally you might populate it as follows:

```
BLENDFUNCTION blend = {};
blend.SourceConstantAlpha = 255;
blend.AlphaFormat = AC_SRC_ALPHA;
```

The AC_SRC_ALPHA constant just indicates that the source bitmap has an alpha channel, which is the most common scenario.

The SourceConstantAlpha, however, is interesting in that you can use it in much the same way you might use the SetLayeredWindowAttributes function to control the opacity of the window as a whole. When it is set to 255, the layered window will just use the per-pixel alpha values, but you can adjust it all the way to zero, or fully transparent, to produce effects such as fading the window in or out without the cost of redrawing. It should now be obvious why the BLENDFUNCTION structure is named as it is: the resulting alpha-blended window is a function of this structure's value.

Last, there's the UPDATELAYEREDWINDOWINFO structure that ties it all together:

```
UPDATELAYEREDWINDOWINFO info = {};
info.cbSize = sizeof(UPDATELAYEREDWINDOWINFO);
info.pptSrc = &sourcePosition;
info.pptDst = &windowPosition;
info.psize = &size;
info.pblend = &blend;
info.dwFlags = ULW_ALPHA;
```

This should be pretty self-explanatory at this point, with the only undocumented member being the dwFlags variable. A value of ULW_ALPHA, which should look familiar if you've used the older UpdateLayeredWindow function before, just indicates that the blend function should be used.

Finally, you need to provide the handle to the source DC and call the UpdateLayeredWindowIndirect function to update the window:

```
info.hdcSrc = sourceDC;

Verify(UpdateLayeredWindowIndirect(
  windowHandle, &info));
```

And that's it. The window won't receive any WM_PAINT messages. Any time you need to show or update the window, just call the UpdateLayeredWindowIndirect function. To keep all of this boilerplate code out of the way, I'm going to use the LayeredWindowInfo wrapper class shown in **Figure 2** in the rest of this article.

Figure 3 provides a basic skeleton for a layered window using ATL/WTL and the LayeredWindowInfo wrapper class from **Figure 2**. This first thing to notice is that there's no need to call UpdateWindow since this code doesn't use WM_PAINT. Instead it immediately calls the Render method, which in turn is required to perform some drawing and to provide a DC to LayeredWindowInfo's Update method. How that drawing occurs and where the DC comes from is where it gets interesting.

The GDI/GDI+ Way

I'll first show you how it was done in GDI/GDI+. First you need to create a pre-multiplied 32-bits-per-pixel (bpp) bitmap using a blue-green-red-alpha (BGRA) color channel byte order. Pre-multiplied just means that the color channel values have already been multiplied by the alpha value. This tends to provide better performance for alpha blending images, but it means you need to reverse the process by dividing the color values by the alpha value to get their true color values. In GDI terminology, this is called a 32-bpp device-independent bitmap (DIB) and is created by filling out a BITMAPINFO structure and passing it to the CreateDIBSection function (see **Figure 4**).

There are a lot of details here, but they aren't relevant to the discussion. This API function goes back a long way. What you should take note of is that I've specified a negative height for the bitmap. The BITMAPINFOHEADER structure defines either a bottom-up or a top-down bitmap. If the height is positive you'll end up with a bottom-up bitmap, and if it's negative you'll get a top-down bitmap. Top-down bitmaps have their origin in the upper-left corner, whereas bottom-down bitmaps have their origin in the lower-left corner.

Although not strictly necessary in this case, I tend to use top-down bitmaps as that is the format used by most of the modern imaging

Does your Team do more than just track bugs?

Free Trial and Single User FreePack™ available at www.alexcorp.com

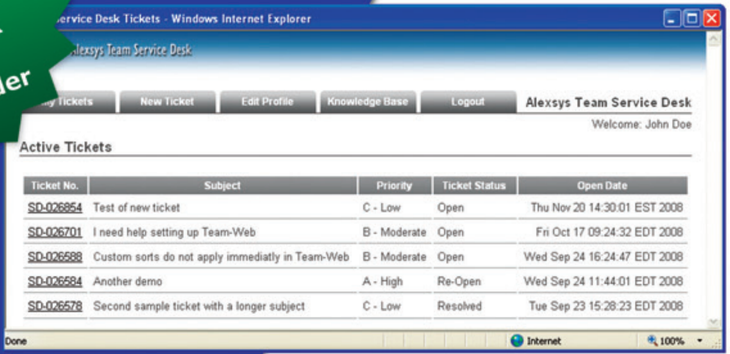
Alexsys Team® does! Alexsys Team 2 is a multi-user Team management system that provides a powerful yet easy way to manage all the members of your team and their tasks - including defect tracking. Use Team right out of the box or tailor it to your needs.

Track all your project tasks in one database so you can work together to get projects done.

- Quality Control / Compliance Tracking
- Project Management
- Bugs and Features
- Help Desk
- Sales and Marketing
- Action Items

New Alexsys Team Service Desk with Visual Builder

Alexsys Team


Service Desk Tickets - Windows Internet Explorer

Alexsys Team Service Desk

Alexsys Team Service Desk

Welcome: John Doe

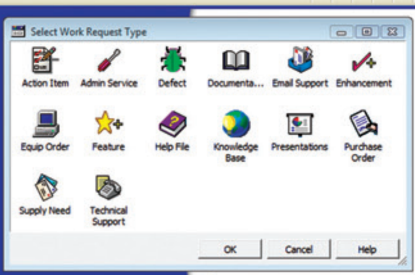
Active Tickets

Ticket No.	Subject	Priority	Ticket Status	Open Date
SD-026854	Test of new ticket	C - Low	Open	Thu Nov 20 14:30:01 EST 2008
SD-026701	I need help setting up Team-Web	B - Moderate	Open	Fri Oct 17 09:24:32 EDT 2008
SD-026588	Custom sorts do not apply immediatly in Team-Web	B - Moderate	Open	Wed Sep 24 16:24:47 EDT 2008
SD-026584	Another demo	A - High	Re-Open	Wed Sep 24 11:44:01 EDT 2008
SD-026578	Second sample ticket with a longer subject	C - Low	Resolved	Tue Sep 23 15:28:23 EDT 2008

Done Internet 100%

New in Team 2.10

- Calculated Fields, Formulas and Functions
- Field display formats
- XML and .NET support
- Database Migration tool
- Attachment and URL Links



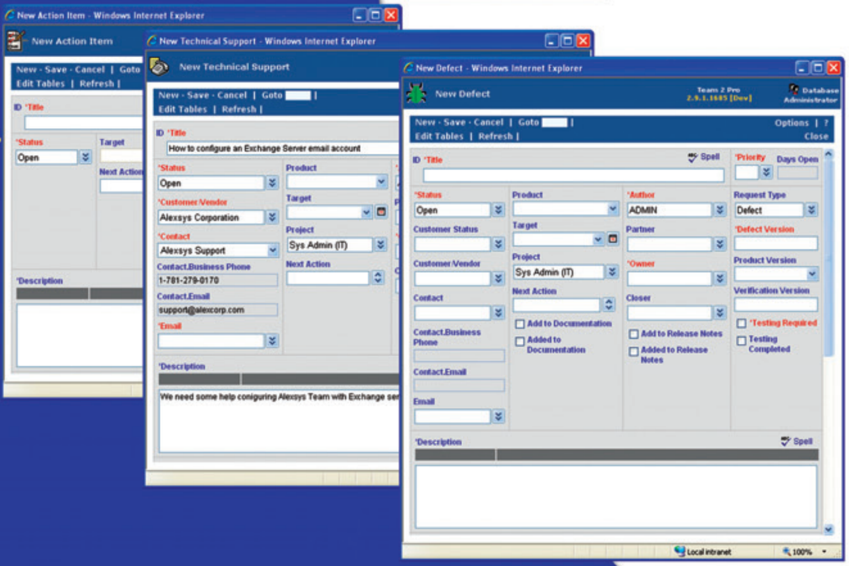
New Service Desk

- Fully Secure
- Unlimited Users Self Registered or Active Directory
- Integrated into Your Web Site
- Fast/AJAX Dynamic Content
- Unlimited Service Desks
- Visual Service Desk Builder

Team 2 Features

- Windows and Web Clients
- Multiple Work Request Forms
- Customizable Database
- Point and Click Workflows
- Role Based Security
- Clear Text Database
- Project Trees
- Time Recording
- Notifications and Escalations
- Outlook Integration

ZDNet Editors' Pick


New Action Item - Windows Internet Explorer
New Action Item
New - Save - Cancel | Goto
Edit Tables | Refresh |
New Technical Support - Windows Internet Explorer
New Technical Support
New Defect - Windows Internet Explorer
New Defect
Team 2 Pro 2.9.1.1685 [Dev]
Database Administrator
Options | ?
Close
Spell
Local intranet
100%

New Action Item: ID *Title; *Status: Open; Target; Next Action; *Description

New Technical Support: ID *Title: How to configure an Exchange Server email account; *Status: Open; *Customer/Vendor: Alexsys Corporation; *Contact: Alexsys Support; Contact.Business Phone: 1-781-279-0170; Contact.Email: support@alexcorp.com; *Email; Product; Target; Project: Sys Admin (IT); Next Action; *Description: We need some help coniguring Alexsys Team with Exchange se

New Defect: ID *Title; *Priority; Days Open; *Status: Open; Customer Status; Customer/Vendor; Contact; Contact.Business Phone; Contact.Email; Email; Product; Target; Project: Sys Admin (IT); Next Action; ☐ Add to Documentation; ☐ Added to Documentation; *Author: ADMIN; Partner; *Owner; Closer; ☐ Add to Release Notes; ☐ Added to Release Notes; Request Type: Defect; *Defect Version; Product Version; Verification Version; ☐ *Testing Required; ☐ Testing Completed; *Description

Free Trial and Single User FreePack™ available at www.alexcorp.com.
FreePack™ includes a free single user Team Pro and Team-Web license.
Need more help? Give us a call at 1-888-880-ALEX (2539).

Team 2 works with its own standard database, while Team Pro works with Microsoft SQL, MySQL, and Oracle Servers.
Team 2 works with Windows 7/2008/2003/Vista/XP.
Team-Web works with Internet Explorer, Firefox, Netscape, Safari, and Chrome.

components in Windows and thus improves interoperability. This also leads to a positive stride, which can be calculated as follows:

```
UINT stride = (width * 32 + 31) / 32 * 4;
```

At this point you have enough information to start drawing in the bitmap through the bits pointer. Of course, unless you're completely insane you'll want to use some drawing functions, but unfortunately most of those provided by GDI don't support the alpha channel. That's where GDI+ comes in.

Although you could pass the bitmap data directly to GDI+, let's instead create a DC for it since you'll need it anyway to pass to the UpdateLayeredWindowIndirect function. To create the DC, call the aptly named CreateCompatibleDC function, which creates a memory DC that is compatible with the desktop. You can then call the SelectObject function to select the bitmap into the DC. The GdiBitmap wrapper class in **Figure 5** wraps all of this up and provides some extra housekeeping.

The GDI+ Graphics class, which provides methods for drawing to some device, can be constructed with the bitmap's DC. **Figure 6** shows how the LayeredWindow class from **Figure 3** can be updated to support rendering with GDI+. Once you have all of the boilerplate GDI code out of the way, it's quite straightforward. The window's size is passed to the GdiBitmap constructor and the bitmap's DC is passed to the Graphics constructor and the Update method. Although straightforward, neither GDI nor GDI+ are hardware-accelerated (for the most part), nor do they provide particularly powerful rendering functionality.

The Architecture Problem

By contrast, this is all it takes create a layered window with Windows Presentation Foundation (WPF):

```
class LayeredWindow : Window {
  public LayeredWindow() {
    WindowStyle = WindowStyle.None;
    AllowsTransparency = true;

    // Do some drawing here
  }
}
```

Although incredibly simple, it belies the complexity involved and the architectural limitations of using layered windows. No matter how you sugarcoat it, layered windows must follow the architectural principles outlined thus far in this article. Although WPF may be able to use hardware acceleration for its rendering, the results still need to be copied to a pre-multiplied BGRA bitmap selected into a compatible DC before the display is updated via a call to the UpdateLayeredWindowIndirect function. Since WPF is not exposing anything more than a bool variable, it has to make certain choices on your behalf that you have no control over. Why does that matter? It comes down to hardware.

A graphics processing unit (GPU) prefers dedicated memory to achieve the best performance. This means that if you need to manipulate an existing bitmap, it needs to be copied from system memory (RAM) to GPU memory, which tends to be much slower than copying between two locations in system memory. The converse is also true: if you create and render a bitmap using the GPU, then decide to copy it to system memory, that's an expensive copy operation.

Figure 3 **Layered Window Skeleton**

```
class LayeredWindow :
  public CWindowImpl<LayeredWindow,
  CWindow, CWinTraits<WS_POPUP, WS_EX_LAYERED>> {

  LayeredWindowInfo m_info;

public:

  BEGIN_MSG_MAP(LayeredWindow)
    MSG_WM_DESTROY(OnDestroy)
  END_MSG_MAP()

  LayeredWindow() :
    m_info(600, 400) {

    Verify(0 != __super::Create(0)); // parent
    ShowWindow(SW_SHOW);
    Render();
  }

  void Render() {
    // Do some drawing here

    m_info.Update(m_hWnd,
      /* source DC goes here */);
  }

  void OnDestroy() {
    PostQuitMessage(1);
  }
};
```

Figure 4 **Creating a DIB**

```
BITMAPINFO bitmapInfo = {};
bitmapInfo.bmiHeader.biSize =
  sizeof(bitmapInfo.bmiHeader);
bitmapInfo.bmiHeader.biWidth =
  m_info.GetWidth();
bitmapInfo.bmiHeader.biHeight =
  0 - m_info.GetHeight();
bitmapInfo.bmiHeader.biPlanes = 1;
bitmapInfo.bmiHeader.biBitCount = 32;
bitmapInfo.bmiHeader.biCompression =
  BI_RGB;

void* bits = 0;

CBitmap bitmap(CreateDIBSection(
  0, // no DC palette
  &bitmapInfo,
  DIB_RGB_COLORS,
  &bits,
  0, // no file mapping object
  0)); // no file offset
```

Normally this should not occur as bitmaps rendered by the GPU are typically sent directly to the display device. In the case of layered windows, the bitmap must travel back to system memory since User32/GDI resources involve both kernel-mode and user-mode resources that require access to the bitmap. Consider, for example, the fact that User32 needs to hit test layered windows. Hit testing of a layered window is based on the alpha values of the bitmap, allowing mouse messages through if the pixel at a particular point is transparent. As a result, a copy of the bitmap is required in system memory to allow this to happen. Once the bitmap has been copied by UpdateLayeredWindowIndirect, it is sent straight back to the GPU so the DWM can compose the desktop.

Figure 5 **DIB Wrapper Class**

```
class GdiBitmap {
  const UINT m_width;
  const UINT m_height;
  const UINT m_stride;
  void* m_bits;
  HBITMAP m_oldBitmap;

  CDC m_dc;
  CBitmap m_bitmap;

public:

  GdiBitmap(__in UINT width,
            __in UINT height) :
    m_width(width),
    m_height(height),
    m_stride((width * 32 + 31) / 32 * 4),
    m_bits(0),
    m_oldBitmap(0) {

    BITMAPINFO bitmapInfo = { };
    bitmapInfo.bmiHeader.biSize =
      sizeof(bitmapInfo.bmiHeader);
    bitmapInfo.bmiHeader.biWidth =
      width;
    bitmapInfo.bmiHeader.biHeight =
      0 - height;
    bitmapInfo.bmiHeader.biPlanes = 1;
    bitmapInfo.bmiHeader.biBitCount = 32;
    bitmapInfo.bmiHeader.biCompression =
      BI_RGB;

    m_bitmap.Attach(CreateDIBSection(
      0, // device context
      &bitmapInfo,
      DIB_RGB_COLORS,
      &m_bits,
      0, // file mapping object
      0)); // file offset
    if (0 == m_bits) {
      throw bad_alloc();
    }

    if (0 == m_dc.CreateCompatibleDC()) {
      throw bad_alloc();
    }

    m_oldBitmap = m_dc.SelectBitmap(m_bitmap);
  }

  ~GdiBitmap() {
    m_dc.SelectBitmap(m_oldBitmap);
  }

  UINT GetWidth() const {
    return m_width;
  }

  UINT GetHeight() const {
    return m_height;
  }

  UINT GetStride() const {
    return m_stride;
  }

  void* GetBits() const {
    return m_bits;
  }

  HDC GetDC() const {
    return m_dc;
  }
};
```

Besides the expense of copying memory back and forth, forcing the GPU to synchronize with the CPU is costly as well. Unlike typical CPU-bound operations, GPU operations tend to all be performed asynchronously, which provides great performance when batching a stream of rendering commands. Every time we need to cross paths with the CPU, it forces batched commands to be flushed and the CPU to wait until the GPU has completed, leading to less than optimal performance.

This all means that you need to be careful about these roundtrips and the frequency and costs involved. If the scenes being rendered are sufficiently complex, then the performance of hardware acceleration can easily outweigh the cost of copying the bitmaps. On the other hand, if the rendering is not very costly and can be performed by the CPU, you might find that opting for no hardware acceleration will ultimately provide better performance. These choices aren't easy to make. Some GPUs don't even have dedicated memory and instead use a portion of system memory, which reduces the cost of the copy.

The catch is that neither GDI nor WPF give you a choice. In the case of GDI, you're stuck with the CPU. In the case of WPF, you're forced into using whatever rendering approach WPF uses, which is typically hardware acceleration via Direct3D.

Then Direct2D came along.

Direct2D to GDI/DC

Direct2D was designed to render to whatever target you choose. If it's a window or Direct3D texture, Direct2D does this directly on the GPU without involving any copying. If it's a Windows Imaging Component (WIC) bitmap, Direct2D similarly renders directly using the CPU instead. Whereas WPF strives to put much of its rendering on the GPU and uses a software rasterizer as a fallback, Direct2D provides the best of both worlds with unparalleled immediate mode rendering on the GPU for hardware acceleration, and highly optimized rendering on the CPU when a GPU is either not available or not desired.

As you can imagine, there are quite a few ways to render a layered window with Direct2D. Let's take a look at a few and I'll point out

Figure 6 **GDI Layered Window**

```
class LayeredWindow :
  public CWindowImpl< ... {

  LayeredWindowInfo m_info;
  GdiBitmap m_bitmap;
  Graphics m_graphics;

public:
  LayeredWindow() :
    m_info(600, 400),
    m_bitmap(m_info.GetWidth(), m_info.GetHeight()),
    m_graphics(m_bitmap.GetDC()) {
    ...
  }

  void Render() {
    // Do some drawing with m_graphics object

    m_info.Update(m_hWnd,
      m_bitmap.GetDC());
  }
...
```

Figure 7 **Render Target DC Wrapper Class**

```
class RenderTargetDC {
  ID2D1GdiInteropRenderTarget* m_renderTarget;
  HDC m_dc;

public:
  RenderTargetDC(ID2D1GdiInteropRenderTarget* renderTarget) :
    m_renderTarget(renderTarget),
    m_dc(0) {

    Verify(m_renderTarget->GetDC(
      D2D1_DC_INITIALIZE_MODE_COPY,
      &m_dc));

  }

  ~RenderTargetDC() {
    RECT rect = {};
    m_renderTarget->ReleaseDC(&rect);
  }

  operator HDC() const {
    return m_dc;
  }
};
```

the recommended approaches depending on whether you want to use hardware acceleration.

First, you could just rip out the GDI+ Graphics class from **Figure 3** and replace it with a Direct2D DC render target. This might make sense if you have a legacy application with a lot invested in GDI, but it's definitely not the most efficient solution. Instead of rendering directly to the DC, Direct2D renders first to an internal WIC bitmap, then copies the result to the DC. Although faster than GDI+, this nevertheless involves extra copying that could be avoided if you didn't need to use a DC for rendering.

To use this approach, start by initializing a D2D1_RENDER_TARGET_PROPERTIES structure. This tells Direct2D the format of the bitmap to use for its render target. Recall that it needs to be a pre-multiplied BGRA pixel format. This is expressed with a D2D1_PIXEL_FORMAT structure and can be defined as follows:

```
const D2D1_PIXEL_FORMAT format =
  D2D1::PixelFormat(DXGI_FORMAT_B8G8R8A8_UNORM,
  D2D1_ALPHA_MODE_PREMULTIPLIED);

const D2D1_RENDER_TARGET_PROPERTIES properties =
  D2D1::RenderTargetProperties(
  D2D1_RENDER_TARGET_TYPE_DEFAULT,
  format);
```

You can now create the DC render target using the Direct2D factory object:

```
CComPtr<ID2D1DCRenderTarget> target;

Verify(factory->CreateDCRenderTarget(
  &properties,
  &target));
```

Finally, you need to tell the render target to which DC to send its drawing commands:

```
const RECT rect = {0, 0, bitmap.GetWidth(), bitmap.GetHeight()};

Verify(target->BindDC(bitmap.GetDC(), &rect));
```

At this point you can draw with Direct2D as usual between BeginDraw and EndDraw method calls, and then call the Update method as before with the bitmap's DC. The EndDraw method ensures that all drawing has been flushed to the bound DC.

Figure 8 **GDI-Compatible Render Method**

```
void Render() {
  CreateDeviceResources();
  m_target->BeginDraw();
  // Do some drawing here
  {
    RenderTargetDC dc(m_interopTarget);
    m_info.Update(m_hWnd, dc);
  }

  const HRESULT hr = m_target->EndDraw();

  if (D2DERR_RECREATE_TARGET == hr) {
    DiscardDeviceResources();
  }
  else {
    Verify(hr);
  }
}
```

Direct2D to WIC

Now if you can avoid the GDI DC entirely and just use a WIC bitmap directly, you can achieve the best possible performance without hardware acceleration. To use this approach start by creating a pre-multiplied BGRA bitmap directly with WIC:

```
CComPtr<IWICImagingFactory> factory;
Verify(factory.CoCreateInstance(
  CLSID_WICImagingFactory));

CComPtr<IWICBitmap> bitmap;

Verify(factory->CreateBitmap(
  m_info.GetWidth(),
  m_info.GetHeight(),
  GUID_WICPixelFormat32bppPBGRA,
  WICBitmapCacheOnLoad,
  &bitmap));
```

Next you need to once again initialize a D2D1_RENDER_TARGET_PROPERTIES structure in much the same way as before, except that you must also tell Direct2D that the render target needs to be GDI-compatible:

```
const D2D1_PIXEL_FORMAT format =
  D2D1::PixelFormat(
  DXGI_FORMAT_B8G8R8A8_UNORM,
  D2D1_ALPHA_MODE_PREMULTIPLIED);

const D2D1_RENDER_TARGET_PROPERTIES properties =
  D2D1::RenderTargetProperties(
  D2D1_RENDER_TARGET_TYPE_DEFAULT,
  format,
  0.0f, // default dpi
  0.0f, // default dpi
  D2D1_RENDER_TARGET_USAGE_GDI_COMPATIBLE);
```

You can now create the WIC render target using the Direct2D factory object:

```
CComPtr<ID2D1RenderTarget> target;

Verify(factory->CreateWicBitmapRenderTarget(
  bitmap,
  properties,
  &target));
```

But what exactly does D2D1_RENDER_TARGET_USAGE_GDI_COMPATIBLE do? It's a hint to Direct2D that you will query the render target for the ID2D1GdiInteropRenderTarget interface:

```
CComPtr<ID2D1GdiInteropRenderTarget> interopTarget;
Verify(target.QueryInterface(&interopTarget));
```

For simplicity and efficiency of implementation, querying for this interface will always succeed. It is only when you try to use it, however, that it will fail if you didn't specify your desires up front.

The ID2D1GdiInteropRenderTarget interface has just two methods: GetDC and ReleaseDC. To optimize cases where hardware acceleration is used, these methods are restricted to being used between calls to the render target's BeginDraw and EndDraw methods. GetDC will flush the render target before returning the DC. Since the interop interface's methods need to be paired, it makes sense to wrap them in a C++ class as shown in **Figure 7**.

The window's Render method can now be updated to use the RenderTargetDC, as shown in **Figure 8**. The nice thing about this approach is that all of the code that is specific to creating a WIC render target is tucked away in the CreateDeviceResources method. Next I'll show you how to create a Direct3D render target to gain hardware acceleration, but in either case, the Render method shown in **Figure 8** stays the same. This makes it possible for your application to fairly easily switch render target implementations without changing all your drawing code.

Direct2D to Direct3D/DXGI

To obtain hardware-accelerated rendering, you need to use Direct3D. Because you're not rendering directly to an HWND via ID2D1HwndRenderTarget, which would gain hardware acceleration automatically, you need to create the Direct3D device yourself and connect the dots in the underlying DirectX Graphics Infrastructure (DXGI) so that you can get GDI-compatible results.

DXGI is a relatively new subsystem that lives on a layer below Direct3D to abstract Direct3D from the underlying hardware and provide a high-performance gateway for interop scenarios. Direct2D also takes advantage of this new API to simplify the move to future versions of Direct3D. To use this approach, start by creating a Direct3D hardware device. This is the device that represents the GPU that will perform the rendering. Here I'm using the Direct3D 10.1 API as this is required by Direct2D at present:

```
CComPtr<ID3D10Device1> device;

Verify(D3D10CreateDevice1(
  0, // adapter
  D3D10_DRIVER_TYPE_HARDWARE,
  0, // reserved
  D3D10_CREATE_DEVICE_BGRA_SUPPORT,
  D3D10_FEATURE_LEVEL_10_0,
  D3D10_1_SDK_VERSION,
  &device));
```

The D3D10_CREATE_DEVICE_BGRA_SUPPORT flag is crucial for Direct2D interoperability, and the BGRA pixel format should by now look familiar. In a traditional Direct3D application, you might create a swap chain and retrieve its back buffer as a texture to render into before presenting the rendered window. Since you're using Direct3D for rendering only and not for presentation, you can simply create a texture resource directly. A texture is a Direct3D resource for storing texels, which are the Direct3D equivalent of pixels. Although Direct3D provides 1-, 2- and 3-dimensional textures, all you need is a 2D texture, which most closely maps to a 2D surface (see **Figure 9**).

The D3D10_TEXTURE2D_DESC structure describes the texture to create. The D3D10_BIND_RENDER_TARGET constant indicates that the texture is bound as the output buffer, or render target, of the Direct3D pipeline. The DXGI_FORMAT_B8G8R8A8_UNORM constant ensures that Direct3D will produce the correct pixel format for GDI. Finally, the D3D10_RESOURCE_MISC_GDI_COMPATIBLE constant instructs the underlying DXGI surface to offer a GDI DC through which the results of rendering can be obtained. This Direct2D exposes through the ID2D1GdiInteropRenderTarget interface I discussed in the previous section.

Figure 9 A 2D Texture

```
D3D10_TEXTURE2D_DESC description = {};
description.ArraySize = 1;
description.BindFlags =
  D3D10_BIND_RENDER_TARGET;
description.Format =
  DXGI_FORMAT_B8G8R8A8_UNORM;
description.Width = GetWidth();
description.Height = GetHeight();
description.MipLevels = 1;
description.SampleDesc.Count = 1;
description.MiscFlags =
  D3D10_RESOURCE_MISC_GDI_COMPATIBLE;

CComPtr<ID3D10Texture2D> texture;

Verify(device->CreateTexture2D(
  &description,
  0, // no initial data
  &texture));
```

As I mentioned, Direct2D is capable of rendering to a Direct3D surface via the DXGI API to avoid tying the API to any particular version of Direct3D. This means you need to get the Direct3D texture's underlying DXGI surface interface to pass to Direct2D:

```
CComPtr<IDXGISurface> surface;
Verify(texture.QueryInterface(&surface));
```

At this point you can use the Direct2D factory object to create a DXGI surface render target:

```
CComPtr<ID2D1RenderTarget> target;

Verify(factory->CreateDxgiSurfaceRenderTarget(
  surface,
  &properties,
  &target));
```

The render target properties are the same as those I described in the previous section. Just remember to use the correct pixel format and request GDI compatibility. You can then query for the ID2D1GdiInteropRenderTarget interface and use the same Render method from **Figure 8**.

And that's all there is to it. If you want to render your layered window with hardware acceleration, use a Direct3D texture. Otherwise use a WIC bitmap. These two approaches will provide the best possible performance with the least amount of copying.

Be sure to check out the DirectX blog and, in particular, Ben Constable's August 2009 article on componentization and interoperability at blogs.msdn.com/directx. ■

KENNY KERR *is a software craftsman passionate about Windows. He is also the creator of Window Clippings (windowclippings.com). Reach him at weblogs.asp.net/kennykerr.*

THANKS to the following technical experts for reviewing this article: *Ben Constable and Mark Lawrence*

Pure Visual Studio and .NET

GET TIPS

GET CODE

GET THE BEST
HOW-TO ARTICLES
ON THE NET

Visit VisualStudioMagazine.com
and RedDevNews.com

Visual Studio MAGAZINE RedmondDeveloper NEWS

Enhancing Windows Touch Applications for Mobile Users

Windows 7 introduces Windows Touch, which enhances touch input on capable hardware and provides a solid platform for building touch applications. Potentially, this means that you can develop remarkably intuitive interfaces that users of all ages and computing abilities can understand with a minimal amount of training or instruction.

The magic behind this functionality is the Windows Touch API. Using this API, you can retrieve information about where a user is touching the screen and about a user's on-screen gestures. You also have access to real-world physics for user interface elements. Moving an on-screen object becomes as easy as moving an object in the real world. Stretching an object is like stretching a piece of elastic. When users interact with a well-implemented touch application, they feel as though they're interacting with the technology of the future, or even better, they don't notice that they're using an application at all. They don't have to use a mouse, a stylus or shortcut keys or precisely select menu items to get at the application's core functionality.

Applications tailored to mobile use should incorporate specific requirements to ensure that the experience is well suited to the user's environment. A poorly implemented touch application can completely defeat the purpose of using Windows Touch. The Windows Touch User Experience guidelines (go.microsoft.com/fwlink/?LinkId=156610) highlight ways that developers can improve the experience for users on the go. These guidelines cover various scenarios relevant to mobile application developers and make it easier to avoid potential pitfalls of Windows Touch development.

If you take away only one thing from this article, remember that when creating an application that targets mobile users, you need to consider aspects that are specific to your type of application. For instance, if your application uses Windows controls, be sure they are of adequate size and have sufficient spacing so that users can touch them easily. If you are creating an application that can take advantage of flicks, be sure that the flick actions are properly handled.

First Things First

In this article, I'll take a sample touch application and enhance it for mobile applications. I assume that you have some knowledge of COM and Windows Touch, and have Windows Touch-capable hardware. For a primer on Windows Touch, go to go.microsoft.com/fwlink/?LinkId=156612 or read Yochay Kiriaty's article at msdn.microsoft.com/magazine/ee336016.aspx.

The example is on the MSDN Code Gallery at code.msdn.microsoft.com/windowstouchmanip. The Downloads tab contains two .zip files, the first without mobile enhancements and the second with them. Download the file named Multiple Manipulators.zip, expand it and compile the project.

Send your questions and comments for Gus to goplaces@microsoft.com.

Code download available at code.msdn.microsoft.com/windowstouchmanip.

Figure 1 **Utility Functions for Simulating Touch Input with the Mouse**

```
VOID Drawable::FillInputData(TOUCHINPUT* inData, DWORD cursor, DWORD
eType, DWORD time, int x, int y)
{
    inData->dwID = cursor;
    inData->dwFlags = eType;
    inData->dwTime = time;
    inData->x = x;
    inData->y = y;
}

void Drawable::ProcessMouseData(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM
    lParam){
    TOUCHINPUT tInput;
    if (this->getCursorID() == MOUSE_CURSOR_ID){
        switch (msg){
            case WM_LBUTTONDOWN:
                FillInputData(&tInput, MOUSE_CURSOR_ID, TOUCHEVENTF_DOWN,
(DWORD)GetMessageTime(),LOWORD(lParam) * 100,HIWORD(lParam) * 100);
                ProcessInputs(hWnd, 1, &tInput, 0);
                break;

            case WM_MOUSEMOVE:
                if(LOWORD(wParam) == MK_LBUTTON)
                {
                    FillInputData(&tInput, MOUSE_CURSOR_ID, TOUCHEVENTF_
MOVE, (DWORD)GetMessageTime(),LOWORD(lParam) * 100, HIWORD(lParam) *
100);
                    ProcessInputs(hWnd, 1, &tInput, 0);
                }
                break;

            case WM_LBUTTONUP:
                FillInputData(&tInput, MOUSE_CURSOR_ID, TOUCHEVENTF_UP,
(DWORD)GetMessageTime(),LOWORD(lParam) * 100, HIWORD(lParam) * 100);
                ProcessInputs(hWnd, 1, &tInput, 0);
                setCursorID(-1);
                break;
            default:
                break;
        }
    }
}
```

Figure 2 **Changes from WndProc**

```
    case WM_LBUTTONDOWN:
      case WM_MOUSEMOVE:
      case WM_LBUTTONUP:
          for (i=0; i<drawables; i++){
            // contact start
            if (message == WM_LBUTTONDOWN && draw[i]-
>IsContacted(LOWORD(lParam), HIWORD(lParam), MOUSE_CURSOR_ID)){
                draw[i]->setCursorID(MOUSE_CURSOR_ID);
            }
            // contact end
            if (message == WM_LBUTTONUP && draw[i]->getCursorID() == MOUSE_
CURSOR_ID){
              draw[i]->setCursorID(-1);
            }
            draw[i]->ProcessMouseData(hWnd, message, wParam, lParam);
          }
          InvalidateRect(hWnd, NULL, false);
          break;
```

Figure 3 **Converting Screen Points to Client Points**

```
POINT ptInput;
void Drawable::ProcessInputs(HWND hWnd, UINT cInputs,
     PTOUCHINPUT pInputs, LPARAM lParam){
  for (int i=0; i < static_cast<INT>(cInputs); i++){
...
      ScreenToClient(hWnd, &ptInput);

      if (ti.dwFlags & TOUCHEVENTF_DOWN){
        if (IsContacted( ptInput.x, ptInput.y, ti.dwID) ){
          pManip->ProcessDownWithTime(ti.dwID, static_cast<FLOAT>
(ptInput.x), static_cast<FLOAT>( ptInput.y), ti.dwTime);
          setCursorID(ti.dwID);

          if (!CloseTouchInputHandle((HTOUCHINPUT)lParam)) {
            // Error handling
          }
        }
      }
      if (pInputs[i].dwFlags & TOUCHEVENTF_MOVE){
        pManip->ProcessMoveWithTime(ti.dwID, static_cast<FLOAT>
(ptInput.x), static_cast<FLOAT>( ptInput.y), ti.dwTime);
      }
      if (pInputs[i].dwFlags & TOUCHEVENTF_UP){
        pManip->ProcessUpWithTime(ti.dwID, static_cast<FLOAT>
(ptInput.x), static_cast<FLOAT>( ptInput.y), ti.dwTime);
        setCursorID(-1);
      }
      // If you handled the message and don't want anything else done
      // with it, you can close it

  }
}
```

Figure 4 **Updating the Touch Input Handler**

```
POINT ptInput;
void Drawable::ProcessInputs(HWND hWnd, UINT cInputs,
     PTOUCHINPUT pInputs, LPARAM lParam){
  BOOL fContinue = TRUE;
  for (int i=0; i < static_cast<INT>(cInputs) && fContinue; i++){
...
      if (ti.dwFlags & TOUCHEVENTF_DOWN){
        if (IsContacted( ptInput.x, ptInput.y, ti.dwID) ){
          pManip->ProcessDownWithTime(ti.dwID, static_cast<FLOAT>
(ptInput.x), static_cast<FLOAT>(ptInput.y), ti.dwTime);
          setCursorID(ti.dwID);

          fContinue = FALSE;
        }
      }
...
  }
  CloseTouchInputHandle((HTOUCHINPUT)lParam);

}
```

To be honest, using the example is at times like trying to thread a needle while wearing mittens: the functionality is diminished to a point that frustrates users. For example, if you try to select overlapping objects in an overlapping region, you will select and move both objects. You also can resize an object so that it's so small that you can't resize it again. I'll show you how to fix these problems and make other changes that improve the user experience in the areas of general usability, object selection and the use of a natural user interface. Remember that considerations you make for each mobile application depend on how users will interact with it. The issues I cover here should be used as guidelines only for this specific application.

General Usability

When a user is manipulating graphical objects in a mobile application, he must be able to perform tasks without the use of a keyboard and mouse. Also, when a mobile user is using high DPI settings or is connected to multiple screens, the application must behave consistently. (High DPI requirements are discussed in detail at go.microsoft.com/fwlink/?LinkId=153387.)

For the sample application, Windows Touch implicitly addresses the issue of obtaining input from the user without a mouse and keyboard. Users can use touch input to perform actions such as object translation, scaling and so on. A related consideration is supporting mouse and keyboard input in an application designed for touch input so that a user can drive the manipulation processor using any input, including mouse input. **Figure 1** shows how you could let a user simulate touch input through mouse input by adding some utility functions to the sample application's Drawable class. You also have to add handlers to WndProc to hook mouse input to the input processor (see **Figure 2**).

To address high DPI requirements, you can add a project manifest to the build settings to make the application aware of the DPI settings. You do this so that the coordinate space is correct when you are working at various DPI levels. (If you are interested in seeing how the application behaves after you have changed the DPI level, right-click your desktop, click Personalize and then change your DPI level in the Display control panel.)

The following XML shows how this manifest could be defined to make your application compatible with high DPI settings:

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0"
   xmlns:asmv3="urn:schemas-microsoft-com:asm.v3" >
  <asmv3:application>
    <asmv3:windowsSettings xmlns=
"http://schemas.microsoft.com/SMI/2005/WindowsSettings">
      <dpiAware>true</dpiAware>
    </asmv3:windowsSettings>
  </asmv3:application>
</assembly>
```

Once the project manifest is added to the project's properties, the application correctly sends touch input information to the manipulation processor regardless of the user's DPI settings. You can also use the ScreenToClient method (see go.microsoft.com/fwlink/?LinkID=153391 for more information) to ensure that the coordinate space is set to the application coordinates rather than to the screen coordinates. **Figure 3** shows the changes to the ProcessInputs member function of the Drawable class that convert the screen points to client points. Now when the user connects an external monitor to a Windows Touch–enabled PC, the coordinate space of your application will remain consistent and DPI aware.

Object Selection

To ensure that object selection functions as the user expects, the user must be able to select overlapping objects in a natural and intuitive manner, and the user must be able to select and easily transform objects on screens on smaller form factors or screens with limited touch input resolution.

As the application currently operates, when a user selects an overlapping object, the application sends touch data to all the objects that are under the point where the user touches the window. To modify the application to stop handling touch input after the first touched object is encountered, you need to close the touch input handle when an object is selected. **Figure 4** shows how you can update the touch input handler to stop handling the touch message after the first object is contacted.

After you implement this change, when a touched object is contacted, touch data stops getting sent to other objects in the array. To change the application so that only the first object under mouse input receives touch input, you can break out of the switch in the input processing statement for mouse down input, which short-circuits the logic for mouse input. **Figure 5** demonstrates the changes to the switch statement in the mouse input handler.

Figure 5 **Changing the Switch Statement in the Mouse Input Handler**

```
    case WM_LBUTTONDOWN:
        for (i=0; i<drawables; i++){
          if (draw[i]->IsContacted(LOWORD(lParam), HIWORD(lParam), MOUSE_
CURSOR_ID)){
              draw[i]->setCursorID(MOUSE_CURSOR_ID);
              draw[i]->ProcessMouseData(hWnd, message, wParam, lParam);
              break;
          }
        }
...
```

Next, you should change your application to ensure that when a user resizes objects, the objects will not become so small that the user cannot select or resize them again. To address this, you can use settings in the Manipulations API to restrict how small an object can be sized. The following changes are made to the manipulation processor utility of the Drawable object:

```
void Drawable::SetUpManipulator(void){
  pManip->put_MinimumScaleRotateRadius(4000.0f);
}
```

Figure 6 **Implementations of IManipulationProcessor and IInertiaProcesor Constructors**

```
CManipulationEventSink::CManipulationEventSink(IManipulationProcessor
*manip, IInertiaProcessor *inert, Drawable* d){
    drawable = d;
    // Yes, we are extrapolating inertia in this case
    fExtrapolating = false;

    //Set initial ref count to 1
    m_cRefCount = 1;

    m_pManip = NULL;
    m_pInert = inert;

    m_cStartedEventCount = 0;
    m_cDeltaEventCount = 0;
    m_cCompletedEventCount = 0;

    HRESULT hr = S_OK;

    //Get the container with the connection points
    IConnectionPointContainer* spConnectionContainer;

    hr = manip->QueryInterface(
      IID_IConnectionPointContainer,
      (LPVOID*) &spConnectionContainer
      );

    if (spConnectionContainer == NULL){
        // Something went wrong, try to gracefully quit
    }

    //Get a connection point
    hr = spConnectionContainer->FindConnectionPoint
(__uuidof(_IManipulationEvents), &m_pConnPoint);

    if (m_pConnPoint == NULL){
        // Something went wrong, try to gracefully quit
    }

    DWORD dwCookie;

    //Advise
    hr = m_pConnPoint->Advise(this, &dwCookie);
}
```

```
CManipulationEventSink::CManipulationEventSink(IInertiaProcessor *inert,
Drawable* d)
{
    drawable = d;
    // Yes, we are extrapolating inertia in this case
    fExtrapolating = true;

    //Set initial ref count to 1
    m_cRefCount = 1;

    m_pManip = NULL;
    m_pInert = inert;

    m_cStartedEventCount = 0;
    m_cDeltaEventCount = 0;
    m_cCompletedEventCount = 0;

    HRESULT hr = S_OK;

    //Get the container with the connection points
    IConnectionPointContainer* spConnectionContainer;

    hr = inert->QueryInterface(
      IID_IConnectionPointContainer,
      (LPVOID*) &spConnectionContainer
      );

    if (spConnectionContainer == NULL){
        // Something went wrong, try to gracefully quit
    }

    //Get a connection point
    hr = spConnectionContainer->FindConnectionPoint
(__uuidof(_IManipulationEvents), &m_pConnPoint);
    if (m_pConnPoint == NULL){
        // Something went wrong, try to gracefully quit
    }

    DWORD dwCookie;

    //Advise
    hr = m_pConnPoint->Advise(this, &dwCookie);
}
```

Figure 7 **Updating the Drawable Class**

```
interface IInertiaProcessor;
public:
...
    // Inertia Processor Initiation
    virtual void SetUpInertia(void);

...
protected:

    HWND m_hWnd;

    IManipulationProcessor* pManip;
    IInertiaProcessor*      pInert;
    CManipulationEventSink* pEventSink;
```

Figure 8 **Incorporating the New Event Sink Constructors**

```
Drawable::Drawable(HWND hWnd){
. .

    // Initialize manipulators
    HRESULT hr = CoCreateInstance(CLSID_ManipulationProcessor,
          NULL,
          CLSCTX_INPROC_SERVER,
          IID_IUnknown,
          (VOID**)(&pManip)
    );

    // Initialize inertia processor
    hr = CoCreateInstance(CLSID_InertiaProcessor,
          NULL,
          CLSCTX_INPROC_SERVER,
          IID_IUnknown,
          (VOID**)(&pInert)
    );

    //TODO: test HR
    pEventSink = new CManipulationEventSink(pManip,pInert, this);
    pInertSink = new CManipulationEventSink(pInert, this);
    pEventSink->SetWindow(hWnd);
    pInertSink->SetWindow(hWnd);

    SetUpManipulator();
    SetUpInertia();
    m_hWnd = hWnd;
}
```

Now when you scale an object, scale values less than 4,000 centipixels are ignored by the application. Each Drawable object can have unique constraints set in the SetUpManipulator method to ensure that the object can be manipulated in an appropriate manner.

Natural User Interface

In an application designed to have a natural look and feel, a user should be able to perform simultaneous manipulations on multiple objects. Objects should have simple physics when they're moved across the screen, similar to how they behave in the real world, and the user should not be able to manipulate objects off screen.

By design, applications that use the Manipulations API should support simultaneous manipulation of objects. Because this example uses the Manipulations API, simultaneous manipulations are enabled automatically. When you use the Gestures API for Windows Touch support, simultaneous manipulation of objects is not possible and compound gestures such as pan+zoom and zoom+rotate aren't either. For this reason, you should use the Manipulations API when you are designing a Windows Touch application that targets mobile PCs.

The Windows Touch API includes the IInertiaProcessor interface to enable support for simple physics (inertia). IInertiaProcessor uses some of the same methods as the IManipulationProcessor interface to simplify adding support for inertia to applications that are already using manipulations. To enable support for inertia, you need to extend the existing event sink for the manipulation processor, add a reference to an IInertiaProcessor interface instance on the Drawable object, connect event data from the event sink to the IInertiaProcessor object and use a timer to trigger the IInertia-Processor interface to trigger manipulation events for inertia. Let's look at each operation in more detail.

First you need to update the event sink to enable support for sending data to an IInertiaProcessor interface. The following members and constructor definitions are added to the event sink implementation header:

```
class CManipulationEventSink : _IManipulationEvents
{
public:
    CManipulationEventSink(IInertiaProcessor *inert, Drawable* d);
    CManipulationEventSink(IManipulationProcessor *manip,
IInertiaProcessor *inert, Drawable* d);

...
protected:
    IInertiaProcessor*      m_pInert;
    BOOL fExtrapolating;
```

You also add a member and an access method to the event sink for setting a HWND that is used for timers, as shown here:

```
public:
    void SetWindow(HWND hWnd) {m_hWnd = hWnd;}
...

private:
...
HWND m_hWnd;
```

Next, change the constructor that takes an IManipulation-Processor interface to accept an IInertiaProcessor interface, and add a constructor that accepts only an IInertiaProcessor interface. The constructor that takes an IManipulationProcessor interface uses the reference to the IInertiaProcessor interface to trigger inertia from the ManipulationCompleted event. The constructor that takes only an IInertiaProcessor interface handles events that are for inertia. **Figure 6** shows the implementations of these constructors.

Next, you update the Drawable class to enable support for inertia. The forward definition shown in **Figure 7** should be added as well as a member variable, pInert.

The following code shows the simplest implementation for the SetUpInertia method. This method finishes any processing, resets the inertia processor and then sets any configuration settings:

```
void Drawable::SetUpInertia(void){
    // Complete any previous processing
    pInert->Complete();

    pInert->put_InitialOriginX(originX*100);
    pInert->put_InitialOriginY(originY*100);

    // Configure the inertia processor
    pInert->put_DesiredDeceleration(.1f);
}
```

After you update the Drawable class, change the Drawable constructor to incorporate the new event sink constructors, as shown in **Figure 8**.

And now add the following timer message handler to the main program:

```
case WM_TIMER:
    // wParam indicates the timer ID
    for (int i=0; i<drawables; i++){
        if (wParam == draw[i]->GetIndex() ){
            BOOL b;
            draw[i]->ProcessInertia(&b);
        }
    }
break;
```

Once you have your timer handler and your timer is set up, you need to trigger it from the completed message in the event where there is no inertia. **Figure 9** shows changes to the completed event that start the timer when the user is finished manipulating an object and stop the timer once inertia is complete.

Notice that reducing the timer interval, the third parameter for SetTimer, results in smoother animation but triggers more update events, potentially causing performance degradation depending on what operations the event handlers perform. For example, changing this value to 5 results in very smooth animation, but the window is updated more frequently because of additional calls to CManipulationEventSink::ManipulationDelta.

Now you can build and run your application, but without additional changes, manipulated objects will drift off screen. To prevent objects from being manipulated off screen, configure the IInertiaProcessor interface to use elastic bounds. **Figure 10** shows the changes that should be made to the SetUpInertia method for the Drawable object to initialize the screen boundaries.

Figure 9 **Changes to the Completed Event**

```
HRESULT STDMETHODCALLTYPE CManipulationEventSink::ManipulationCompleted(
    /* [in] */ FLOAT x,
    /* [in] */ FLOAT y,
    /* [in] */ FLOAT cumulativeTranslationX,
    /* [in] */ FLOAT cumulativeTranslationY,
    /* [in] */ FLOAT cumulativeScale,
    /* [in] */ FLOAT cumulativeExpansion,
    /* [in] */ FLOAT cumulativeRotation)
{
    m_cCompletedEventCount ++;

    m_fX = x;
    m_fY = y;


    if (m_hWnd){
        if (fExtrapolating){
            //Inertia Complete, stop the timer used for processing
            KillTimer(m_hWnd,drawable->GetIndex());
        }else{
            // Setup velocities for inertia processor
            float vX, vY, vA = 0.0f;
            m_pManip->GetVelocityX(&vX);
            m_pManip->GetVelocityY(&vY);
            m_pManip->GetAngularVelocity(&vA);

            drawable->SetUpInertia();

            // Set up the touch coordinate data
            m_pInert->put_InitialVelocityX(vX / 100);
            m_pInert->put_InitialVelocityY(vY / 100);

            // Start a timer
            SetTimer(m_hWnd, drawable->GetIndex(), 50, 0);

            // Reset sets the initial timestamp
            pInert->Reset();
        }
    }
}
```

Figure 10 **Initializing Screen Boundaries**

```
void Drawable::SetUpInertia(void){
(...)

    // Reset sets the  initial timestamp
    pInert->put_DesiredDeceleration(.1f);

    RECT rect;
    GetClientRect(m_hWnd, &rect);

    int width = rect.right - rect.left;
    int height = rect.bottom - rect.top;

    int wMargin = width  * .1;
    int hMargin = height * .1;

    pInert->put_BoundaryLeft(rect.left * 100);
    pInert->put_BoundaryTop(rect.top * 100);
    pInert->put_BoundaryRight(rect.right * 100);
    pInert->put_BoundaryBottom(rect.bottom * 100);

    pInert->put_ElasticMarginTop((rect.top - hMargin) * 100);
    pInert->put_ElasticMarginLeft((rect.left + wMargin) * 100);
    pInert->put_ElasticMarginRight((rect.right - wMargin) * 100);
    pInert->put_ElasticMarginBottom((rect.bottom + hMargin) * 100);

...
}
```

Looking Forward

Using the Windows Touch API is an effective way to add value to existing applications and is a great way to make your applications stand out. Taking extra time to address the context that your application will be used in allows you to make the most of the Windows Touch API. If you take into consideration the mobility and usability requirements of your application, the application becomes more intuitive, and users need less time to discover its functionality. (Additional resources, including the complete documentation reference for Windows Touch, can be found on MSDN at msdn.microsoft.com/library/dd562197(VS.85).aspx).

With the release of Windows Presentation Framework (WPF) and .NET 4, Microsoft will support managed development using controls that enable multiple contact points. If you are a developer working with managed code looking to enhance your application with multiple-input support, this release is worth checking out. Currently, examples of managed Windows Touch wrappers for C# are included in the Windows SDK. ■

GUS "GCLASSY" CLASS *is a programming writer/evangelist for Microsoft, where he has worked on Windows Touch, Tablet PC and Microsoft's DRM systems. He discusses developer gotchas and offers programming examples on his blog at gclassy.com.*

THANKS to the following technical expert for reviewing this article: *Xiao Tu*

IGOR OSTROVSKY

Data-Parallel Patterns and PLINQ

Multicore processors are now ubiquitous on mainstream desktop computers, but applications that use their full potential are still difficult to write. Multicore parallelism is certainly feasible, however, and a number of popular applications have been retrofitted to provide a performance boost on multicore computers. Version 4 of the .NET Framework will deliver several tools that programmers can employ to make this task easier: a set of new coordination and synchronization primitives and data structures, the Task Parallel Library and Parallel LINQ (PLINQ). This article focuses on the last item in this list, PLINQ.

PLINQ is an interesting tool that makes writing code that scales on multicore machines much easier—provided that your problem matches a data-parallel pattern. PLINQ is a LINQ provider, so to program against it, you use the familiar LINQ model. PLINQ is very similar to LINQ to Objects, except that it uses multiple threads to schedule the work to evaluate a query. To bind a query to PLINQ instead of LINQ to Objects, you simply add an AsParallel call after the data source, as shown in the following code. This step wraps the data source with a ParallelQuery wrapper and causes the remaining extension methods in the query to bind to PLINQ rather than to LINQ to Objects.

```
IEnumerable<int> src = ...
var query =
      src.AsParallel()
      .Where(x => x % 2 == 0)
      .Select(x => Foo(x));

foreach(var x in query)
{
      Bar(x);
}
```

The same code looks like the following using the C# query syntax:

```
IEnumerable<int> src = ...
var query =
      from x in src.AsParallel()
      where x % 2 == 0
      select Foo(x);

foreach(var x in query)
{
      Bar(x);
}
```

However, putting AsParallel into a LINQ-to-Objects query does not guarantee that your program will run faster. PLINQ attempts to use appropriate algorithms to partition the data, execute parts of the query independently in parallel and then merge the results. Whether this strategy results in a performance improvement on multicore machines depends on several factors.

To benefit from PLINQ, the total work in the query has to be large enough to hide the overhead of scheduling the work on the thread pool, and the work per element should be significant enough to hide the small amount of overhead to process that element. Also, PLINQ performs best when the most expensive part of the query can be decomposed in such a way that different worker threads evaluate the expensive computation on different input elements.

In the remaining part of the article, I'll look at the types of data-parallel patterns that can be effectively parallelized using PLINQ.

Projection

Projection, mapping, Select operator—all these terms refer to the same common and naturally data-parallel operation. In a projection, you have a projection function that takes one argument and computes an answer, and you need to evaluate that function on a set of inputs.

Projection is naturally data-parallel because the projection function can be evaluated on different input elements concurrently. If the function is at least somewhat computationally expensive, PLINQ should be able to speed up the computation by distributing the work of evaluating the function among multiple cores on the machine.

For example, in the following query, PLINQ evaluates the calls to ExpensiveFunc in parallel (at least on multicore machines):

```
int[] src = Enumerable.Range(0, 100).ToArray();
var query = src.AsParallel()
             .Select(x => ExpensiveFunc(x));

foreach(var x in query)
{
      Console.WriteLine(x);
}
```

This block of code prints the values ExpensiveFunc(0), ExpensiveFunc(1) and so on to ExpensiveFunc(99) on the console. However, the values are not necessarily printed in the expected order. By default, PLINQ treats sequences as unordered, so the values are printed in an undefined order.

To tell PLINQ to treat the src array as an ordered sequence, you can use the AsOrdered operator:

```
int[] src = Enumerable.Range(0, 100).ToArray();
var query = src.AsParallel().AsOrdered()
            .Select(x => ExpensiveFunc(x));

foreach(var x in query)
{
      Console.WriteLine(x);
}
```

Now the results are printed on the screen in the expected order, from ExpensiveFunc(0) to ExpensiveFunc(99). PLINQ incurs some additional overhead for each input element to preserve the ordering, but this is typically only a modest cost.

In the cases that we've examined so far, the PLINQ query is always consumed in a for loop. In such scenarios, PLINQ sets up asynchronous workers that compute the results in the background, and the for loop waits whenever the next result is not yet ready. However, this is not the only way to consume a PLINQ query. Alternatively, you can execute the query by operators such as ToArray, ToList and ToDictionary:

```
int[] src = Enumerable.Range(0, 100).ToArray();
var query = src.AsParallel().AsOrdered()
            .Select(x => ExpensiveFunc(x));

int[] results = query.ToArray(); // The query runs here
```

Again, PLINQ makes the calls to ExpensiveFunc in parallel, speeding up the query execution. This time, however, the execution is synchronous—the entire query is completed on the one line identified in the code sample.

Instead of converting the results to an array, you could compute the sum of the results, or use min, max, average, or a user-defined aggregation of the results:

```
int[] src = Enumerable.Range(0, 100).ToArray();
var query = src.AsParallel()
            .Select(x => ExpensiveFunc(x));

int resultSum = query.Sum(); // The query runs here
```

As yet another possibility, you could execute an action for each element produced:

```
int[] src = Enumerable.Range(0, 100).ToArray();
var query = src.AsParallel()
            .Select(x => ExpensiveFunc(x));

int resultSum = query.ForAll(
      x => Console.WriteLine(x)
);
```

There is an important difference between this example, which uses ForAll, and the first example in this section, which uses a for loop. In the ForAll example, the actions execute on the PLINQ worker threads. In the for loop example, the loop body obviously executes on the thread that creates the PLINQ query.

Finally, in writing parallel projection queries, you might run into one more difficulty that is worth calling out. PLINQ achieves parallel execution by splitting the input sequence into multiple sequences and then processing the sequences in parallel. The sequence-splitting step is called "partitioning," and your choice of partitioning algorithm could have a significant impact on the performance of your queries.

PLINQ typically chooses a good algorithm to partition your input sequence, but one case where you might want to override PLINQ's choice is if the input is an array (or another type implementing IList). In such cases, the default PLINQ behavior is to partition the array statically into the same number of sections as there are cores on the machine. But if the cost of the projection element varies per element, all expensive elements could end up in one partition.

To get PLINQ to use a load-balancing partitioning algorithm for arrays (or other IList types), you can use the Partitioner.Create method, passing in a true value for the loadBalancing argument:

```
int[] src = Enumerable.Range(0, 100).ToArray();
var query = Partitioner.Create(src, true).AsParallel()
            .Select(x => ExpensiveFunc(x));

foreach(var x in query)
{
      Console.WriteLine(x);
}
```

Filtering

A slight variation of the projection pattern is filtering. Here, instead of having a projection function that computes an output from each input, you have a filtering function that decides whether a particular element should be included in the output.

For best parallel speedup, the filtering function should be computationally expensive to evaluate. In certain cases, filtering with even a cheap function might scale very well, especially when the filtering function rejects most inputs. In this sample, PLINQ prints those numbers in the range [0 to 99] on which ExpensiveFilter returns true:

```
int[] src = Enumerable.Range(0, 100).ToArray();
var query = src.AsParallel()
            .Where(x => ExpensiveFilter(x));

foreach(var x in query)
{
      Console.WriteLine(x);
}
```

As in the first projection example, the results here will be unordered. The solution to making the results ordered is the same: simply add AsOrdered after AsParallel. In fact, all the other follow-up points explained earlier about projections apply to filtering as well. This means that the query can be consumed using foreach, ToArray/ToList/ToDictionary, aggregation, or ForAll. Also, you may want to override the default partitioning scheme if your input is in an array, and static partitioning may lead to load imbalances. (These options work generally the same way for the remaining patterns in this article as well.)

Independent Actions

In the projection and filtering patterns, the expensive part of the computation is converting an input sequence into an output sequence. A simpler pattern is an expensive action that needs to be performed for each sequence element. The action does not need to return a value; it simply executes some computationally expensive and thread-safe side effect, as shown here:

```
int[] src = Enumerable.Range(0, 100).ToArray();
src.AsParallel()
.ForAll(
     x => { ExpensiveAction(x); }
);
```

For concurrency, PLINQ executes ExpensiveAction on worker threads. This means that ExpensiveAction should be computationally expensive and, even more importantly, thread safe. Since ExpensiveAction is invoked on different threads, no order is implied among the invocations.

As it turns out, this pattern is so simple that you don't need PLINQ and can simply use the Parallel.ForEach method available in the Task Parallel Library (as of .NET Framework 4). However, ForAll in PLINQ is often handy when it is combined with other PLINQ operators:

```
int[] src = Enumerable.Range(0, 100).ToArray();
src.AsParallel()
.Where(x => x%2 == 0)
.ForAll(
     x => { ExpensiveAction(x); }
);
```

Sequence Zipping

Sequence zipping is a pattern similar to projection, except that two input sequences are present rather than one. Instead of having an expensive function that converts one input element into one output element, you have an expensive function that converts one input from one sequence and one input from the other sequence into a single output element.

This pattern is supported by using the Zip LINQ-to-Objects operator introduced in .NET 4.0. You can use the operator in PLINQ as well. For the best performance, the input sequences should be in arrays or IList collections:

```
int[] arr1 = ..., arr2 = ...;
int[] results =
       arr1.AsParallel().AsOrdered()
       .Zip(
            arr2.AsParallel().AsOrdered(),
            (arr1Elem, arr2Elem) => ExpensiveFunc(arr1Elem, arr2Elem))
       .ToArray();
```

In fact, you might notice that if you have the input sequences in arrays, the Zip operator can be conveniently restated as a projection:

```
int[] arr1 = ..., arr2 = ...;
int length = Math.Min(arr1.Length, arr2.Length);
int[] results =
       ParallelEnumerable.Range(0, length).AsOrdered()
       .Select(index => ExpensiveFunc(arr1[index], arr2[index]))
       .ToArray();
```

Regardless of which implementation of the pattern you choose, remember that this type of workload can be nicely sped up with PLINQ.

Reduction

Reduction, also known as aggregation or folding, is an operation in which elements of a sequence are combined until you are left with a single result. Sum, average, min and max are a few popular reductions, and these reductions are so frequently used that they are directly supported by PLINQ as operators (Sum, Average, Min and Max). However, these operators perform little work per element, so in PLINQ they are usually used in queries that also contain an expensive computation—a projection or a filter, for example. One possible exception to this rule is a min or max operation with an expensive comparison function. However, if the reduction function is an expensive operation, a reduction can be a parallel workload in its own right.

There are several different overloads of Aggregate, but I will not discuss them in this article because of space constraints. (See blogs.msdn.com/pfxteam/archive/2008/01/22/7211660.aspx and blogs.msdn.com/pfxteam/archive/2008/06/05/8576194.aspx for a more thorough discussion of PLINQ reductions.) The most general overload of Aggregate has this signature:

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
    this ParallelQuery<TSource> source,
    Func<TAccumulate> seedFactory,
    Func<TAccumulate, TSource, TAccumulate> updateAccumulatorFunc,
    Func<TAccumulate, TAccumulate, TAccumulate> combineAccumulatorsFunc,
    Func<TAccumulate, TResult> resultSelector)
```

And here is how you'd use it to implement a parallel Average operator:

```
public static double Average(this IEnumerable<int> source)
{
      return source.Aggregate(
             () => new double[2],
             (acc, elem) => {
                   acc[0] += elem; acc[1]++; return acc;
             },
             (acc1, acc2) => {
                   acc1[0] += acc2[0]; acc1[1] += acc2[1]; return acc1;
             },
             acc => acc[0] / acc[1]);
}
```

Each PLINQ worker initializes its accumulator by using seedFactory, so it will get its own array of two double values. Then the worker processes part of the input sequence, updating its accumulator with each element by using updateAccumulatorFunc. Next, different workers combine their accumulators by using combineAccumulatorsFunc, and finally the single accumulator is converted into the return value by using resultSelector.

Keep in mind that although the parallel Average operator sample is convenient for explaining the semantics of the Aggregate operator, its work per element (two additions) is probably too low to make parallelization worthwhile. Scenarios with a more expensive reduction function often come up in the real world, though.

Sorting

LINQ supports sorting via the OrderBy operator, and PLINQ naturally implements the sort by using a parallel algorithm. Usually, the sorting algorithm seems to get a decent speedup against the LINQ-to-Objects sort (perhaps two to three times on a four-core machine). However, one fact to remember is that the LINQ-to-Objects sorting model imposes a fairly heavy interface on OrderBy. The key selector is mandatory and passed in as a delegate rather than an expression tree, so PLINQ does not ignore the key selector even if it is an identity function, x => x. Consequently, PLINQ manipulates key-value pairs, even in cases where keys are equal to values. Also, as a result of the functional nature of LINQ, PLINQ cannot sort the sequence in place even if it is in an array because that would destroy the original sequence.

With this in mind, if you are using the LINQ-to-Objects OrderBy operator, you should be able to speed up your query by using PLINQ. However, if you need to sort only an array of integers, an in-place sort like Array.Sort is likely to be faster than PLINQ's OrderBy. If you need to speed up an in-place sort, you might have

SharePoint Live
Virtual Conference & Expo

ON-DEMAND UNTIL
FEBRUARY 5, 2010

NOW AVAILABLE ON DEMAND!

SharePoint 2010 Virtual Conference & Expo
… and it's FREE!

We've heard from you that it's hard to get out of the office these days and training dollars are limited, so we're bringing you a FREE virtual event on one of the most anticipated software releases—SharePoint 2010—now available On-Demand from the convenience of your office.

Keynote Address: Arpan Shah, *Director, SharePoint Product Management team, Microsoft*

Enjoy Sessions On:
- SharePoint Developer Roadmap
- A Lap Around The New Security Features in SharePoint 2010
- Silverlight and SharePoint
- Architecting a Robust Global SharePoint Infrastructure
- Setting up a SharePoint 2010 Lab on Microsoft HyperV

Learn more and register for free at SharePointVCX.com
use promo code NQ9P03

PLATINUM SPONSORS AvePoint Critical Path Training MIMOSA SYSTEMS QUEST SOFTWARE

GOLD SPONSOR metalogix

SILVER SPONSORS Double-Take EMC² where information lives K2

www.SharePointVCX.com

to implement your own parallel sorting algorithm on top of Task Parallel Library.

One-to-Many Transformation

A projection converts every input element into exactly one output element. Using a filter, you can convert every input element into zero or one output elements. But what if you want to be able to generate an arbitrary number of outputs from each input? PLINQ supports that case as well through the SelectMany operator. Here's an example:

```
IEnumerable<int> inputSeq = ...
int[] results =
      inputSeq.AsParallel()
      .SelectMany(input => ComputeResults(input))
      .ToArray();
```

This code calls ComputeResults on every element in the input sequence. Each ComputeResults returns an IEnumerable type (for example, an array) that contains zero, one, or multiple results. The output of the query contains all the results returned by the ComputeResults calls.

Because this pattern is a little less intuitive than the other patterns in this article, let's take a look at a concrete example of its use. The one-to-many pattern could implement a search algorithm for a problem such as the familiar N-Queens problem (find all placements of queens on a chessboard so that no two queens attack each other). The input sequence would be a sequence of chessboards with a few queens already in place. Then you would use a query with the SelectMany operator to find all N-Queens solutions that can be reached starting from any of the initial states in the input:

```
IEnumerable<ChessboardState> initStates = GenerateInitialStates();
ChessboardState[] solutions =
      initStates.AsParallel()
      .SelectMany(board => board.FindAllSolutions())
      .ToArray();
```

More Complex Queries

The PLINQ patterns discussed in this article are all short query snippets, generally with one or two operators. Of course, different patterns can be used together in one query. The following query combines a filter, a projection and independent actions:

```
int[] src = Enumerable.Range(0, 100).ToArray();
src.AsParallel()
            .Where(x => ExpensiveFilter(x))
            .Select(x => ExpensiveFunc(x));
            .ForAll(x => { ExpensiveAction(x); });
```

PLINQ will effectively parallelize this query regardless of whether the filtering, projection and side-effect actions are about the same cost, or if one of them dominates the execution time. And like LINQ to Objects, PLINQ does not materialize the result set after each operator. So, PLINQ won't execute the Where for the entire sequence, store the filtered sequence, do the Select and finally the ForAll. Operations are combined as much as possible, and in simpler queries a worker will "flow" an input element through the entire query before moving on to the next element.

In addition to combining the parallel patterns with one another, you can combine them with any of the LINQ operators. PLINQ strives to maintain parity with LINQ to Objects, so all LINQ operators are available. Even though PLINQ will execute just about any LINQ-to-Objects query, it does not necessarily make the query faster. Some operators and query shapes do not parallelize well, if at all. Ideally, the most computationally expensive part of the query should have the form of one of the parallel patterns from this article.

In addition to combining the parallel patterns with one another, you can combine them with any of the LINQ operators.

One thing to be aware of is that in some complex queries, PLINQ decides to execute parts of the query sequentially instead of using potentially expensive algorithms needed for parallel execution. That may not be what you want, especially if your query contains an expensive delegate that would dominate the execution time anyway. In this code sample, PLINQ decides to execute the ExpensiveFunc() delegates sequentially:

```
int[] src = Enumerable.Range(0, 100).ToArray();
int[] res = src.AsParallel()
             .Select(x => ExpensiveFunc(x))
             .TakeWhile(x => x % 2 == 0)
             .ToArray();
```

You can solve this issue in two ways. You can give PLINQ a hint to execute the query in parallel, even if potentially expensive algorithms have to be used:

```
int[] src = Enumerable.Range(0, 100).ToArray();
int[] res = src.AsParallel()
             .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
             .Select(x => ExpensiveFunc(x))
             .TakeWhile(x => x % 2 == 0)
             .ToArray();
```

Or you can decompose the query so that PLINQ executes only the expensive part of the query and LINQ to Objects executes the rest. You can use the AsSequential operator in a PLINQ query to get subsequent operators to bind to LINQ to Objects:

```
int[] src = Enumerable.Range(0, 100).ToArray();
int[] res = src.AsParallel()
             .Select(x => ExpensiveFunc(x))
             .AsSequential()
             .TakeWhile(x => x % 2 == 0)
             .ToArray();
```

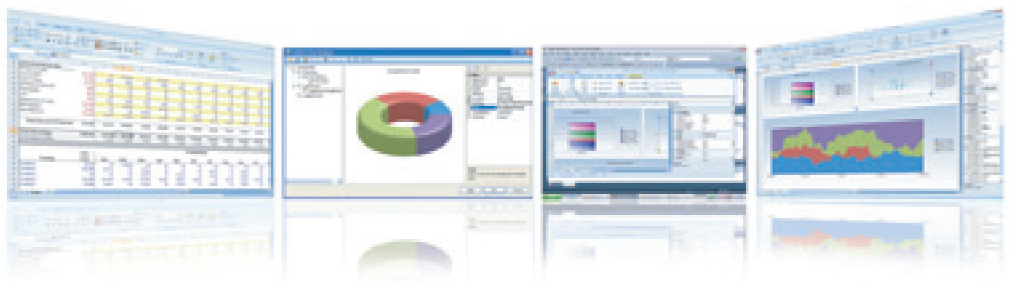
Make the Most of PLINQ

Writing multicore applications can be hard, but it does not always have to be. PLINQ is a useful tool to have in your toolbox to speed up data-parallel computations when you need to. Remember the patterns, and use them appropriately in your programs. ■

Igor Ostrovsky *is a software development engineer on the Parallel Computing Platform team at Microsoft. He is the primary developer for PLINQ.*

Thanks to the following technical experts for reviewing this article: *Michael Blome*

World's Best-Selling, Award-Winning Microsoft®
Excel® Compatible Spreadsheet ... Now with Charting!
SPREAD



Rapidly Develop Great Looking Spreadsheet Dashboards

Demand the best. When you need high-performance, fully-customizable spreadsheet components, join the professional developers around the world who consistently turn to FarPoint Spread for powerful, extendable spreadsheet solutions. Download your free evaluation today.

- ✔ Microsoft® Excel® Compatibility
- ✔ Familiar Spreadsheet Experience
- ✔ Extensible Object Model
- ✔ Familiar Formula Syntax and Functions
- ✔ Charts for Clear Data Visualization
- ✔ Spreadsheets connected to Corporate Data

GrapeCity® FarPoint
www.FarPointSpread.com

NEW!
RELEASE 5.0
SPREAD
for Windows Forms

Dundas
Data Visualization

The Evolution of Dashboarding

From the **industry leader** in data visualization technology comes an easy-to-integrate, customizable, turnkey **dashboard solution.**

- Rapid dashboard development
- Flexible integration and customization
- The latest Silverlight 3.0 technology

Dundas Dashboard

Powered by
Microsoft Silverlight

For more information please visit
www.dundas.com/dashboard
(416) 467-5100 • (800) 463-1492