The Microsoft Journal for Developers

msdn® magazine

AUGUST 2009 VOL 24 NO 8

msdn

Microsoft

Microsoft msdn magazine
AUGUST 2009 VOLUME 24 NUMBER 8

LUCINDA ROWLEY Director

EDITORIAL: mmeditor@microsoft.com
HOWARD DIERKING Editor-in-Chief

WEB SITE
MICHAEL RICHTER Webmaster

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

MSDN Magazine (ISSN # 1528-4859) is published monthly by TechWeb, a division of United Business Media LLC., 600 Community Drive, Manhasset, NY 11030 516-562-5000. Periodicals Postage Paid at Manhasset, NY and at additional mailing offices. Back issue rates: U.S. $10. All others: $12. Basic one-year subscription rates: U.S. $45. Canada and Mexico $55. Registered for GST as TechWeb, a division of United Business Media LLC., GST No. R13288078, Customer No. 2116057 Canada Post: Publications Mail Agreement #40612608. Canada Returns to be sent to Bleuchip International, P.O. Box 25542, London, ON N6C 6B2. All other foreign orders $70, must be prepaid in U.S. dollars drawn on a U.S. bank. Circulation Department, MSDN Magazine, P.O. Box 1081, Skokie, IL 60076-8081, fax 847-763-9583. Subscribers may call from 8:00 AM to 4:00 PM CST M-F. In the U.S. and Canada 888-847-6188; all others 847-763-9606. U.K. subscribers may call Jill Sutcliffe at Parkway Gordon 01-49-1875-386. Manuscript submissions and all other correspondence should be sent to MSDN Magazine, 6th Floor, 1290 Avenue of the Americas, New York, NY 10104. Copyright © 2009 Microsoft Corporation. All rights reserved; reproduction in part or in whole without permission is prohibited.

POSTMASTER: Send address changes to MSDN Magazine, P.O. Box 1081, Skokie, IL 60076-8081

READERS: Order, renew, or make payment for subscriptions; order back issues; and submit customer service inquiries via the Web at http://msdn.microsoft.com/msdnmag/service.

PUBLISHER:
Jill Thiry jthiry@techweb.com

United Business Media

BPA WORLDWIDE CONSUMER

ADVERTISING SALES: 785-838-7573/dtimmons@techweb.com

David Timmons Associate Publisher
Jon Hampson Regional Account Director, Northeast US and EMEA
Ed Day Regional Account Manager, West & South US and APAC
Brenner Fuller Strategic Account Manager, West & South US and APAC
Michele Hurabiell-Beasley Strategic Account Manager, Key Accounts
Julie Thibault Strategic Account Manager, Northeast US and EMEA
Meagon Marshall Account Manager
Pete C. Scibilia Production Manager / 516-562-5134
Wright's Reprints 877-652-5295
ubmreprints@wrightsreprints.com

ONLINE SERVICES: 415-947-6158/mposth@techweb.com

Mark Posth Community Manager
Meagon Marshall Online Specialist

AUDIENCE SERVICES: 516-562-7833/kmcaleer@techweb.com

Karen McAleer Audience Development Director

SUBSCRIBER SERVICES: 800-677-2452/lawrencecs@techinsights.com

TechWeb, a division of United Business Media LLC.–The Global Leader in Business Technology Media

Tony L. Uphoff CEO
Bob Evans SVP and Content Director
Eric Faurot SVP, Live Events Group
Joseph Braue SVP, Light Reading Communications Group
John Siefert VP and Publisher, InformationWeek and TechWeb Network
Scott Vaughan VP, Marketing Services
John Ecke VP, Financial Technology Group
Jill Thiry Publisher, MSDN Magazine and TechNet Magazine
John Dennehy General Manager
Fritz Nelson Executive Producer, TechWeb TV
Scott Popowitz Senior Group Director, Audience Development
Beth Rivera Senior Human Resources Manager

Printed in the USA

Editor's Note

The Tale of Two Database Schemas

I recently had the opportunity to author the editor's note for *TechNet Magazine*. I have to say that, being a developer, addressing an audience of IT professionals was a bit daunting. Both disciplines are vital to any business, but many times their paths only cross when something is broken. However, I believe that when it comes to the management of data, both developers and IT professionals need to be involved up front in planning solutions. Given that the theme of that particular *TechNet Magazine* issue was business intelligence and that the theme of this issue of *MSDN Magazine* is data, I'll address some of the main points I made in that editor's note but more from the developer perspective.

When you get right down to it, the role that software generally plays in most businesses is to get, process, and store data. Therefore, while we may have all sorts of high-level discussions and debates around architectural patterns and object-oriented heuristics, the fact remains that the most elegantly designed application is still effectively taking data from somewhere, doing something to it and putting it somewhere else.

Now don't get me wrong—I'm not suggesting that design heuristics we argue so fervently over are immaterial. After all, a Ford Model T and a Lamborghini Diablo both accomplish the task of moving people from one place to another; but given a choice between the two, it's pretty clear in my mind which one I would choose. Instead, I'm suggesting that we put the same level of thinking and technology consideration into data structure and management that we do for our class models. Furthermore, I'm not just talking about relational schema design or whether or not to use stored procedures. I'm talking much more generally about understanding how the business needs to consume data in a way that provides meaningful value.

One prime example of poor application planning and design as it relates to data is found in reporting. Forget the object-relational impedance mismatch for a moment—the transactional-reporting impedance mismatch seems to be one of those problems that rears its head in just about every business support system that we touch. And in nearly every case, the transactional concerns win by a landslide. Reporting requirements become limited by the inherent complexity and performance limitations found in the highly normalized database schemas of well-designed transactional systems. Even when well-meaning system designers try to accommodate both sets of concerns in the application schema, the result generally does slightly better at meeting the reporting requirements, and it does so at the great expense of the transactional requirements.

So what's the solution? First, get comfortable with the reality that there is not, nor will there ever be, a relational database design that will successfully meet both transactional and reporting requirements—at least not in a sustained way. From there, start assuming that your system should have at least two database schemas—one highly normalized schema that is optimized for processing transactions and one denormalized schema that is optimized for reporting and for mining. What I'm describing is known as the difference between relational and dimensional data modeling. For a great resource on getting started with dimensional modeling, check out *The Data Warehouse Toolkit* by Ralph Kimball and Margy Ross (Wiley, 2002).

Freeing yourself from the burden of trying to build a single relational schema that takes into account both reporting and transactional concerns will enable you to truly optimize both new schemas according to how they are actually used. Put another way, you will have effectively shifted your problem from a design problem to an extract, transform and load (ETL) problem—and the latter is generally a much more straightforward type of problem to solve. Additionally, I think that once you dig into some of the technologies that support dimensional modeling, from online analytical processing to data mining, you may just find that implementing reporting requirements can actually become a great deal more fun.

At the very least, think of it as ensuring that your databases follow the single responsibility principle.

[illegible]

THANKS TO THE FOLLOWING MICROSOFT TECHNICAL EXPERTS FOR THEIR HELP WITH THIS ISSUE: *Mike Barnett, Bob Brumfield, Pablo Castro, Jeff Currier, Diego Dagum, Nelly Delgado, Jeff Derstadt, Manuel Fahndrich, Shawn Farkas, Elisa Flasko, Shawn Hernan, Tim Heuer, Carl Perry, Ayende Rahien, and Blaine Wastell.*

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

Data Snapshots, Subversion, Source Code Organization and More

Save, Organize, and Annotate Snapshots of Your Data

Virtually all computer programs allow users to serialize their current working state to a portable, self-contained file, which can then be opened from another computer that has the same software installed. In less formal terms, most programs have a Save option that lets you persist your work to a file. Wouldn't it be nice if such functionality were possible with database queries? Imagine if you could run one or more SELECT statements and then save the resultsets to a file. Such files could be used to store and review historical data, to serve as a "before" snapshot of data before performing a bulk modification, and to allow testers, analysts, and other stakeholders the ability to view a precise set of data without needing access to the database.

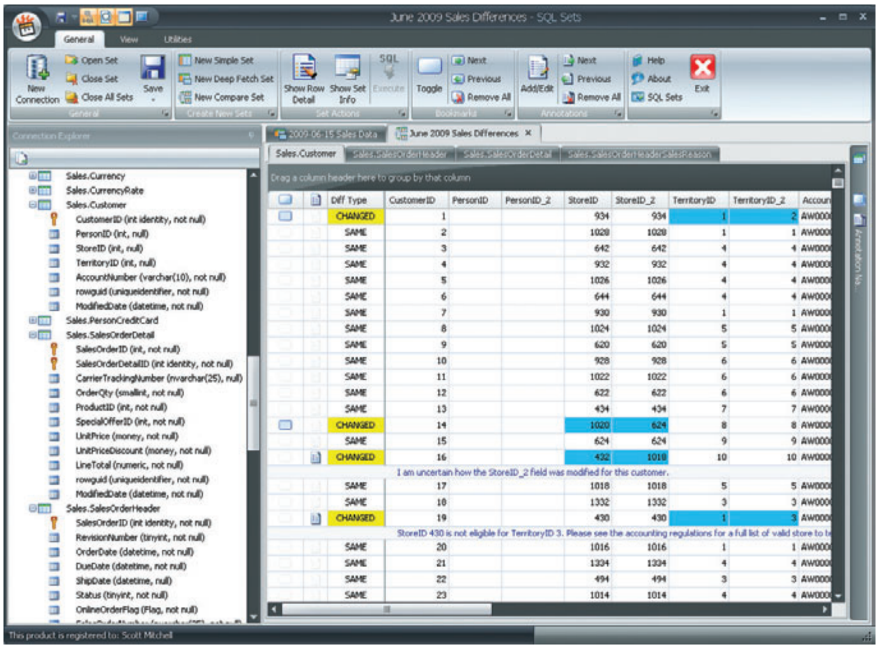
SQL Sets version 1.5 is an ingenious piece of software that brings such functionality to the database world. SQL Sets has three primary use cases: storing the results of one or more queries into a portable data document file, or "set"; viewing, organizing, and annotating the data stored in a set; and comparing the data between two sets. A set is stored on disk as a self-contained, static, read-only snapshot of data at a specific point in time. It stores the raw data and has no dependency on the database. This means that sets can be moved to other computers, viewed, and shared among team members without having to grant anyone access to the database.

Creating a set file is a breeze. Launch the SQL Sets application, connect to the database that contains the data of interest, and specify the query (or queries) whose data you want to capture. You can write these queries yourself or have SQL Sets build them for you by selecting one or more tables from the Connection Explorer window. Once the queries have been specified, click the Save icon to save the data returned from the queries to a set file.

When you view a set, its data is displayed in a grid that supports sorting, filtering, and grouping by column. Rows can be bookmarked for quick access and annotated to include notes or other information about the row. All of these features are available through pointing and clicking. Also, the person viewing the set does not need to be familiar with SQL syntax. What's more, with the click of a button you can export the set's data to Microsoft Excel or to HTML. There's also an Export to ADO.NET DataSet option, which translates the set's schema and data into an XML serialized ADO.NET DataSet that you can use in a .NET application.

SQL Sets also allows you to compare two sets. Start by selecting the two set files to compare, then indicate whether to show only rows that are the same in both, rows that are different, rows that are in one set but not the other, or any combination thereof. SQL Sets then loads the data and clearly highlights those rows with differences.

SQL Sets makes it remarkably easy to take snapshots of database data and



- Sales.Currency
- Sales.CurrencyRate
- Sales.Customer
  - CustomerID (int identity, not null)
  - PersonID (int, null)
  - StoreID (int, null)
  - TerritoryID (int, null)
  - AccountNumber (varchar(10), not null)
  - rowguid (uniqueidentifier, not null)
  - ModifiedDate (datetime, not null)
- Sales.PersonCreditCard
- Sales.SalesOrderDetail
  - SalesOrderID (int, not null)
  - SalesOrderDetailID (int identity, not null)
  - CarrierTrackingNumber (nvarchar(25), null)
  - OrderQty (smallint, not null)
  - ProductID (int, not null)
  - SpecialOfferID (int, not null)
  - UnitPrice (money, not null)
  - UnitPriceDiscount (money, not null)
  - LineTotal (numeric, not null)
  - rowguid (uniqueidentifier, not null)
  - ModifiedDate (datetime, not null)
- Sales.SalesOrderHeader
  - SalesOrderID (int identity, not null)
  - RevisionNumber (tinyint, not null)
  - OrderDate (datetime, not null)
  - DueDate (datetime, not null)
  - ShipDate (datetime, null)
  - Status (tinyint, not null)
  - OnlineOrderFlag (Flag, not null)

Diff Type	CustomerID	PersonID	PersonID_2	StoreID	StoreID_2	TerritoryID	TerritoryID_2	Accoun
CHANGED	1			934	934	1	2	AW000
SAME	2			1028	1028	1	1	AW000
SAME	3			642	642	4	4	AW000
SAME	4			932	932	4	4	AW000
SAME	5			1026	1026	4	4	AW000
SAME	6			644	644	4	4	AW000
SAME	7			930	930	1	1	AW000
SAME	8			1024	1024	5	5	AW000
SAME	9			620	620	5	5	AW000
SAME	10			928	928	6	6	AW000
SAME	11			1022	1022	6	6	AW000
SAME	12			622	622	6	6	AW000
SAME	13			434	434	7	7	AW000
CHANGED	14			1020	624	8	8	AW000
SAME	15			624	624	9	9	AW000
CHANGED	16			432	1018	10	10	AW000
	I am uncertain how the StoreID_2 field was modified for this customer.							
SAME	17			1018	1018	5	5	AW000
SAME	18			1332	1332	3	3	AW000
CHANGED	19			430	430	1	3	AW000
	StoreID 430 is not eligible for TerritoryID 3. Please see the accounting regulations for a full list of valid store to t							
SAME	20			1016	1016	1	1	AW000
SAME	21			1334	1334	4	4	AW000
SAME	22			494	494	3	3	AW000
SAME	23			1014	1014	4	4	AW000

This product is registered to: Scott Mitchell

SQL Sets

All prices confirmed at press time and 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.

to allow team members to review, sort, filter, group, bookmark, and annotate the snapshot data. These snapshots can also serve as archived data or as "before" and "after" snapshots when performing a bulk modification.

Price: $149

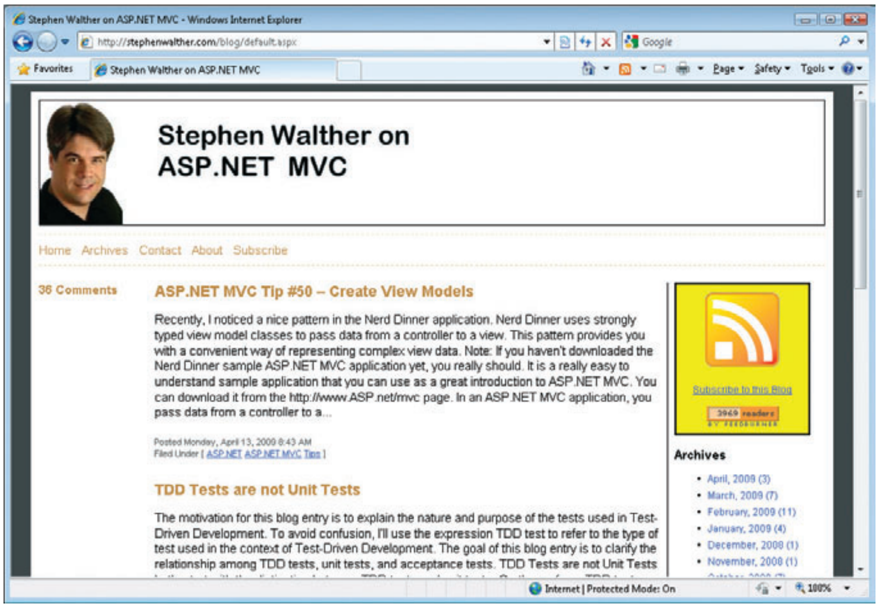
sqlsets.com

Blogs of Note

At MIX09, Microsoft released ASP.NET MVC version 1.0, a framework for creating ASP.NET Web applications using a Model-View-Controller pattern. ASP.NET MVC offers developers precise control over the markup emitted from Web pages; a much cleaner separation of presentation and business logic concerns; better testability; and human-readable, terse, SEO-friendly URLs. Moving from the ASP.NET Web Forms model to ASP.NET MVC requires a shift in thinking and problem solving. Web Forms allow ASP.NET developers to almost forget about the client/server nature of the Web and to think of HTML, JavaScript, and CSS as low-level details that are abstracted away. ASP.NET MVC puts the distinction between the client and the server into sharp focus and requires a working knowledge of HTML and client-side script.

Developers who are interested in learning or are currently using ASP.NET MVC should check out the tips, tutorials, and sample chapters available on **Stephen Walther's Blog**. There you'll find more than 50 tips on ASP.NET MVC. Each tip shows how to perform a very specific task and provides step-by-step instructions with detailed code snippets and screen shots. For example, Tip #41 is titled, "Create Cascading Drop-Down Lists with Ajax," and walks through three different ways to create such lists in an ASP.NET MVC application.

Stephen's blog also includes a number of end-to-end tutorials that illustrate how to create a particular type of application using ASP.NET MVC. For instance, there's a six-part tutorial on building an online message board application and a five-part tutorial on creating a family video Web site. The blog is also home to the rough drafts of chapters from Stephen's book, *ASP.NET MVC Framework Unleashed,* forthcoming from Sams.


Stephen Walther on ASP.NET MVC - Windows Internet Explorer

http://stephenwalther.com/blog/default.aspx

Stephen Walther on ASP.NET MVC

Home Archives Contact About Subscribe

36 Comments

ASP.NET MVC Tip #50 – Create View Models

Recently, I noticed a nice pattern in the Nerd Dinner application. Nerd Dinner uses strongly typed view model classes to pass data from a controller to a view. This pattern provides you with a convenient way of representing complex view data. Note: If you haven't downloaded the Nerd Dinner sample ASP.NET MVC application yet, you really should. It is a really easy to understand sample application that you can use as a great introduction to ASP.NET MVC. You can download it from the http://www.ASP.net/mvc page. In an ASP.NET MVC application, you pass data from a controller to a...

Posted Monday, April 13, 2009 8:43 AM
Filed Under [ASP.NET ASP.NET MVC Tips]

TDD Tests are not Unit Tests

The motivation for this blog entry is to explain the nature and purpose of the tests used in Test-Driven Development. To avoid confusion, I'll use the expression TDD test to refer to the type of test used in the context of Test-Driven Development. The goal of this blog entry is to clarify the relationship among TDD tests, unit tests, and acceptance tests. TDD Tests are not Unit Tests

Subscribe to this Blog

3969 readers

Archives
- April, 2009 (3)
- March, 2009 (7)
- February, 2009 (11)
- January, 2009 (4)
- December, 2008 (1)
- November, 2008 (1)

Stephen Walther's Blog

In addition to maintaining his blog, Stephen also writes many of the tutorials and How-To videos for ASP.NET MVC on the official ASP.NET Web site, asp.net.

Price: Free

stephenwalther.com/blog

The Easy Way to Install and Configure Subversion

Regardless of how many developers are employed, every company that creates software should be using source control. Over the years, I've helped a number of independent consultants and small companies set up and configure source control systems. The first step is selecting which source control system to use. There are a variety of free and commercial source control systems available; Wikipedia lists more than 50 offerings in its "List of revision control software" entry. One of the more popular source control systems is Subversion, a free, open-source option that was first released in 2000. Subversion has a strong online community and is the source control system of choice for many open-source projects. It is also a popular source control system within the enterprise.

Although installing, configuring, and managing Subversion is not rocket science, these processes are not the most intuitive or user-friendly, either. For instance, in order to access Subversion through HTTP, you need to also install and configure the Apache Web server. Creating user accounts involves editing a particular text file. And because Subversion lacks a graphical user interface, much of the configuration and maintenance must be done from the command line. The good news is that installing, configuring, and managing Subversion is a breeze with **VisualSVN Server** version 1.7.1, a free product from the same company that makes VisualSVN, a Visual Studio plug-in that integrates source control through Subversion into the Visual Studio IDE. (VisualSVN was reviewed in the 2008 Launch issue: msdn.microsoft.com/en-us/magazine/cc164246.aspx.)

With VisualSVN Server, there's no need to download and install Subversion and Apache, or to interface with Subversion through the command line or to tinker with its configuration files. Installing VisualSVN Server automatically installs the latest versions of Subversion and Apache for you. During the installation process, you are prompted for key Subversion and Apache settings, such as the location where Subversion should store its repositories, what port it should use, whether to support secure HTTPS connections, and whether authentication should be handled by Subversion or Windows. VisualSVN Server then applies these settings to the

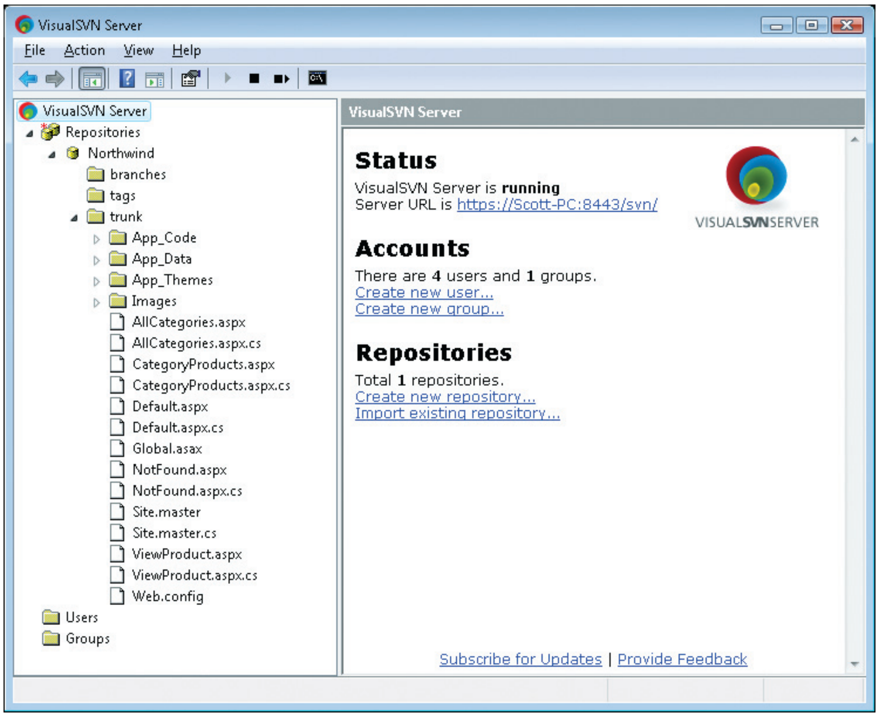
Subversion and Apache configurations on your behalf.

Once it is installed, use the VisualSVN Server Manager to view and manage repositories, users, and groups. With a few clicks of the mouse, you can create new repositories, manage users, specify permissions and other security settings, and manage the files in a repository. Without VisualSVN Server, these tasks would have to be done from the command line or by modifying configuration files. VisualSVN Server also offers a graphical interface for specifying hooks, which are programs that run in response to certain source control events, such as check-in and check-out. And because VisualSVN Server installs and configures Apache, you can view the contents of repositories from your Web browser and access the repository and check in items over the Internet.

If you plan to install Subversion in a Windows environment, there's no reason not to use VisualSVN. It greatly simplifies installing and managing Subversion and is available for free.

Price: Free

visualsvn.com/server



Status

VisualSVN Server is **running**
Server URL is https://Scott-PC:8443/svn/

Accounts

There are **4** users and **1** groups.
Create new user...
Create new group...

Repositories

Total **1** repositories.
Create new repository...
Import existing repository...

Subscribe for Updates | Provide Feedback

VisualSVN Server

Automatically Organize Your Source Code

Code refactoring, or "cleaning up" code, can greatly improve the readability and understandability of the source code, thereby making the application more maintainable and updatable. Some changes, such as renaming a variable to a more fitting name or moving a block of code into a new function, make the code easier to understand. Other changes, such as adding white space or rearranging the methods in a file so that they are in alphabetical order, make the code easier to read.

Manually refactoring code can be a tedious process. Fortunately, there are tools to help automate many common refactoring tasks. For instance, Visual Studio has a Refactor menu that offers one-click access to common refactoring tasks. Another useful tool is **NArrange** (version 0.2.7), which automatically organizes C# and Visual Basic source code into a more readable format. NArrange can be run from the command line or from within Visual Studio to arrange a single file, all code files in a specified directory, or all code files in a Visual Studio Project or Solution. When invoked, NArrange begins by saving a backup of the files that will be modified. Next, it parses each of the specified files, rearranges their contents based on the configuration options, and then writes the rearranged source code back to disk.

By default, NArrange groups constructors, fields, properties, methods, and events into regions and alphabetizes the members within each region. Consecutive blank lines are removed, tabs are converted into spaces, and the using or Import directives within a class file are consolidated and sorted. However, NArrange's formatting and parsing rules can be customized. For example, you can instruct NArrange to not use regions and to not delete consecutive blank lines.

NArrange provides a fast and easy way to organize source code into a much more readable format. Use it to beautify your code or to reformat legacy code you've inherited to make it more readable. NArrange can also be used to ensure a consistent formatting style among developers in a team setting.

Price: Free, open source

narrange.net

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

Code Contracts

Often there are certain facts about code that exist only in the developer's head or, if you're lucky, in the code comments. For example, method Foo assumes that the input parameter is always positive and fails to do anything useful on negative numbers, so you had better make sure you're calling it with a positive number. Or class Bar guarantees that property Baz is always non-null, so you don't have to bother checking it. If you violate one of these conditions, it can lead to difficult-to-find bugs. In general, the later a bug is found, the more difficult it is to fix. Wouldn't it be great if there were a way to encode and check this kind of assumption to make it easier to catch bugs, or even help prevent you from writing them in the first place?

This is where programming with contracts comes in. The practice was first introduced by Bertrand Meyer with the Eiffel programming language. The basic idea is that classes and methods should explicitly state what they require and what they guarantee if those requirements are met, i.e., their contracts. Ideally, they are decided upon at design time, and not tacked on after development has already happened. These contracts are not only human-readable, but they can also be picked up by tooling that can perform runtime checking or static verification, or perhaps include them in generated documentation.

For those familiar with Debug.Assert, you may be thinking this is a solved problem. But Debug.Assert only allows you to express that a particular condition should be true at a particular point in the code. Code contracts allow you to declare once that a particular condition should hold any time certain events occur, such as every exit point from a method. They can also express invariants that should be true class-wide, or requirements and guarantees that should exist even for subclasses.

The Common Language Runtime (CLR) team is introducing a library to allow programming with contracts in the Microsoft .NET Framework 4. Adding them as a library allows all .NET languages to take advantage of contracts. This is different from Eiffel or Spec#, a language from Microsoft Research (research.microsoft.com/en-us/projects/specsharp/), where the contracts are baked into the language.

This article will share some of the best practices that the Base Class Libraries (BCL) team devised as it added the code contract libraries and started to take advantage of them in its own code. This article is based on a prerelease version of code contracts that is more recent than the beta 1 version, so there may be some details that are different from the released version. But the general principles should remain the same.

Parts of the Code Contracts System

There are four basic parts that are involved in using code contracts in the .NET Framework 4. The first part is the contract library. Contracts are encoded using static method calls defined in the Contract class in the System.Diagnostics.Contracts namespace in mscorlib.dll. Contracts are declarative, and these static calls at the beginning of your methods can be thought of as part of the method signature. They are methods, and not attributes, because attributes are very limited in what they can express, but the concepts are similar.

The second part is the binary rewriter, ccrewrite.exe. This tool modifies the Microsoft Intermediate Language (MSIL) instructions of an assembly to place the contract checks where they belong. With the library, you declare your contracts at the beginning of the method. Ccrewrite.exe will place checks for the method guarantees at all return points from the method and will inherit contracts from other locations, such as base classes or interfaces. This is the tool that enables runtime checking of contracts to help you debug your code. Without it, contracts are simply documentation and shouldn't be compiled into your binary.

The third part is the static checker, cccheck.exe, that examines code without executing it and tries to prove that all of the contracts are satisfied. This tool is used only for advanced scenarios where the programmer is willing to go through the effort required to track down unproven contracts and add extra information as needed. Attributes exist that let you specify which assemblies, types, or members should be checked. It is generally a good plan to start small and then expand the scope for your static analysis.

Running the rewriter and adding many extra checks to your assemblies is beneficial to help you catch errors and write quality code. But those checks can slow down your code, and you don't always want to include them in your shipping assemblies. However, if you are developing APIs that others might write code against, it would be useful for them to have access to the contracts for your code.

This column is based on a prerelease version of the Microsoft .NET Framework 4. Details are subject to change.

Send your questions and comments to clrinout@microsoft.com.

To that end is the fourth part, the tool ccrefgen.exe, which will create separate contract reference assemblies that contain only the contracts. The rewriter and static checker will then make use of any contract assemblies when they are doing their instrumentation and analysis.

To get more information about all of these tools or to get the latest releases, please check the Code Contracts site on DevLabs: msdn.microsoft.com/en-us/devlabs/dd491992.aspx.

The Code Contract Library

There are three basic types of code contracts: preconditions, postconditions, and object invariants. Preconditions express what program state is required for the method to run successfully. Postconditions tell you what you can rely upon at the completion of the method. Object invariants are guarantees about conditions that will always be true for an object. They can be also thought of as postconditions that apply to every (public) method. Each of these three types has several flavors, and there are a few other types of contracts that we will eventually get into in some detail. If you want all of the nitty-gritty details about the library, please look in the documentation.

On the BCL team, most contracts are included in debug builds to provide more information for finding bugs.

There are a few things that are common to all types of code contracts. First, since code contracts are primarily to help find bugs in code, they are conditionally compiled upon the symbol CONTRACTS_FULL. This way, the developer can choose whether to include the checks as needed. On the BCL team, most contracts are included in debug builds to provide more information for finding bugs, but are not included in retail builds. Second, all conditions that are checked by contracts must be side-effect free. Third, contracts are inherited. This is because you often have an API that expects type T, but might receive a subclass of T instead. The programmer expects T's guarantees to hold, and contract inheritance ensures this.

Preconditions

There are three basic forms of preconditions, two of which take the form of different Requires methods on the Contract class. Both of these also have overloads that allow you to include a message to display if the contract is violated. Here is an example of using Requires statements to encode preconditions:

```
public Boolean TryAddItemToBill(Item item)
{
    Contract.Requires<NullReferenceException>(item != null);
    Contract.Requires(item.Price >= 0);
    …
```

The Requires method is simply a way to encode that a particular condition must be true upon entry to the method. It can only use data that is at least as visible as the method itself, so that callers might actually be able to satisfy the condition. The other form, Requires<TException>, makes that same statement, but further guarantees that if the condition is not met, an exception of type TException should be thrown. It is also unique in that it is always compiled, so use of this method entails a hard dependency on the tools. You should decide if you want that before using this method.

The last form of precondition is something developers have been using since the beginning of the .NET Framework. It is the if-then-throw form used for parameter validation. For example:

```
public Boolean ExampleMethod(String parameter)
{
    if (parameter == null)
        throw new ArgumentNullException("parameter must be non-null");
}
```

The benefit of this type of precondition is that it is always there to perform the runtime check. But there are several things that the code contract system provides that are not present with this form of validation: these exceptions can be swallowed by catch statements; they aren't inherited; and it is difficult for tools to recognize them. For that reason, there exists the Contract.EndContractBlock method. This method is a no-op at runtime, but indicates to the tools that all preceding if-then-throw statements ought to be treated as contracts. So, to let the tools know about these contracts, we could modify the above example as follows:

```
public Boolean ExampleMethod(String parameter)
{
    if (parameter == null)
        throw new ArgumentNullException("parameter must be non-null");
    // tells tools the if-check is a contract
    Contract.EndContractBlock();
```

Note that if-then-throw statements may appear in many places in your code, such as for validating user input, but the only place one counts as a contract is when it is at the beginning of your method and is followed by a call to EndContractBlock or one of the Requires or Ensures methods.

There are three different ways to encode preconditions, but which one should you use? That might vary from class to class or assembly to assembly, but there are some general guidelines you should follow. If you don't want to do any argument validation in your release code, use Requires. That way you enable contract checking only for your debug builds.

If you do want argument validation in your released code, there are several things to consider. One factor is whether you are writing brand new code or updating existing code. In the BCL, we use the if-then-throw contracts to match our existing patterns. This does mean that we need to do any inheritance manually, since we do not run the tools on our final builds. If you are writing new code, you can decide whether you want to use the old form or switch to the new form and get the other benefits of contracts. Part of that decision should be determining whether you are willing to take a dependency on the binary rewriter as part of your build process. The CLR team chose not to, as the tool is currently under active development. So, we use the if-then-throw form for anything we want to make sure is in the retail build, but we can use the Requires form for extra checks to help with debugging.

Postconditions

There are two basic types of postconditions: guarantees about normal returns and guarantees about exceptional returns. For this, there are two different methods on the Contract class. Again, each has an overload that will allow the developer to pass in a message for when the contract is violated. To continue with the example from the preconditions, here are some postconditions on that same method:

```
public Boolean TryAddItemToBill(Item item)
{
    Contract.Ensures(TotalCost >= Contract.
      OldValue(TotalCost));
    Contract.Ensures(ItemsOnBill.Contains(item) ||
      (Contract.Result<Boolean>() == false));
    Contract.EnsuresOnThrow<IOException>(TotalCost ==
      Contract.OldValue(TotalCost))
    …
```

The Ensures method simply makes a statement about a condition that is guaranteed to be true at normal return from a method. In general, these methods are not intended to be included in your retail builds, but are only for debugging purposes. Their use is encouraged wherever they make sense. EnsuresOnThrow<TException> makes a guarantee for the case where a particular exception is thrown. The ability to make statements about exceptional conditions is another benefit over a simple Debug.Assert. Note that this should only be used for exceptions that you expect to throw from your method

To express more useful postconditions, it helps to provide information about values at various points in the method.

and should be as specific as possible. Using the type Exception for TException is not recommended, as then you are making guarantees about program state after events you do not control, such as an OutOfMemoryException or StackOverflowException.

You may have noticed some extra Contract methods in that example. In order to express more useful postconditions, it helps to have a way to express information about values at various points in the method. For example, say you have a method that ensures that the value of the instance at the end of the method is the same as the value when the method was called, and you want to be able to check that guarantee with contracts. The Contract class provides several methods that can only be used in postconditions to help out with that:

```
public static T Result<T>();
public static T OldValue<T>(T value);
public static T ValueAtReturn<T>(out T value);
```

The Result<T> method is used to represent the return value of the method. OldValue<T> is used to represent the value of a variable at the beginning of the method. Each Ensures method is evaluated at any exit from a method, so all of the variables used refer to the value at the end of the method and no special syntax is needed. However, the Ensures methods are declared at the beginning of the method. So out parameters would not have been assigned to yet. Most compilers will complain about this, so the ValueAtReturn<T> method exists to allow you to use out parameters in postcondition contracts.

So if you wanted to implement the aforementioned example, you could write the following:

```
public class ExampleClass
{
    public Int32 myValue;
    public Int32 Sum(Int32 value)
    {
        Contract.Ensures(Contract.OldValue(this.myValue) == this.myValue);
        myValue += value; //this violates the contact and will be caught
        return myValue;
    }
}
```

Notice the error in the method above. It claims that myValue will be the same at the end of the method as it was at the beginning, but a line in the code violates that. When you enable contract checking, this bug will be detected and the developer can fix it.

One thing to keep in mind when writing postconditions is that they can be very difficult to get correct after the fact. We added some postconditions to the BCL that we thought were fairly straight-forward and obvious. But when we tested them, we found several of them that were violated in subclasses or in corner cases that we hadn't thought about. And we could not break the existing code to follow the new, cleaner postconditions, so we had to modify or remove our annotations. It helps to decide on the guarantees you want to make before you implement the class, so that you can catch any violations and fix them while you are writing them.

Object Invariants

The third major type of code contract is the object invariant. These are object-wide contracts about a condition that is guaranteed to always hold. They can be thought of as postconditions on every single public member of the object. Object invariants are encoded with the Invariant method on the Contract class:

```
public static void Invariant(bool condition);
public static void Invariant(bool condition, String userMessage);
```

They are declared in a single method on the class that contains only calls to Invariant, and it must be marked with the Contract-InvariantMethod attribute. It is common practice to name that method "ObjectInvariant" and to make it protected so that users cannot accidentally call this method. For example, an object invariant for the same object that contained the TryAddItemToBill method might be the following:

```
[ContractInvariantMethod]
protected void ObjectInvariant()
{
    Contract.Invariant(TotalCost >= 0);
}
```

Again, it is useful to decide upon the object invariants before implementing the class. That way you can try to avoid violating them, and thus avoid writing bugs in the first place.

Other Contracts

The remaining contracts are very similar to Debug.Assert in that they make a guarantee only about a particular point in the code. In fact, if you are using code contracts, the following two methods can be used in place of Debug.Assert:

```
public static void Assert(bool condition);
public static void Assert(bool condition, String userMessage);
public static void Assume(bool condition);
public static void Assume(bool condition, String userMessage);
```

These methods are conditionally compiled on both the CONTRACTS_FULL and DEBUG symbols, so that they can be used anywhere Debug.Assert would be. They are useful mainly for implementation details, such as placing requirements on internal data. At runtime, these two methods have the same behavior. The difference comes during static analysis. The static checker will attempt to prove any Assert, but it will treat the Assume statements as definitely true and add them to its collection of facts.

Debugging with Code Contracts

After you have taken the time to add contracts to your code, how can you take advantage of them to find bugs? One scenario is to run the static analysis tool and investigate any contracts it cannot prove. The other is to enable runtime checking. To get the most out of runtime checking, it helps to know what happens when a contract is violated or evaluates to false. There are two stages for this: notification and reaction.

When a failure is detected, the contract raises an event with the following EventArgs:

```
public sealed class ContractFailedEventArgs : EventArgs
{
    public String Message { get; }
    public String Condition { get; }
    public ContractFailureKind FailureKind { get; }
    public Exception OriginalException { get; }
    public Boolean Handled { get; }
    public Boolean Unwind { get; }
    public void SetHandled();
    public void SetUnwind();
    public ContractFailedEventArgs(ContractFailureKind failureKind,
        String message, String condition,
        Exception originalException);
}
```

Remember that this is still a prerelease version of the class, so things could fluctuate a bit before the final release.

There is no default handler for this event, so the recommended practice is to register one with your desired behavior, if you want behavior other than the default. You might treat this as simply a logging mechanism, and record the information according to your general practices. You can also choose to handle the failure with anything from tearing down the process to ignoring it and continuing. If you choose to do the latter, you should call SetHandled so that the next step of the failure will not take place. You might also just want to break into the debugger. When running the handlers, all exceptions are swallowed. But if you really want to unwind the stack, you can call SetUnwind. Then, after all of the handlers have been called, an exception will be thrown.

When adding contracts to the BCL, we quickly realized that registering a handler should be one of the first things you do in your code, either in your main method or as you start an AppDomain. Object invariants are checked after any constructor, so you might end up with contract failures before you are ready to handle them if you do not register your handler right away.

If no handler sets Handled or Unwind, the default behavior is an assertion. The exception to that is if the application is hosted, and then escalation policy is triggered so that the host can decide upon appropriate behavior. Skipping the handler, and letting the assertion happen, may be the most reasonable thing to do as you are developing. The dialog gives you the option to break into the debugger and find your problem, so you can fix it. Recall that contract violations are never an expected outcome, so they should always be fixed.

However, if you are testing code using a testing framework, assertions are likely not what you want. In that case, you want to register a handler that will report contract failures as test failures in your framework. Here is one example of how to do this with Visual Studio's unit test framework:

```
[AssemblyInitialize]
public static void AssemblyInitialize(TestContext testContext)
{
    Contract.ContractFailed += (sender, eventArgs) =>
    {
        eventArgs.SetHandled();
        eventArgs.SetUnwind(); // cause code to abort after event
        Assert.Fail(eventArgs.Message); // report as test failure
    };
}
```

Where to Get More Information

This article is mostly an overview of code contracts, as well as coverage of some best practices the BCL team developed as it started using contracts. To get more details on the class and find out what more you can do with code contracts, you should check out the MSDN documentation. As of the writing of this article, the documentation for code contracts in the first beta release of the Microsoft .NET Framework 4 can be found here: msdn.microsoft.com/en-us/library/system.diagnostics.contracts(VS.100).aspx. There are also recordings of two talks from the 2008 Microsoft Professional Developers Conference that give some examples and demos of code contracts: channel9.msdn.com/pdc2008/TL51/ on some tools from Microsoft Research, and channel9.msdn.com/pdc2008/PC49/ on new features in the CLR.

To get the tools and more information about their use, check out the Code Contracts site on DevLabs: msdn.microsoft.com/en-us/devlabs/dd491992.aspx. The site contains a forum, documentation, an FAQ, and downloads for the tools. ■

MELITTA ANDERSEN *is a Program Manager on the Base Class Libraries team of the CLR. She mainly works on base types, numerics, collections, globalization, and code contracts.*

Data Performance and Fault Strategies in Silverlight 3

Silverlight applications often rely on Web services for their data. The performance of data retrieval and the ability to retrieve meaningful information about exceptions that may occur in Web services are two critical areas that have been improved in Silverlight 3.

Poor performance can be an application killer. Good strategies for retrieving data from a Web service can help, but sometimes it is necessary to retrieve an object graph that can be huge and take a long time to pass from a Web service to a client. Silverlight 3 offers a new feature that passes data from a Web service using binary encoding, and this can dramatically improve performance when passing large object graphs.

A lot can go wrong when passing data between services and Silverlight applications. That is why it is important to have a good strategy for handling exceptions that may occur when calling a Web service. Silverlight 3 offers some networking enhancements that give developers more options to pass information about managed exceptions from Web services.

In this month's column, I will demonstrate how binary encoding works, the effect it has on an application's performance, and how it behaves by demonstrating it in action. I will also walk through several techniques that can be used to pass exception information using undeclared and declared faults from Windows Communication Foundation (WCF) Web services to Silverlight. I will start by demonstrating what happens when an exception occurs and how to add some quick changes to the configuration to show information while debugging. Then, I will show you how to set up a strategic fault pattern to handle the passing exception information over SOAP services, using declared faults. All code is based on the Silverlight 3 beta and accompanies this article online.

Built for Speed

SOAP and XML passed as text severely bloats the message being passed between WCF and Silverlight. This can have a negative effect on performance, in both processing the data and the time it takes to pass the data over HTTP. Silverlight 3 introduces the ability to use a binary message encoder with WCF services that communicate with Silverlight 3 client applications. The binary message encoder can improve the performance of WCF services, especially when passing large objects graphs. The biggest gains in performance using binary message encoding are realized when passing arrays, numbers, and object graphs; lesser gains are found with very small messages and strings.

This is not a compression strategy, so there is no negative effect on performance for packing and unpacking compressed data. However, the binary encoding usually does reduce the size of the data being passed. Size reduction is not guaranteed, but is readily apparent when using large object graphs and integer data. The key improvement gained from binary encoding is that it is optimized to increase server throughput.

Configuring Binary Encoding

WCF services can communicate with Silverlight 2 applications using basicHttpBinding, which sends data as text over HTTP. When using the Silverlight-enabled WCF Service file template—which is installed when you install the Silverlight tools for Visual Studio—to create a WCF service for Silverlight 2, the binding was configured to use basicHttpBinding. This file template has been changed in Silverlight 3 to configure the WCF service to use the binary message encoder instead of text.

The Silverlight-enabled WCF Service file template configured a WCF service to use binary-encoded messaging. If you use an existing WCF service, it can be configured to use binary message encoding by creating a custom binding in the bindings section of the Web.config file. The following code sample shows the custom binding, named silverlightCustomBinding, as it appears in the <system.serviceModel> section of a configuration file. The silverlightCustomBinding, configured to use binaryMessageEncoding, is then referenced by its name in the service's endpoint configuration.

```
<endpoint address="" binding="silverlightCustomBinding"
  contract="MyTestService" />
<bindings>
  <customBinding>
    <binding name="silverlightBinaryBinding">
      <binaryMessageEncoding />
      <httpTransport />
    </binding>
  </customBinding>
</bindings>
```

Since basicHttpBinding sends messages as text over HTTP, it is easy to debug the messages through a tool such as Fiddler. While basicHttpBinding can still be configured, the advantages of the

This article is based on prerelease versions of Silverlight 3.

Send your questions and comments for John to mmdata@microsoft.com.

Code download available at msdn.microsoft.com/mag200908DataPoints.

binary encoder can be so great that it is the recommended approach. It is easy to toggle back and forth between binary and text, simply by changing the config file. This is convenient when debugging a WCF service. Binary encoding is only supported by WCF services and clients. If you need a non-WCF client application to consume your WCF service, it is best not to use binary encoding.

Binary Data Versus Text Data

The first demonstration will show the differences, in both configuration and performance, between using basicHttpBinding and a binary message encoding between WCF and Silverlight 3. The sample application included with this article breaks both examples (text and binary) out into separate services that can be called from the same Silverlight 3 client.

The configuration for these services in the Web.config file of the sample application is shown in **Figure 1**. The differences between the text and binary encoding configurations are in bold. Notice that the service SpeedService0 uses the basicHttpBinding, while SpeedService1 uses a customBinding with the binding configuration named silverlightBinaryBinding (shown in the previous code sample.)

The services SpeedService0 and SpeedService1 both retrieve all products and each product's category, supplier, and order details. The query (shown in **Figure 2**) uses the Entity Framework to retrieve the object graph.

One of the best aspects of the binary message encoding is that the only changes are found in the configuration file. No changes need be made to the code.

When the sample application is run and the Text encoding option is selected (as shown in **Figure 3**), the service that uses basicHttpBinding is executed. The object graph is returned and using an HTTP monitoring tool such as Fiddler or FireBug, the results show that the object graph in text form was 4MB in size and took 850ms to retrieve. When choosing the Binary encoding option, the object graph returned is 3MB and took 600ms to retrieve. While this is a small sample using a moderately sized object graph from the Northwind database, the results are in line with the Silverlight Web Service team's benchmarks (blogs.msdn.com/silverlightws/archive/2009/06/07/improving-the-performance-of-Web-services-in-sl3-beta.aspx).

Figure 1 **Configuring Text vs. Binary**

```
<services>
  <service
behaviorConfiguration="SilverlightFaultData.Web.SpeedServiceBehavior"
    name="SilverlightFaultData.Web.SpeedService0">
    <endpoint address="" binding="basicHttpBinding"
      contract="SilverlightFaultData.Web.SpeedService0" />
    <endpoint address="mex" binding="mexHttpBinding"
      contract="IMetadataExchange" />
  </service>
  <service
behaviorConfiguration="SilverlightFaultData.Web.SpeedServiceBehavior"
    name="SilverlightFaultData.Web.SpeedService1">
    <endpoint address="" binding="customBinding"
      bindingConfiguration="silverlightBinaryBinding"
      contract="SilverlightFaultData.Web.SpeedService1" />
    <endpoint address="mex" binding="mexHttpBinding"
      contract="IMetadataExchange" />
  </service>
</services>
```

Figure 2 **Retrieving an Object Graph**

```
[OperationContract]
public IList<Products> DoWork()
{
    var ctx = new NorthwindEFEntities();
    var query = from p in ctx.Products
                .Include("Categories")
                .Include("Suppliers")
                .Include("OrderDetails")
                select p;
    var productList = query.ToList<Products>();
    return productList;
}
```

In this sample using the binary encoding, the object graph contains about 2,300 total objects and is reduced in size by 25% and is 30% faster than the text encoding.

Error Messages Are Data

When .NET managed exceptions are thrown in a Web service, they cannot be converted to a SOAP message and passed back to a Silverlight 2 client application. Also, Silverlight 2 cannot read SOAP faults. These two issues make debugging Web services difficult with Silverlight 2. Because SOAP Faults cannot be used with Silverlight 2, a common error message that most Silverlight 2 developers eventually run into when accessing a Web service is the infamous "The remote server returned an error: NotFound," which contains no pratical information. The original exception and its details are not transported to the Silverlight 2 client, which makes debugging the Web services difficult. Error messages contain data that is often critical in determining how the client application should respond. For example, **Figure 4** shows the results of calling a Web service where an exception is thrown because the database cannot be found.

When the exception is raised, an HTTP status code of 500 is returned to Silverlight. The browser networking stack prevents Silverlight from reading responses with a status code of 500, so any SOAP fault information contained within is unavailable to the Silverlight client application. Even if the message could be retrieved, Silverlight 2 is not capable of converting the fault back into a managed exception. Both of these issues have been addressed in Silverlight 3.

Tackling the Issues

Handling exceptions with WCF and Silverlight 3 requires tackling both of these issues. First, for the exception to be returned to the Silverlight client without the networking browser stack preventing Silverlight from reading it, the status code must be changed from 500 to something that allows Silverlight to read the response. This

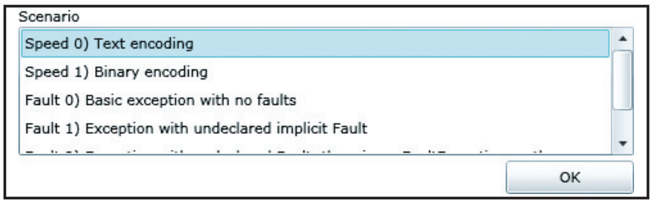
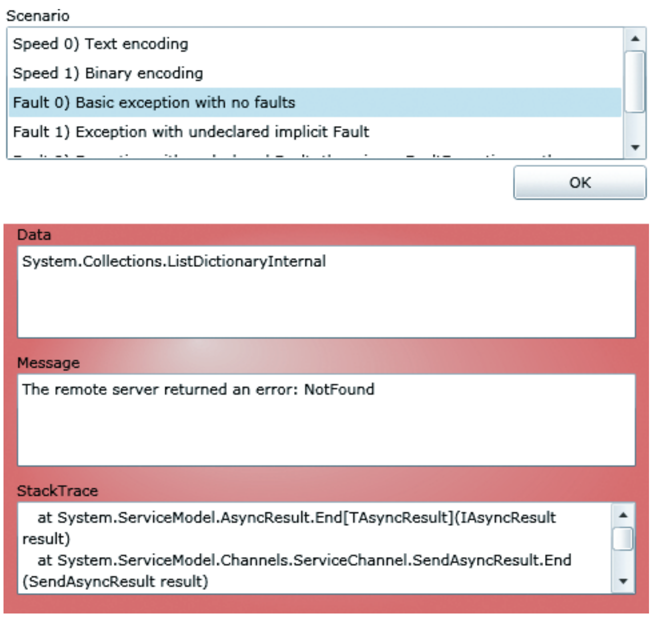

Scenario
Speed 0) Text encoding
Speed 1) Binary encoding
Fault 0) Basic exception with no faults
Fault 1) Exception with undeclared implicit Fault
OK

Figure 3 **Getting the Data via BasicHttpBinding**


Scenario

Speed 0) Text encoding

Speed 1) Binary encoding

Fault 0) Basic exception with no faults

Fault 1) Exception with undeclared implicit Fault

Data

System.Collections.ListDictionaryInternal

Message

The remote server returned an error: NotFound

StackTrace

```
   at System.ServiceModel.AsyncResult.End[TAsyncResult](IAsyncResult
result)
   at System.ServiceModel.Channels.ServiceChannel.SendAsyncResult.End
(SendAsyncResult result)
```

Figure 4 **Infamous NotFound Error**

can be achieved by deriving from the BehaviorExtensionElement and implementing IEndpointBehavior class, making it change the status code from 500 to 200 prior to whenever a fault occurs, and setting the services to use the behavior in the configuration file. The MSDN documentation contains a WCF endpoint behavior (msdn.microsoft.com/en-us/library/dd470096(VS.96).aspx) that can be used to accomplish this, and thus allow Silverlight clients access to the contents of the fault. The following code sample shows the specific code in the SilverlightFaultBehavior class that converts the status code:

```
public void BeforeSendReply(ref Message reply, object correlationState)
{
    if (reply.IsFault)
    {
        HttpResponseMessageProperty property =
          new HttpResponseMessageProperty();
        // Here the response code is changed to 200.
        property.StatusCode = System.Net.HttpStatusCode.OK;
        reply.Properties[HttpResponseMessageProperty.Name] = property;
    }
}
```

The SilverlightFaultBehavior class can be referenced in the Web.config file as a behavior extension, as shown in the following code snippet:

```
<extensions>
  <behaviorExtensions>
    <add name="silverlightFaults"
      type="SilverlightFaultBehavior.SilverlightFaultBehavior,
        SilverlightFaultBehavior, Version=1.0.0.0,
        Culture=neutral, PublicKeyToken=null" />
  </behaviorExtensions>
</extensions>
```

With the SilverlightFaultBehavior in place, the second issue is getting Silverlight to be able to convert the fault to a managed exception, so it can read it. Though this is not possible with Silverlight 2, Silverlight 3 now has the ability to process faults. This allows Silverlight 3 to read a fault and present appropriate information to the user when an exception is thrown in a Web service.

The entire process of reading the exception information in Silverlight 3 goes something like this:

1) An exception is thrown in a Web service.
2) The service uses the SilverlightFaultBehavior to convert the HTTP status code from 500 to 200.
3) The exception is converted to a SOAP fault and passed to the Silverlight 3 client.
4) The browser allows Silverlight to read the message because it has a status code of 200.
5) Code in the Silverlight 3 application checks the type of error to see if it is a FaultException or a FaultException<ExceptionDetail>.

Undeclared Faults

SOAP-based WCF services communicate errors using SOAP fault messages, or .NET managed exceptions. Therefore, the .NET managed exceptions are converted to a SOAP fault, passed to the client, and then translated back to a .NET managed exception.

Faults can be undeclared or declared, and all are strongly typed. Undeclared faults are not specified in the operation contract and should only be used for debugging. Undeclared faults return the exception message to the client exactly as it was raised in the Web service. To allow an undeclared fault, the config element's includeExceptionDetailInFaults attribute must be set to true, as shown below:

```
<serviceDebug>
<behavior name="SilverlightFaultData.Web.Service1Behavior">
  <serviceMetadata httpGetEnabled="true" />
  <serviceDebug includeExceptionDetailInFaults="true" />
</behavior>
```

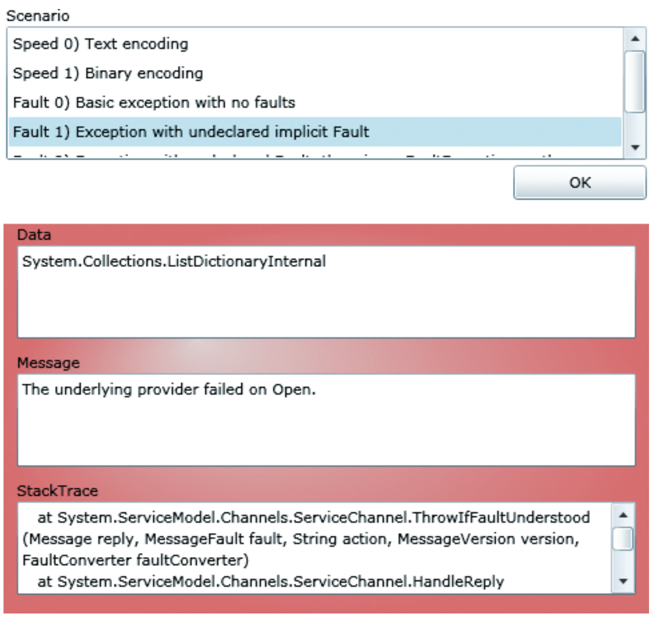
The sample application's Service1 uses this behavior, which then allows the exception to be converted automatically into a FaultException<ExceptionDetail>. The Silverlight 3 client can then check the e.Error argument in its asynchronous completion event handler and take appropriate action, as shown in the code below and in **Figure 5**:

```
if (e.Error != null) {
    ErrorPanel.DataContext = e.Error;
    if (e.Error is FaultException<ExceptionDetail>) {
        var fault = e.Error as FaultException<ExceptionDetail>;
        ErrorPanel.DataContext = fault;
    }
}
```

Undeclared faults show the exception in its raw state with all of the ugly error information, which is obviously not a good idea to show to a user. For this reason, it is not recommended to use undeclared faults in a production application. The managed exceptions can contain internal application information too (sometimes sensitive information). Setting the includeExceptionDetailInFaults to true should only be done when temporarily debugging an application error, and not in production environments. I strongly recommend that the includeExceptionDetailInFaults is set to false for production applications.

Declared Faults

A declared fault is created when the service operation is decorated with a FaultContractAttribute (or a derived type of the FaultContractAttribute). Unlike undeclared faults, declared faults are good for production as they specifically translate an exception's


Scenario

Speed 0) Text encoding

Speed 1) Binary encoding

Fault 0) Basic exception with no faults

Fault 1) Exception with undeclared implicit Fault

OK

Data

System.Collections.ListDictionaryInternal

Message

The underlying provider failed on Open.

StackTrace

```
   at System.ServiceModel.Channels.ServiceChannel.ThrowIfFaultUnderstood
(Message reply, MessageFault fault, String action, MessageVersion version,
FaultConverter faultConverter)
   at System.ServiceModel.Channels.ServiceChannel.HandleReply
```

Figure 5 **Undeclared FaultException**

information in code to the fault type. In the Web service, a developer can create the fault type in code and set its properties with information that is appropriate to send to Silverlight. The fault should only be filled with information that the client must know. Any sensitive information (such as credentials) should not be sent to the client in the fault. In the Silverlight client, a developer can write code to look for that type and tell the user something appropriate.

Figure 6 shows the service operation being decorated with the FaultContract attribute with a type of DataFault. The general F aultContract<typeof(ExceptionDetail)> could have been used, though I recommend using a specific custom fault type for the operation. In this case, the operation uses the DataFault type that I created in the sample application. This service operation will fail because the database cannot be found. An exception will be thrown and then caught by the try/catch block, where the exception is read and key information is put into the DataFault before it is thrown. At this point, the DataFault is converted to a SOAP fault and sent back to the Silverlight client with a status code of 200.

Figure 6 **Creating a Declared Fault**

```
[OperationContract]
[FaultContract(typeof(DataFault))]
public IList<Products> DoWork()
{
    try
    {
        var ctx = new NorthwindEFEntities();
        var query = from p in ctx.Products
                    select p;
        return query.ToList<Products>();
    }
    catch (Exception ex)
    {
        DataFault fault = new DataFault { Operation = Operation.Other,
          Description = ex.Message };
        throw new FaultException<DataFault>(fault, "because");
    }
}
```

Figure 7 **DataFault class**

```
public class DataFault
{
    public Operation Operation { get; set; }
    public string Description { get; set; }
}

public enum Operation
{
    Select,
    Insert,
    Update,
    Delete,
    Other
}
```

The DataFault class (shown in **Figure 7**) defines an Operation property and a Description property. The properties that the fault contains are up to the developer. The properties should represent the key information for the fault so it can be examined by the Silverlight client. The operation is set to a custom enumeration of type Operation (also shown in **Figure 7**) that will indicate the type of SQL operation that was being performed when the exception occurred. The Description should be set to a custom message and not to the exception message to avoid sending any sensitive information. (The sample application uses ex.Message just for demonstration purposes. I do not recommend passing the exception's Message directly back to the Silverlight client.) The FaultException also accepts a parameter that represents the reason for the exception. In the sample, the reason is set to "because." The reason can be used to help the client classify the cause of the exception.

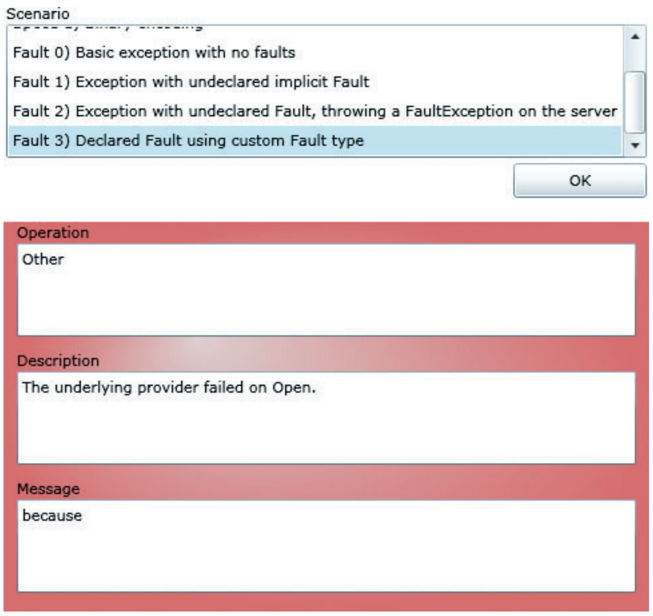
The sample's Service3 has a configuration whose endpoint indicates that the behaviorConfiguration should use the SilverlightFaultBehavior class (this translates the status code from 500 to 200). The configuration is shown here:

```
<service
  behaviorConfiguration="SilverlightFaultData.Web.Service3Behavior"
  name="SilverlightFaultData.Web.Service3">
  <endpoint address=""
    behaviorConfiguration="SilverlightFaultBehavior"
    binding="customBinding" bindingConfiguration="silverlightBinaryBinding"
    contract="SilverlightFaultData.Web.Service3" />
  <endpoint address="mex" binding="mexHttpBinding"
    contract="IMetadataExchange" />
</service>
```

When the service that uses the declared fault is executed, the Silverlight client receives and can read the fault. The following code is executed when the asynchronous Web service operation completes:

```
if (e.Error is FaultException<ServiceReference3.DataFault>)
{
    var fault = e.Error as FaultException<ServiceReference3.DataFault>;
    ErrorPanel.DataContext = fault;
}
```

The Error is checked to see if it is a FaultException of type DataFault. If it is, then its individual properties Operation and Description can be examined. **Figure 8** shows the DataFault's custom information displayed directly to the user.


Scenario

Fault 0) Basic exception with no faults

Fault 1) Exception with undeclared implicit Fault

Fault 2) Exception with undeclared Fault, throwing a FaultException on the server

Fault 3) Declared Fault using custom Fault type

OK

Operation

Other

Description

The underlying provider failed on Open.

Message

because

Figure 8 **Examining the Declared Fault**

In production applications, a custom fault strategy should be devised to map some exceptions to SOAP faults. The key here is determining the circumstances under which exceptions should be mapped to faults. This depends on whether the client application should be informed of specific information about errors on the server.

Wrapping Up

This article explained how Silverlight 3 applications can benefit from both binary encoding and exception management features. Binary message encoding is a solid choice over basicHttpBinding when using .NET WCF clients, such as Silverlight. Exceptions often contain critical information that can help in debugging an application. This article showed how to surface exception information in both development and production environments, using the Silverlight 3 enhancements. ■

JOHN PAPA *(johnpapa.net) is a senior consultant and a baseball fan who spends summer nights rooting for the Yankees with his family. John, a Silverlight MVP, Silverlight Insider, and INETA speaker, has authored several books, including his latest, titled* Data-Driven Services with Silverlight 2 *(O'Reilly, 2009). He often speaks at conferences such as VSLive!, DevConnections, and MIX.*

DINO ESPOSITO

Pros and Cons of Data Transfer Objects

Nearly every developer and architect would agree on the following, though relatively loose, definition of the business logic layer (BLL) of a software application: The BLL is the part of the software application that deals with the performance of business-related tasks. Code in the BLL operates on data that attempts to model entities in the problem domain—invoices, customers, orders, and the like. Operations in the BLL attempt to model business processes.

Under the hood of this largely accepted definition lie a number of key details that are left undefined and unspecified. Design patterns exist to help architects and code designers transform loose definitions into blueprints. In general, BLL design patterns have a slightly different focus. They model operations and data and often serve as the starting point for designing the BLL.

In this article, after a brief refresher on procedural and object-based patterns for organizing the BLL, I'll focus on one side of the problem—data transfer objects—that if not effectively addressed at the architecture level, may have a deep impact on the development of the project.

Procedural Patterns for BLL

When it comes to designing the BLL, you can start from the use-cases that have emerged during the analysis phase. Typically, you end up coding one method for each required interaction between the user and the system. Each interaction forms a logical transaction that includes all due steps—from collecting input to performing the task, and from database access to refreshing the user interface. This approach is referred to as the Transaction Script (TS) pattern.

In TS, you focus on the required actions and don't really build a conceptual model of the domain as the gravitational center of the application.

To move data around, you can use any container objects that may suit you. In the Microsoft .NET space, this mostly means using ADO.NET data containers such as DataSets and DataTables. These objects are a type of super-array object, with some search, indexing, and filtering capabilities. In addition, DataSets and DataTables can be easily serialized across tiers and even persisted locally to enable offline scenarios.

The TS pattern doesn't mandate a particular model for data representation (and doesn't prevent any either). Typically, operations are grouped as static methods in one or more entry-point classes. Alternatively, operations can be implemented as commands in a Command pattern approach. Organization of data access is left to the developer and typically results in chunks of ADO.NET code.

The TS pattern is fairly simple to set up; at the same time, it obviously doesn't scale that well, as the complexity of the application grows. In the .NET space, another pattern has gained wide acceptance over the years: the Table Module pattern. In a nutshell, the Table Module pattern suggests a database-centric vision of the BLL. It requires you to create a business component for each database table. Known as the table module class, the business component packages the data and behavior together.

In the Table Module pattern, the BLL is broken into a set of coarse-grained components, each representing an entire database table. Being strictly table-oriented, the Table Module pattern lends itself to using recordset-like data structures for passing data around. ADO.NET data containers or, better yet, customized and typed version of ADO.NET containers are the natural choice.

As the need for a more conceptual view of the problem domain arises, the BLL patterns that have worked for years in the .NET space need to evolve some more. Architects tend to build an entity/relationship model that represents the problem domain and then look at technologies like LINQ-to-SQL and Entity Framework as concrete tools to help.

Object-Based Patterns for BLL

The Table Module pattern is based on objects, but it's not really an object-based pattern for modeling the business logic. It does have objects, but they are objects representing tables, not objects representing the domain of the problem.

In an object-oriented design, the business logic identifies entities and expresses all of the allowed and required interactions between entities. In the end, the application is viewed as a set of interrelated and interoperating objects. The set of objects mapping to entities, plus some special objects performing calculations form the domain model. (In the Entity Framework, you express the domain model using the Entity Data Model [EDM].)

There are various levels of complexity in a domain model that suggest different patterns—typically the Active Record pattern or the Domain Model pattern. A good measure of this complexity is the gap between the entity model you have in mind and the relational data model you intend to create to store data. A simple

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

domain model is one in which your entities map closely to tables in the data model. A not-so-simple model requires mapping to load and save domain objects to a relational database.

The Active Record pattern is an ideal choice when you need a simple domain model; otherwise, when it is preferable to devise entities and relationships regardless of any database notion, the Domain Model pattern is the way to go.

The Active Record pattern is similar to what you get from a LINQ-to-SQL object model (and the defaultgenerated model with the Entity Designer in the Entity Framework Version 1.0). Starting from an existing database, you create objects that map a row in a database table. The object will have one property for each table column of the same type and with the same constraints. The original formulation of the Active Record pattern recommends that each object makes itself responsible for its own persistence. This means that each entity class should include methods such as Save and Load. Neither LINQ-to-SQL nor Entity Framework does this though, as both delegate persistence to an integrated O/RM infrastructure that acts as the real data access layer, as shown in **Figure 1**.

The Service Layer

In **Figure 1**, you see a logical section of the BLL named as the "service layer" sitting in between the presentation layer and the layer that takes care of persistence. In a nutshell, the service layer defines an interface for the presentation layer to trigger predefined system actions. The service layer decouples presentation and business logic and represents the façade for the presentation logic to call into the BLL. The service layer does its own job, regardless of how the business logic is organized internally.

As a .NET developer, you are quite familiar with event handlers in Web or Windows forms. The canonical Button1_Click method belongs to the presentation layer and expresses the system's behavior after the user has clicked a given button. The system's behavior—more exactly, the use case you're implementing—may require some interaction with BLL components. Typically, you need to instantiate the BLL component and then script it. The code necessary to script the component may be as simple as calling the constructor and perhaps one method. More often, though, such code is fairly rich with branches, and may need to call into multiple objects or wait for a response. Most developers refer to this code as application logic. Therefore, the service layer is the place in the BLL where you store application logic, while keeping it distinct and separate from domain logic. The domain logic is any logic you fold into the classes that represent domain entities.

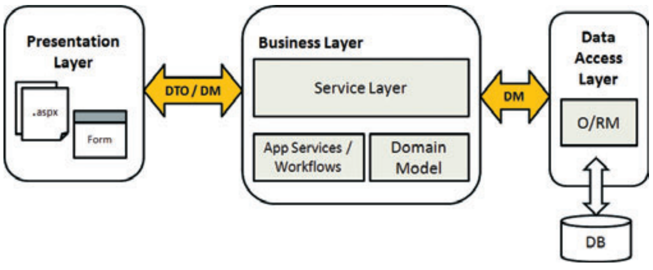


Figure 1 **A Layered Architecture – the Domain Model Pattern Used for the BLL**

In **Figure 1**, the service layer and domain model blocks are distinct pieces of the BLL, although they likely belong to different assemblies. The service layer knows the domain model and references the corresponding assembly. The service layer assembly, instead, is referenced from the presentation layer and represents the only point of contact between any presentation layer (be it Windows, Web, Silverlight, or mobile) and the BLL. **Figure 2** shows the graph of references that connect the various actors. The service layer is a sort of mediator between the presentation layer and the rest of the BLL. As such, it keeps them neatly separated but loosely coupled so that they are perfectly able to communicate. In **Figure 2**, the presentation layer doesn't hold any reference to the domain model assembly. This is a key design choice for most layered solutions.

Introducing Data Transfer Objects

When you have a domain-based vision of the application, you can't help but look seriously into data transfer objects. No multitier solution based on LINQ to SQL or Entity Framework is immune from this design issue. The question is, how would you move data to and from the presentation layer? Put another way, should the presentation layer hold a reference to the domain model assembly? (In an Entity Framework scenario, the domain model assembly is just the DLL created out of the EDMX file.)

Ideally, the design should look like **Figure 3,** where made-to-measure objects are used to pass data from the presentation layer to the service layer, and back. These ad hoc container objects take the name of Data Transfer Objects (DTOs).

A DTO is nothing more than a container class that exposes properties but no methods. A DTO is helpful whenever you

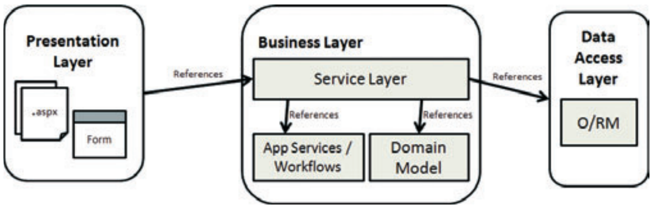


Figure 2 **Graph of References Between Participant Actors**

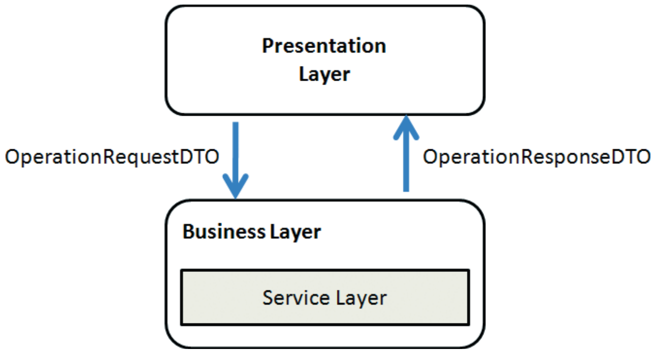


Figure 3 **Communication Between Presentation Layer and Service Layer**

- Business logic and business rules. The business logic drives the business processes, and the business rules can perform validation on the business entities.
- Business entities. These are classes that represent the data of your application.

Code that is more infrastructure-related is usually very hard to multi-target. The following are examples:

- Visual elements (views). The way that you specify visual elements, such as controls, differs enough between WPF and Silverlight to make them hard to multi-target. Not only are different controls available for each platform, but the XAML that's used to specify the layout also has different capabilities. Although it's not impossible to multi-target very simple views or some simple styling, you'll quickly run into limitations.
- Configuration settings. Silverlight does not include the System.Configuration namespace and has no support for configuration files. If you want to make your Silverlight application configurable, you'll need to build a custom solution.
- Data access. The only way a Silverlight application can have access through data is through Web services. Unlike WPF, a Silverlight application cannot directly access databases.
- Interop (with other applications, COM, or Windows Forms). A WPF application in a full-trust environment can interact with other applications on your computer or use existing assets such as COM or Windows Forms objects. This is not possible in Silverlight, because it runs in a protected sandbox.
- Logging and tracing. Because of the protected sandbox, a Silverlight application cannot write log information to the EventLog or trace information to a file (other than in isolated storage).

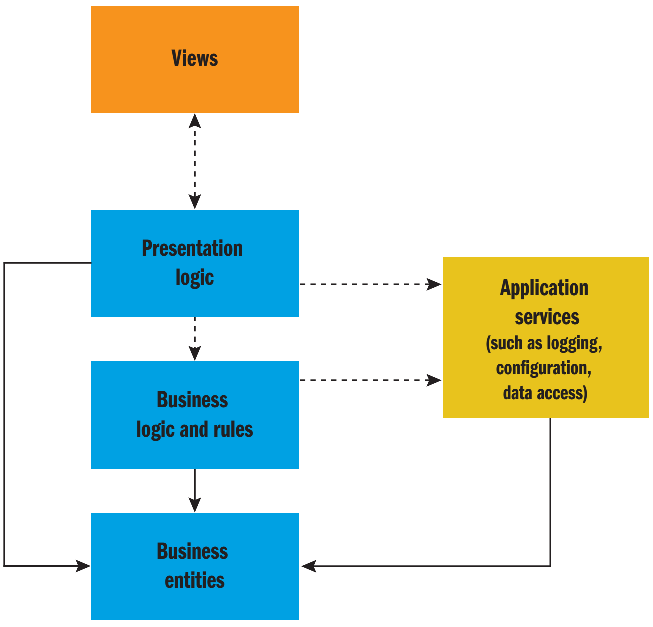

Views
Presentation logic
Business logic and rules
Business entities
Application services (such as logging, configuration, data access)

Figure 2 **Typical Prism Application Architecture**

In order to design an application that allows you to easily reuse your business logic, you should try to separate things that are easy to multi-target from things that are hard to multi-target. Interestingly enough, this is exactly the architecture of a typical Prism application. **Figure 2** shows the typical architecture of a Prism application.

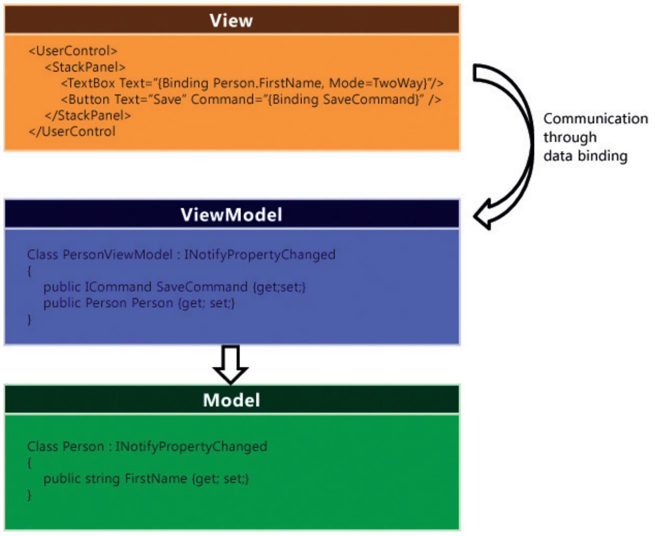
In this diagram, the views are classes that perform the visualization aspect of your application. Typically, these are controls and pages, and in the case of WPF or Silverlight applications, they often define the layout in XAML. The logic of your application is factored out into separate classes. I'll dive a bit more into the design patterns behind this when I talk about separated presentation patterns.

The application services in this diagram can provide a wide variety of functionality. For example, a Logger or a Data Access component can be considered an application service. Prism also offers a couple of these services, such as the RegionManager or the XapModuleTypeLoader. I'll discuss these services more when I talk about building platform-specific services.

Separated Presentation

As part of the guidance that we provide with Prism, we recommend that you separate the visualization aspect of your application from the presentation logic. A lot of design patterns, such as Model-View-ViewModel or Model-View-Presenter, can help you with this. What most of these patterns have in common is that they describe how to split up your user-interface-related code (and markup) into separate classes, each with distinct responsibilities. **Figure 3** shows an example of the Model-View-ViewModel pattern.

The Model class has the code to contain and access data. The View is usually a control that has code (preferably in the form of XAML markup) that visualizes some of the data in your model and ViewModel. And then there is a class named either ViewModel, PresentationModel, or Presenter that will hold as much of the UI logic as possible. Typically, a separated presentation pattern is implemented to make as much of your UI-related code unit testable as possible. Because the code in your views is notoriously hard to



```
<UserControl>
   <StackPanel>
      <TextBox Text="{Binding Person.FirstName, Mode=TwoWay}"/>
      <Button Text="Save" Command="{Binding SaveCommand}" />
   </StackPanel>
</UserControl
```

```
Class PersonViewModel : INotifyPropertyChanged
{
   public ICommand SaveCommand {get;set;}
   public Person Person {get; set;}
}
```

```
Class Person : INotifyPropertyChanged
{
   public string FirstName {get; set;}
}
```

Figure 3 **Example of Model-View-ViewModel Pattern**

unit test, these separated presentation patterns help you place as much of the code as possible in a testable ViewModel class. Ideally, you would not have any code in your views, just some XAML markup that defines the visual aspects of your application and some binding expressions to display data from your ViewModel and Model.

When it comes to multi-targeting, a separated presentation pattern has another significant advantage. It allows you to reuse all of your UI logic, because you have factored out that logic into separate classes. Although it's not impossible to multi-target some of the code in your views (the XAML, controls, and code-behind), we've found that the differences between WPF and Silverlight are big enough that multi-targeting your XAML is not practical. XAML has different abilities, and the controls that are available for WPF and Silverlight are not the same. This not only affects the XAML, but it also affects the code-behind.

Although it's not likely that you are able to reuse all of your UI-related code, a separated presentation pattern helps you reuse as much of the presentation logic as possible.

Building Platform-Specific Services

While building the Prism libraries and the Stock Trader Reference Implementation, we strictly followed the single-responsibility principle. This principle describes that each class should have only one reason to change. If a class addresses multiple concerns or has more than one responsibility, it has multiple reasons to change. For example, a class that can load a report from a database and print that report can change if the database changes or if the layout of the report changes. An interesting indication if your class does too much: if you find that you have difficulty determining a name for your class that describes its responsibility, it has too many responsibilities.

If you follow the single-responsibility principle, you'll often end up with a lot of smaller classes, each with its own discrete responsibility and a descriptive name. We often consider many of these classes to be application services, because they provide a service to your application.

This single-responsibility principle really helps when it comes to multi-targeting. Take, for example, the module loading process in Prism. A lot of aspects of this process are similar for both WPF and Silverlight. Some similarities include how the ModuleCatalog keeps track of which modules are present in the system and how the ModuleInitializer creates the module instances and calls the IModule.Initialize() method on them. But then again, how we are loading the assembly files that contain the modules differs quite a bit between WPF and Silverlight. **Figure 4** illustrates this.

It's perfectly reasonable for a WPF application to load its modules from disk. So this is what the FileModuleTypeLoader does. However, this doesn't make sense for a Silverlight application, because its protected sandbox doesn't give access to the file system. But for Silverlight, you'll need a XapModuleTypeLoader to load modules from a .xap file.

Because we created smaller classes, each with a distinct responsibility, it was a lot easier to reuse most of these classes and create only platform-specific services to encapsulate the behavior that differs between the platforms.

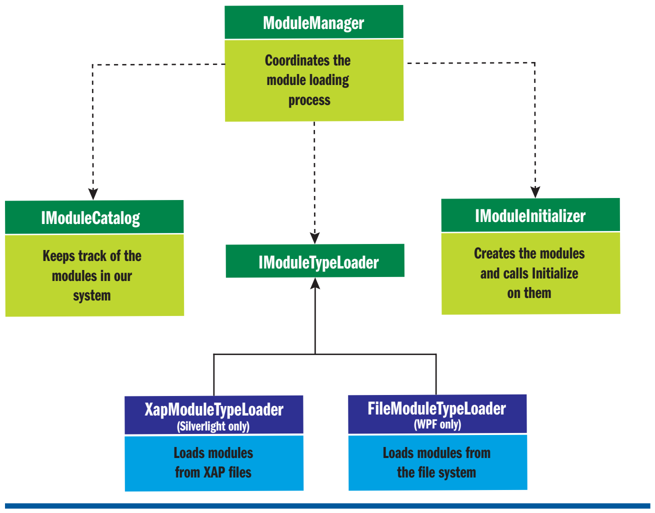

ModuleManager
Coordinates the module loading process
IModuleCatalog
Keeps track of the modules in our system
IModuleTypeLoader
IModuleInitializer
Creates the modules and calls Initialize on them
XapModuleTypeLoader (Silverlight only)
Loads modules from XAP files
FileModuleTypeLoader (WPF only)
Loads modules from the file system

Figure 4 **Module Loading in Prism**

Avoid Inconsistencies and Try to Keep a Single Code Base

Even though most functionality in Prism was easily ported to Silverlight, we inevitably ran into situations where we would rely on a feature in WPF that didn't exist in Silverlight. Dependency property inheritance was one of them. In WPF, you could set a dependency property on a control and it would automatically be inherited by any of its children. We were using this capability to associate a region with a region manager. Unfortunately, automatic property inheritance is not available in Silverlight.

For Silverlight, we had to create a solution that delayed the creation of regions until the region manager could be located through some other mechanism. With a couple of tweaks, we could reuse this code for WPF. We could have kept the original, much simpler solution for WPF and used only the new solution for Silverlight, but then we would have had to maintain two code bases and offer a different public API.

When trying to build a functionality for use in both WPF and Silverlight, you'll inevitably run into situations where one of the platforms doesn't support a feature that you want to use. Your best defense against these situations is to try to work around these "incompatibilities" and create a solution that works in both environments. Maintaining a single code base is a lot easier than maintaining two code bases!

Accommodate for Different Platform Capabilities

There are cases where it doesn't make sense or isn't possible to work around platform differences, such as when there is no common solution that would work in both WPF and Silverlight. When this happens, there are a couple of strategies to consider. For anything but small and isolated platform differences, I would recommend building platform-specific services. But for small need to group values in ad hoc structures for passing data around.

From a pure design perspective, DTOs are a solution really close to perfection. DTOs help to further decouple presentation from the service layer and the domain model. When DTOs are used, the presentation layer and the service layer share data contracts rather than classes. A data contract is essentially a neutral representation of the data that interacting components exchange. The data contract describes the data a component receives, but it is not a system-specific class, like an entity. At the end of the day, a data contract is a class, but it is more like a helper class specifically created for a particular service method.

A layer of DTOs isolates the domain model from the presentation, resulting in both loose coupling and optimized data transfer.

Other Benefits of DTOs

The adoption of data contracts adds a good deal of flexibility to the service layer and subsequently to the design of the entire application. For example, if DTOs are used, a change in the requirements that forces a move to a different amount of data doesn't have any impact on the service layer or even the domain. You modify the DTO class involved by adding a new property, but leave the overall interface of the service layer intact.

It should be noted that a change in the presentation likely means a change in one of the use cases and therefore in the application logic. Because the service layer renders the application logic, in this context a change in the service layer interface is still acceptable. However, in my experience, repeated edits to the service layer interface may lead to the wrong conclusion that changes in the domain objects—the entities—may save you further edits in the service layer. This doesn't happen in well-disciplined teams or when developers have a deep understanding of the separation of roles that exists between the domain model, the service layer, and DTOs.

As **Figure 4** shows, when DTOs are employed, you also need a DTO adapter layer to adapt one or more entity objects to a different interface as required by the use case. In doing so, you actually implement the "Adapter" pattern—one of the classic and most popular design patterns. The Adapter pattern essentially

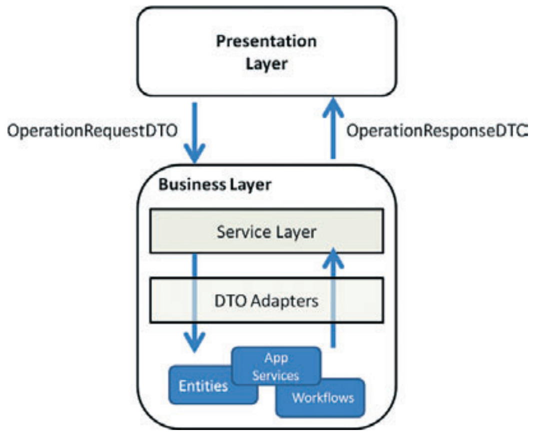


Figure 4 **DTO Adapters in the BLL**

converts the interface of one class into another interface that a client expects.

With reference to **Figure 4**, the adapter layer is responsible for reading an incoming instance of the OperationRequestDTO class and for creating and populating fresh instances of OperationResponseDTO.

When requirements change and force changes in a DTO-based service layer, all you need to do is update the public data contract of the DTO and adjust the corresponding DTO adapter.

The decoupling benefits of DTOs don't end here. In addition, to happily surviving changes in the presentation, you can enter changes to the entities in the domain model without impacting any clients you may have.

Any realistic domain model contains relationships, such as Customer-to-Orders and Order-to-Customer, that form a double link between Customer and Order entities. With DTOs, you also work around the problem of managing circular references during the serialization of entity objects. DTOs can be created to carry a flat stream of values that, if needed, serialize just fine across any boundaries. (I'll return to this point in a moment.)

Drawbacks of DTOs

From a pure design perspective, DTOs are a real benefit, but is this theoretical point confirmed by practice, too? As in many architecture open points, the answer is, it depends.

Having hundreds of entities in the domain model is definitely a good reason for considering alternatives to a pure DTO-based approach. In large projects with so many entities, DTOs add a remarkable level of (extra) complexity and work to do. In short, a pure, 100% DTO solution is often just a 100 percent painful solution.

While normally the complexity added to a solution by DTOs is measured with the cardinality of the domain model, the real number of needed DTOs can be more reliably determined looking at the use cases and the implementation of the service layer. A good formula for estimating how many DTOs you need is to look at the number of methods in the service layer. The real number can be smaller if you are able to reuse some DTOs across multiple service layer calls, or higher if your DTOs group some data using complex types.

In summary, the only argument against using DTOs is the additional work required to write and manage the number of resulting DTO classes. It is not, however, a simple matter of a programmer's laziness. In large projects, decoupling presentation from the service layer costs you hundreds of new classes.

It should also be noted that a DTO is not simply a lightweight copy of every entity you may have. Suppose that two distinct use cases require you to return a collection of orders—say, GetOrdersByCountry and GetOrdersByCustomer. Quite likely, the information to put in the "order" is different. You probably need more (or less) details in GetOrdersByCustomer than in GetOrdersByCountry. This means that distinct DTOs are necessary. For this reason, hundreds of entities are certainly a quick measure of complexity, but the real number of DTOs can be determined only by looking at use cases.

If DTOs are not always optimal, what would be a viable alternate approach?

The only alternative to using DTOs is to reference the domain model assembly from within the presentation layer. In this way though, you establish a tight coupling between layers. And tightly coupled layers may be an even worse problem.

Referencing Entities Directly

A first, not-so-obvious condition to enable the link of entities directly from the presentation layer is that it is acceptable for the presentation layer to receive data in the format of entity objects. Sometimes the presentation needs data formatted in a particular manner. A DTO adapter layer exists to just massage data as required by the client. If you don't use DTOs though, the burden of formatting data properly must be moved onto the presentation layer. In fact, the wrong place in which to format data for user interface purposes is the domain model itself.

Realistically, you can do without DTOs only if the presentation layer and the service layer are co-located in the same process. In this case, you can easily reference the entity assembly from within both layers without dealing with thorny issues such as remoting and data serialization. This consideration leads to another good question: Where should you fit the service layer?

If the client is a Web page, the service layer is preferably local to the Web server that hosts the page. In ASP.NET applications, the presentation layer is all in code-behind classes and lives side by side with the service layer in the same AppDomain. In such a scenario, every communication between the presentation layer and the service layer occurs in-process and objects can be shared with no further worries. ASP.NET applications are a good scenario where you can try a solution that doesn't use the additional layer of DTOs.

Technology-wise, you can implement the service layer via plain .NET objects or via local Windows Communication Foundation (WCF) services. If the application is successful, you can easily increase scalability by relocating the service layer to a separate application server.

If the client is a desktop application, then the service layer is typically deployed to a different tier and accessed remotely from the client. As long as both the client and remote server share the same .NET platform, you can use remoting techniques (or, better, WCF services) to implement communication and still use native entity objects on both ends. The WCF infrastructure will take care of marshaling data across tiers and pump it into copies of native entities. Also, in this case you can arrange an architecture that doesn't use DTOs. Things change significantly if the client and server platforms are incompatible. In this case, you have no chances to link the native objects and invoke them from the client; subsequently, you are in a pure service-oriented scenario and using DTOs is the only possibility.

The Middle Way

DTOs are the subject of an important design choice that affects the implementation of any communication between the presentation and the back end of the system.

If you employ DTOs, you keep the system loosely coupled and open toward a variety of clients. DTOs are the ideal choice, if you can afford it. DTOs add a significant programming overhead to any real-world system. This doesn't mean that DTOs should not be used, but they lead to a proliferation of classes that can really prefigure a maintenance nightmare in projects with a few hundred entity objects and even more use cases.

If you are at the same time a provider and consumer of the service layer, and if you have full control over the presentation, there might be benefits in referencing the entity model assembly from the presentation. In this way, all methods in the service layer are allowed to use entity classes as the data contracts of their signatures. The impact on design and coding is clearly quite softer.

Whether to use DTOs or not is not a point easy to generalize. To be effective, the final decision should always be made looking at the particulars of the project. In the end, a mixed approach is probably what you'll be doing most of the time. Personally, I tend to use entities as much as I can. This happens not because I'm against purity and clean design, but for a simpler matter of pragmatism. With an entity model that accounts for only 10 entities and a few use cases, using DTOs all the way through doesn't pose any significant problem. And you get neat design and low coupling. However, with hundreds of entities and use cases, the real number of classes to write, maintain, and test ominously approaches the order of thousands. Any possible reduction of complexity that fulfills requirements is more than welcome.

As an architect, however, you should always be on the alert to recognize signs indicating that the distance between the entity model and what the presentation expects is significant or impossible to cover. In this case, you should take the safer (and cleaner) route of DTOs.

Mixed Approach

Today's layered applications reserve a section of the BLL to the service layer. The service layer (also referred to as the application layer) contains the application logic; that is, the business rules and procedures that are specific to the application but not to the domain. A system with multiple front ends will expose a single piece of domain logic through entity classes, but then each front end will have an additional business layer specific to the use cases it supports. This is what is referred to as the service (or application) layer.

Triggered from the UI, the application logic scripts the entities and services in the business logic. In the service layer, you implement the use cases and expose each sequence of steps through a coarse-grained method for the presentation to call.

In the design of the service layer, you might want to apply a few best practices, embrace service-orientation, and share data contracts instead of entity classes. While this approach is ideal in theory, it often clashes with the real world, as it ends up adding too much overhead in projects with hundreds of entities and use cases.

It turns out that a mixed approach that uses data contracts only when using classes is not possible, is often the more acceptable solution. But as an architect, you must not make this decision lightly. Violating good design rules is allowed, as long as you know what you're doing. ■

Dino Esposito *is an architect at IDesign and co-author of* Microsoft .NET: Architecting Applications for the Enterprise *(Microsoft Press, 2008). Based in Italy, Dino is a frequent speaker at industry events worldwide. You can join his blog at weblogs.asp.net/despos.*

JEREMY MILLER

Incremental Delivery Through Continuous Design

In earlier Patterns in Practice columns, I've focused mainly on technical "patterns," but in this article I'll discuss the softer "practice" side of software design. The end goal of software projects is to deliver value to the customer, and my experience is that software design is a major factor in how successfully a team can deliver that value. Over design, under design, or just flat out wrong design impedes a project. Good design enables a team to be more successful in its efforts.

My experience is also that the best designs are a product of continuous design (also known as emergent or evolutionary design) rather than the result of an effort that tries to get the entire design right up front. In continuous design, you might start with a modicum of up-front design, but you delay committing to technical directions as long as you can. This approach lets you strive to apply lessons learned from the project to continuously improve the design, instead of becoming locked into an erroneous design developed too early in the project.

In addition, I firmly believe that the best way to create business value is through incremental delivery of working features rather than focusing first on building infrastructure. In this article, I'll explore how incremental delivery of working features enables a project team to better deliver business value, and how using continuous design can enable incremental delivery to be more efficient and help you create better software designs.

Incremental Delivery of Features

In 2002, my then-employer was experimenting with the newly minted Microsoft .Net Framework and had launched a trial project using ASP.NET 1.0. I, along with many others, eagerly watched the project, hoping for success so that we could start using this exciting new framework on projects of our own. Six months later the project was canceled. The team had certainly been busy, and by all accounts it had written a lot of code, but none of that code was suitable for production.

The experience of that project team yields some important lessons. The team first wrote a design specification that was apparently fairly complete and conformed to our organization's standards. With this document in hand, the team started the project by attempting to build the entire data access layer, then the business logic layer, and finally the user interface. When they started to code the user interface screens, the developers quickly realized that the existing data access code wasn't exactly what they needed to build the user interface, even though that code conformed to the design documents. At that point, IT management and the business partners didn't see any value being delivered by the project and threw in the towel in favor of other initiatives.

Ironically, to me the parent company was and is one of the world's leading examples of a lean or just-in-time manufacturer. Most of our competitors used "push" manufacturing, in which large quantities of parts are ordered for factory lines based on the forecasted demand over some period of time. The downfalls of push manufacturing are that you lose money any time you order more parts than you can use; you have to pay extra to store the stocks of surplus parts before you're ready to use them; and you are vulnerable to part shortages on the factory floor any time the forecasts are wrong—and forecasts are rarely accurate.

In contrast, my then-employer used "pull" manufacturing. Several times a day the factory systems scheduled the customer orders they needed to build over the next couple of hours, determined the quantities of parts they needed to complete those orders, and then ordered for immediate delivery exactly the number and type of parts needed. The advantages of pull manufacturing are that by buying only what is needed, you waste much less money on parts that can't be used; factories have far fewer on-hand part stocks to contend with, making manufacturing somewhat more efficient; you can quickly adapt to new circumstances and market forces when you aren't bound by forecasts made months ago; and forecasts and estimates are more accurate when made over the short term rather than a longer term.

So how does the pull versus push issue apply to software development? The failed project I described earlier used push design by trying to first determine all the infrastructural needs of the system and then trying to build out the data access infrastructure before writing other types of code. The team wasted a lot of effort designing, documenting, and building code that was never used in production.

Instead, what if the team had settled for quickly writing a high-level specification with minimal details, then proceeded to develop the highest-priority feature to production-ready quality, then the next highest-priority feature, and so on. In this scenario, the team would build out only infrastructure code, like data access code, that was pulled in by the requirements of the particular feature.

Send your questions and comments to mmpatt@microsoft.com.

Think about this. What is a better outcome for a project at the end of its scheduled timeline?

1. Only 50 percent of the proposed features are complete, but the most important features of the initial project proposal are ready to deploy to production.
2. Most of the coding infrastructure is complete, but no features are completely usable and nothing can be deployed to production.

In both cases the team is only roughly half done and neither outcome is truly a success compared to the initial plan and schedule. But which "failure" would you rather explain to your boss? I know my boss and our sales team would definitely prefer the first outcome based on incremental delivery.

The key advantages of incremental delivery are the following:

1. Working in order of business priority. Building incrementally by feature gives you a better chance to complete the features most important to the business. After all, why should you spend any time whatsoever designing, building, and testing a "nice to have" feature before the "must have" features are complete?
2. Risk mitigation. Frankly, the biggest risk in most projects isn't technical. The biggest risk is that you don't deliver business value or you deliver the wrong system. Also, the harsh reality is that the requirements and project analysis given to you is just as likely to be wrong as your design or code. By demonstrating working features to the business partners early in the project, you can get valuable feedback on your project's requirements. For my team, early demonstrations to our product manager and sales team have been invaluable for fine-tuning our application's usability.
3. Early delivery. Completed features can be put into production before the rest of the system to start earning some return on value.
4. Flexible delivery. Believe it or not, the business partners and your product manager sometimes change their priorities. Instead of gnashing your teeth at the injustice of it all, you can work in such a way that assumes that priorities will change. By tying infrastructure code to the features in play, you reduce the likelihood of wasted effort due to changing priorities.

Now, for the downside of incremental delivery: it's hard to do. In the lean manufacturing example, pull manufacturing worked only because the company's supply chain was ultraefficient and was able to stock factories with parts almost on demand. The same holds true for incremental delivery. You must be able to quickly design the elements of the new features and keep the quality of the code structure high enough that you don't make building future features more difficult. What you don't have time to do is spend weeks or even months at a time working strictly on architectural concerns—but those architectural concerns still exist. You need to change the way you design software systems to fit the incremental delivery model. This is where continuous design comes into the picture.

Continuous Design

Proponents of traditional development often believe that projects are most successful when the design can be completely specified up front to reduce wasted effort in coding and rework. The rise of Agile and Lean programming has challenged traditional notions of the timing of software design by introducing a process of continuous design that happens throughout the project life cycle. Continuous design purposely delays commitments to particular designs, spreads more design work over the life cycle of the project, and encourages a team to evolve a design as the project unfolds by applying lessons learned from the code.

Think of it this way. I simply won't develop the detailed design for a feature until it's time to build that feature. I could try to design it now, but that design work wouldn't provide any benefits until much later—and by the time my team gets to that feature, I'm likely to understand much more about our architecture and system and be able to come up with a better design than I could have at the beginning of the project.

Before I go any further, I'd like to say that continuous design does not imply that no design work takes place up front. I like this quote from Robert C. (Uncle Bob) Martin (www.agilealliance.org/system/article/file/833/file.pdf): "*The goal is to create a small but capable initial design, and then maintain and evolve that design over the life of the system.*"

Before you write off continuous design as risky and prone to error, let's discuss how to make continuous design succeed (in other words, I'm going to try to convince you that this isn't crazy).

The Last Responsible Moment

If not up front, when do you make design decisions? One of the most important lessons you learn through continuous design is to be cognizant of the decisions you make about your design and to consciously decide when to make those decisions. Lean programming teaches us to make decisions at the "last responsible moment." According to Mary Poppendieck (in her book *Lean Software Development*), following this principle means to "*delay commitment until … the moment at which failing to make a decision eliminates an important alternative.*"

The point is to make decisions as late as possible because that's when you have the most information with which to make the decision. Think back to the failed project I described at the beginning of this article. That team developed and committed to a detailed design for the data access code far too early. If the developers had let the user interface and business logic needs drive the shape of the data access code as they built the user interface features, they could have prevented quite a bit of wasted effort. (This is an example of "client-driven design," where you build out the consumer of an API first in order to define the shape and signature of the API itself.)

One of the key ideas here is that you should think ahead and continuously propose design changes, but you shouldn't commit irrevocably to a design direction until you have to. We don't want to act based on speculative design. Committing early to a design precludes the possibility of using a simpler or better alternative that might present itself later in the project. To quote a former colleague, Mike Two of NUnit 2 fame, "Think ahead yes, do ahead no."

Reversibility

Martin Fowler says, "If you can easily change your decisions, this means it's less important to get them right—which makes your life much simpler." Closely related to the last responsible moment is the concept of reversibility, which I would describe as the ability or inability to change a decision. Being cognizant of the inherent reversibility of your decisions is essential to following the principle of the last responsible moment. The first decision my team made for a recent project was whether to develop with Ruby on Rails or stay with a .NET architecture. Choosing a platform and programming language is not an easily reversible decision, and we knew we needed to make that decision early. On other projects, I've had to coordinate with external groups that needed to define and schedule their time months in advance. In cases like those, my team absolutely had to make decisions up front to engage with the external teams.

A classic decision involving reversibility is whether to build caching in an application. Think about cases where you don't know for sure if you really need to cache some piece of data. If you're afraid that caching will be impossible to retrofit later, you invariably have to build that caching at the start—even though that may be a waste of time. On the other hand, what if you've structured the code to isolate the access to this data in such a way that you could easily retrofit caching into the existing code with little risk? In the second case, you can responsibly forgo the caching support for the moment and deliver the functionality faster.

Reversibility also guides my team in what technologies and techniques we use. Because we use an incremental delivery process (Kanban with Extreme Programming engineering practices), we definitely favor technologies and practices that promote higher reversibility. Our system will probably have to support multiple database engines at some point in the future. To that end, we use an Object Relational Mapping framework to largely decouple our middle tier from the actual database engine. Just as important, we've got a fairly comprehensive set of automated tests that exercise our database access. When it's time to swap database engines, we can use those tests to be confident that our system works with the new database engine—or at least point out exactly where we're incompatible.

Predictive versus Reactive Design

First of all, what is software design? For many people, software design means "creating a design specification before coding starts" or the "Planning/Elaboration Phase." I'd like to step away from formal processes and intermediate documentation and define software design more generally as "the act of determining how the code should be structured." That being said, we can now think of software design happening in two different modes: predictive or reactive (or reflective if you prefer).

Predictive design is the design work you do before coding. Predictive design is creating UML or CRC models, performing design sessions with the development team at the beginning of iteration, and writing design specifications. Reactive design is the adjustments you make based on feedback during or after coding. Refactoring is reactive design. Every team and even individuals within a team have different preferences in using predictive or reactive design. Continuous design simply puts more importance on reactive design than does traditional software development processes.

The Importance of Feedback

Many projects are truly straightforward, with well-understood requirements, and strictly use well-known technologies. Up-front design might work fairly well with these projects, but my experience is the opposite. Almost every project I've worked on has had some degree of novelty, either in the technology used, the development techniques employed, or in the requirements. In those cases, I believe that the best way to be successful is to adopt an attitude of humility and doubt. You should never assume that what you're doing and thinking works until you have some sort of feedback that verifies the code or design.

Because continuous design involves the evolution of the code structure, it's even more important when using that approach to create rapid feedback cycles to detect early errors caused by changes to the code. Let's take the Extreme Programming (XP) model of development as an example. XP calls for a highly iterative approach to development that remains controversial. Almost as controversial is the fact that XP specifies a series of practices that are somewhat difficult to accept for many developers and shops. Specifically, XP practices are largely meant to compensate for the rapid rate of iteration by providing rapid and comprehensive feedback cycles.

- Collective ownership through pair programming. Love it or hate it, pair programming requires that at least two pairs of eyes review each and every line of production code. Pair programming provides feedback from a design or code review mere seconds after the code is written
- Test-driven development (TDD), behavior-driven development (BDD), and acceptance tests. All these activities create very rapid feedback. TDD and BDD help drive out defects in the code when initially written, but just as important, the high level of unit-test coverage makes later design changes and additions to the code much safer by detecting regression failures in a fine-grained way.
- Continuous integration. When combined with a high level of automated test coverage and possibly static code analysis tools, continuous integration can quickly find problems in the code base each and every time code is checked in.
- Retrospectives. This requires that the development team stop and discuss how the software design is helping or hurting the development effort. I've seen numerous design improvements come out of iteration and release retrospectives.

The quality and quantity of your feedback mechanisms greatly affect how you do design. For example, high automated test coverage with well-written unit tests makes refactoring much easier and more effective. Refactoring with low or no automated test coverage is probably too risky. Poorly written unit tests can be almost as unhelpful as having no tests whatsoever.

The reversibility of your code is greatly enhanced by solid feedback mechanisms.

YAGNI and the Simplest Thing that Could Possibly Work

To do continuous design, we have to make our code easy to change, but we'd really like to prevent a lot of rework in our code as we're making changes to it. To do incremental delivery, we want to focus on building only the features we're tasked with building right now, but we don't want to make the next feature impossible or harder to develop by making the design incompatible with future needs.

Extreme programming introduced two sayings to the development vernacular that are relevant here: "You aren't gonna need it" (YAGNI, pronounced "yawg-nee"), and "The simplest thing that could possibly work."

First, YAGNI forbids you to add any code to the system now that will not be used by current features. "Analysis paralysis" in software development is a very real problem, and YAGNI cuts through this problem by forcing you to focus on only the immediate problem. Dealing with complexity is hard, but YAGNI helps by reducing the scope of the system design you need to consider at any one time.

Of course, YAGNI can sound scary and maybe even irresponsible because you might very well need the level of complexity you bypassed the first time around. Following YAGNI shouldn't mean that you eliminate future possibilities. One of the best ways to ensure that is to employ "the simplest thing that could possibly work."

I like Alan Shalloway's definition of the simplest thing that could possibly work shown in the following list. (The once-and-only-once rule refers to the elimination of duplication from the code; it's another way of describing the "don't repeat yourself" principle). You should choose the simplest solution that still conforms to these rules:

1. Runs all the tests.
2. Follows the once-and-only-once rule.
3. Has high cohesion.
4. Has loose coupling.

These structural qualities of code make code easier to modify later.

The point of these complementary sayings is that each piece of complexity has to earn its right to exist. Think about all the things that can happen when you choose a more complex solution over a simpler one:

1. The extra complexity is clearly warranted.
2. The extra complexity isn't necessary and represents wasted effort over a simpler approach.
3. The extra complexity makes further development harder.
4. The extra complexity turns out to be flat-out wrong and has to be changed or replaced.

The results of adding complexity include one positive outcome and three negative outcomes. In contrast, until proven otherwise, a simple solution may be adequate. More important, the simple approach will probably be much easier to build and to use with other parts of the code, and if it does have to be changed, well, it's easier to change simple code than complex code. The worst case scenario is that you have to throw away the simple code and start over, but by that time you're likely to have a much better understanding of the problem anyway.

Sometimes a more complex solution will definitely turn out to be justified and the correct choice, but more often than not, using a simpler approach is better in the end. Consistently following YAGNI and "the simplest thing" when you're in doubt is simply following the odds.

How Much Modeling Before Coding?

Let's put documentation requirements aside for the moment. Here's a classic question in software development: "How much design and modeling should I do before starting to code?" There is no definitive answer because every situation is different. The key point here is that when you're unsure how to proceed, this means you are in a learning mode. Whether you do some modeling or exploratory coding first strictly depends on which approach helps you learn faster about the problem at hand, and, of course, I have to repeat this classic quote from Bertrand Meyer: "Bubbles don't crash."

- If you're working with an unfamiliar technology or design pattern, I think that modeling isn't nearly as useful as getting your hands dirty with some exploratory coding.
- If a design idea is much easier for you to visualize in a model than in code, by all means draw some models.
- If you have no idea where to start in the code, don't just stare at the IDE window hoping for inspiration. Take out a pen and paper and write down the logical tasks and responsibilities for the task you're working on.
- Switch to coding the second that you reach a point of diminishing returns with modeling. (Remember, bubbles don't crash!) Better aligning the boxes in your diagram does not help you write better code!
- If you do jump straight into coding and begin to struggle, stop and go back to modeling.
- Remember that you can switch between coding and modeling. Many times when you're confronted with a difficult coding problem, the best thing to do is pick out the simplest tasks, code those in isolation, and use the form of that code to help you determine what the rest of the code should look like.

Another thing to keep in mind is that some forms of modeling are more lightweight than others. If UML isn't helping you with a problem, switch to CRC cards or even entity relationship diagrams.

What's Ahead

This article had no code whatsoever, but I feel strongly that these concepts apply to almost all design decisions. In a future column, I'll talk about some specific concepts and strategies for developing designs that allow you to use continuous design principles. I'll also describe in much more detail how refactoring fits into continuous design. ■

Jeremy Miller, *a Microsoft MVP for C#, is also the author of the open-source StructureMap (structuremap.sourceforge.net) tool for Dependency Injection with .NET and the forthcoming StoryTeller (storyteller.tigris.org) tool for supercharged FIT testing in .NET. Visit his blog, "The Shade Tree Developer," part of the CodeBetter site.*

Visualizing Information with .NET

Laurence Moroney

Information visualization has been around for a long time, but ask different people what it means, and you'll likely get many different answers—for example, charting, innovative animated images, or computationally intensive representations of complex data structures. Information visualization encapsulates all of these answers, and an information visualization platform is one that can support each of these scenarios.

From a scientific perspective, **information visualization** is usually used to define the study of the visual representation of large-scale collections of information that is not necessarily numeric in nature, and the use of graphical representations of this data to allow the data to be analyzed and understood. From a business perspective, **information visualization** is all about deriving value from data through graphical rendering of the data, using tools that allow end users to interact with the data to find the information that they need.

Of course, having just the capability to draw these pictures usually isn't enough for a good information visualization platform; there are also other levels of functionality that need to be addressed, such as:

- **Interactivity** Interactivity can vary from animating the movement of slices in and out of a pie chart to providing users with tools for data manipulation, such as zooming in and out of a time series.
- **Generating related metadata** Many charts have value added to them through related contextual metadata. For example, when you view a time-series chart, you might want to generate a moving average and tweak the period for this moving average or experiment with what-if scenarios. It's not feasible to expect a data source to generate all of these data views for you. Some form of data manipulation is necessary at the presentation layer.
- **Overlaying related data** A common requirement for charting is to take a look at other stimuli that might affect the data and have the visualization reflect this. Consider a time series showing a company's stock value and a feed of news stories about that particular stock. Real value can be added to the chart by showing how the news affected the value. "Good" news might make it go up, "bad"' news might make it go down. Being able to add this data to your time-series chart turns it from a simple chart into information visualization.

This article discusses:

- Data visualization
- Building data-agnostic services
- Building a visualization server

Technologies discussed:

C#, ASP.NET, XML

The key to building a visualization platform that can enable all of this is to have flexibility, so that you can render any data in any way at any time. This is a huge and generally specialized effort, but a technique that you can use to ease this effort is to use with called data agnotisticism.

Data agnosticism arises when you define an architecture for visualizing your data that isn't dependent on the data itself. For example, if you consider the example of a time-series chart that provides related metadata, it's quite easy to program an application to read the time-series data and the related metadata (such as a news feed) and to write the data on to the screen using a charting engine. However, once you've done this, your effort is good for this representation and this representation alone. The application you've written is tightly bound to the data itself.

The principle of data agnosticism allows you to pick a data source, define the data you want, and then tell the visualization engine to go and draw it however you want it to. We'll take a look at how to build a simple version of this engine in this article.

Getting Started

As with anything else, it's good to start with the data. In this section, I'll give a brief overview of a simple XML-over-HTTP service that provides time-series data provided by Yahoo Financial Services.

The Yahoo time-series service returns a CSV file containing basic time-series data with the following fields: Date, Opening Price, Closing Price, High, Low, Volume, and Adjusted Close. The API to call it is very simple:

ichart.finance.yahoo.com/table.csv

You use the following parameters:

Parameter	Value
s	Stock Ticker (for example, MSFT)
a	Start Month (0-based; 0=January, 11=December)
b	Start Day
c	Start Year
d	End Month (0-based; 0=January, 11=December)
e	End Day
f	End Year
g	Always use the letter d
ignore	Always use the value '.csv'

To get the time-series data for Microsoft (MSFT) from January 1, 2008, to January 1, 2009, you use the following URL:

http://ichart.finance.yahoo.com/table.csv?s=MSFT&a=0&b=1&c=2008&d=0&e=1&f=2009&g=d&ignore=.csv

Figure 1 shows a C# function that takes string parameters for ticker, start date, and end date and builds this URI.

Now that you have the URI for the data, you need to read it and to use it. In this case, I'll convert the CSV data to XML. A function that can do this is shown in **Figure 2**.

I put these functions into a class called HelperFunctions and added the class to an ASP.NET Web project. To this, I added an ASP.NET Web Form (ASPX) called GetPriceHistory and edited the ASPX page to remove the HTML markup so that it looks like this:

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeBehind="GetPriceHistory.aspx.cs" Inherits="PriceHistoryService.
  GetPriceHistory" %>
```

The nice thing about this approach is that you can now write code that writes directly to the response buffer and set the response type so that you can write XML over HTTP.

Because the helper functions take strings for the ticker and for the start and end dates, you can use them as parameters to the ASPX. You can then pass them to the helper functions to generate XML, which you then write out to the response buffer. In addition, the MIME type needs to be set to "text/xml" so that any reader sees it as XML and not text.

Figure 1 **A C# Function That Builds a URI to Capture Data**

```
public string BuildYahooURI(string strTicker,
  string strStartDate, string strEndDate)
{
    string strReturn = "";
    DateTime dStart = Convert.ToDateTime(strStartDate);
    DateTime dEnd = Convert.ToDateTime(strEndDate);
    string sStartDay = dStart.Day.ToString();
    string sStartMonth = (dStart.Month -1).ToString();
    string sStartYear = dStart.Year.ToString();
    string sEndDay = dEnd.Day.ToString();
    string sEndMonth = (dEnd.Month - 1).ToString();
    string sEndYear = dEnd.Year.ToString();
    StringBuilder sYahooURI =
      new StringBuilder("http://ichart.finance.yahoo.com/table.csv?s=");
    sYahooURI.Append(strTicker);
    sYahooURI.Append("&a=");
    sYahooURI.Append(sStartMonth);
    sYahooURI.Append("&b=");
    sYahooURI.Append(sStartDay);
    sYahooURI.Append("&c=");
    sYahooURI.Append(sStartYear);
    sYahooURI.Append("&d=");
    sYahooURI.Append(sEndMonth);
    sYahooURI.Append("&e=");
    sYahooURI.Append(sEndDay);
    sYahooURI.Append("&f=");
    sYahooURI.Append(sEndYear);
    sYahooURI.Append("&g=d");
    sYahooURI.Append("&ignore=.csv");
    strReturn = sYahooURI.ToString();
    return strReturn;
}
```

Figure 2 **Converting CSV Data to XML**

```
public XmlDocument getXML(string strTicker,
   string strStartDate, string strEndDate)
        {
            XmlDocument xReturn = new XmlDocument();
            DataSet result = new DataSet();
            string sYahooURI =
               BuildYahooURI(strTicker, strStartDate, strEndDate);
            WebClient wc = new WebClient();
            Stream yData = wc.OpenRead(sYahooURI);
            result = GenerateDataSet(yData);
            StringWriter stringWriter = new StringWriter();
            XmlTextWriter xmlTextwriter = new XmlTextWriter(stringWriter);
            result.WriteXml(xmlTextwriter, XmlWriteMode.IgnoreSchema);
            XmlNode xRoot = xReturn.CreateElement("root");
            xReturn.AppendChild(xRoot);
            xReturn.LoadXml(stringWriter.ToString());

            return xReturn;
        }
```

Figure 3 **Code for the Helper Functions**

```
HelperFunctions hlp = new HelperFunctions();
protected void Page_Load(object sender, EventArgs e)
{
  string strTicker, strStartDate, strEndDate;

  if(Request.Params["ticker"]!=null)
    strTicker = Request.Params["ticker"].ToString();
  else
    strTicker = "MSFT";

  if(Request.Params["startdate"]!=null)
    strStartDate = Request.Params["startdate"].ToString();
  else
    strStartDate = "1-1-2008";

  if(Request.Params["enddate"]!=null)
    strEndDate = Request.Params["enddate"].ToString();
  else
    strEndDate = "1-1-2009";

  XmlDocument xReturn = hlp.getXML(strTicker, strStartDate, strEndDate);

  Response.ContentType = "text/xml";
  Response.Write(xReturn.OuterXml);

}
```

Figure 3 shows the code to do that. Remember that HelperFunctions is the name of a class containing the functions that build the Yahoo URI and that read it and convert the CSV data to XML.

You now have a simple XML-over-HTTP service that returns time-series data. **Figure 4** shows an example of it in action.

Building a Data-Agnostic Service That Uses This Data

With server-generated visualization, a client renders an image, and all processing is done on the server. Some very smart visualization engines provide code that can post back to the server to provide interactivity by using image maps in the image that is rendered back, but this is extremely complex to generate, and the functionality can be limited. This approach is useful if you want to generate static charts that require no end-user runtime because the browser can render the common image formats. **Figure 5** shows a typical architecture for this approach.

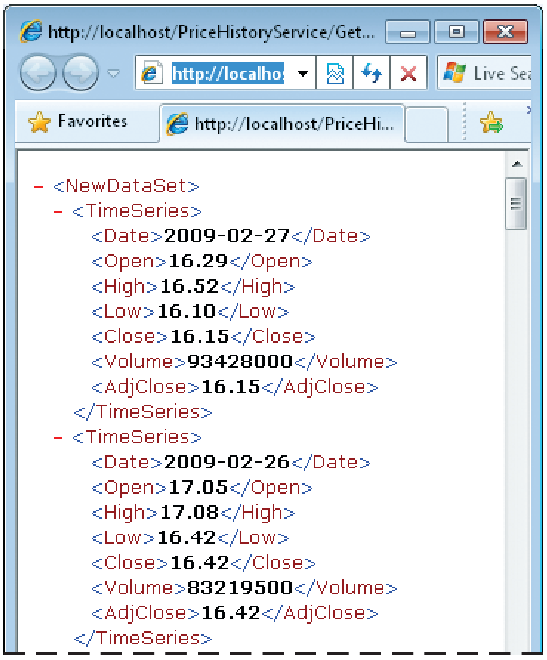
When you build this architecture, you usually write server code that understands the data. In the previous case, for example, if you're writing a time-series chart that is plotting the Close value, you would write code that reads in the XML and takes the Close data and loads it into a series on the chart so that it can be plotted.

If you are using the Microsoft ASP.NET charting engine (which is freely downloadable; see the link later in this article), you'd typically define a chart like this:

```
<asp:Chart ID="Chart1" runat="server">
    <Series>
        <asp:Series Name="Series1">
        </asp:Series>
    </Series>
    <ChartAreas>
        <asp:ChartArea Name="ChartArea1">
        </asp:ChartArea>
    </ChartAreas>
</asp:Chart>
```

This approach, however, usually limits you to **charting** rather than **visualization** because the ability to provide interactivity is limited. The ability to generate related metadata is also limited in this scenario because all requests require a post-back to the server to generate a new chart and would be limited to the functionality that is provided on the server. The ability to overlay related metadata is also limited for the same reasons.

However, the important capabilities of data agnosticism can be enabled by this scenario. It's relatively easy for you to configure metadata about your data source and where in the data source


http://localhost/PriceHistoryService/Get...
http://localho
Live Sea
Favorites
http://localhost/PriceHi...

```
- <NewDataSet>
  - <TimeSeries>
      <Date>2009-02-27</Date>
      <Open>16.29</Open>
      <High>16.52</High>
      <Low>16.10</Low>
      <Close>16.15</Close>
      <Volume>93428000</Volume>
      <AdjClose>16.15</AdjClose>
    </TimeSeries>
  - <TimeSeries>
      <Date>2009-02-26</Date>
      <Open>17.05</Open>
      <High>17.08</High>
      <Low>16.42</Low>
      <Close>16.42</Close>
      <Volume>83219500</Volume>
      <AdjClose>16.42</AdjClose>
    </TimeSeries>
```

Figure 4 **A Simple XML-over-HTTP Service**

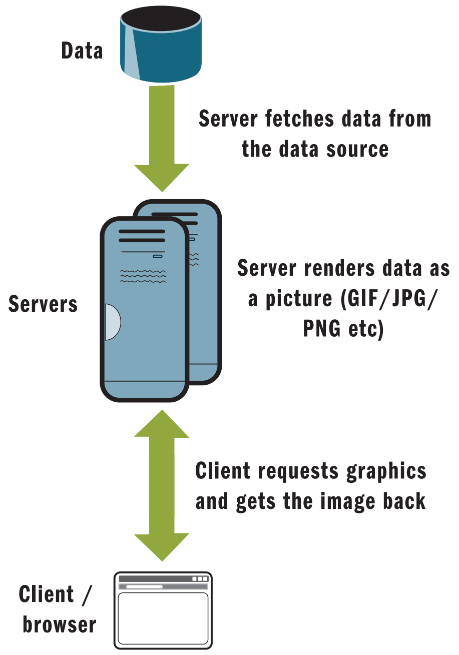

Data
Server fetches data from the data source
Servers
Server renders data as a picture (GIF/JPG/PNG etc)
Client requests graphics and gets the image back
Client / browser

Figure 5 **Typical Server-Rendered Visualization Architecture**

Figure 6 **A Configuration File That Defines a Chart**

```
<root>
  <Chart Name="PriceHistory1">
    <Uri>
      <Path>http://localhost/PriceHistoryService/GetPriceHistory.aspx</Path>
      <Param Name="ticker">MSFT</Param>
      <Param Name="startdate">1-1-2008</Param>
      <Param name="enddate">1-1-2009</Param>
    </Uri>
    <Data>
      <SeriesDefinitions>
        <Series id="ClosePrice">
          <Data>/NewDataSet/TimeSeries/Close</Data>
          <Type>Line</Type>
        </Series>
      </SeriesDefinitions>
    </Data>
  </Chart>
</root>
```

you can find your data series and data categories. An engine can process this metadata and turn it into the series and categories that the server can render, making it easy to add new visualizations without a lot of extra programming.

Building a Data-Agnostic Visualization Server

There are a number of server-side charting technologies available, and the programming APIs change across them, but the principles that I discuss here are similar across all of them. In this section, I'll look at the free ASP.NET charting engine from Microsoft, which you can download from microsoft.com/downloads/details.aspx?FamilyID=130f7986-bf49-4fe5-9ca8-910ae6ea442c&DisplayLang=en. You also need the Visual Studio add-ins for the Charting server, which you can download from microsoft.com/downloads/details.aspx?familyid=1D69CE13-E1E5-4315-825C-F14D33A303E9&displaylang=en.

Let's look at what it takes to build a pie chart with this charting engine. The code is very simple. First, add an instance of the chart control to an ASPX Web form. You'll see something like this in the code view:

```
<asp:Chart ID="Chart1" runat="server">
    <Series>
        <asp:Series Name="Series1">
        </asp:Series>
    </Series>
    <ChartAreas>
        <asp:ChartArea Name="ChartArea1">
        </asp:ChartArea>
    </ChartAreas>
</asp:Chart>
```

Then write code like the following to render some data in the chart control:

```
double[] yValues = { 20, 10, 24, 23 };
string[] xValues = { "England", "Scotland", "Ireland", "Wales" };
Series mySeries = Chart1.Series[0];
mySeries.Points.DataBindXY(xValues, yValues);
mySeries.ChartType = SeriesChartType.Pie;
```

Figure 7 **Plotting the DTime-Series Data on an ASP.NET Chart**

```
protected void Button1_Click(object sender, EventArgs e)
{
  // Variable declarations
  StringBuilder dataURI = new StringBuilder();
  WebClient webClient = new WebClient();
  XmlDocument xmlChartConfig = new XmlDocument();
  XmlDocument xmlData = new XmlDocument();
  // Get the chart config
  Uri uri = new Uri(Server.MapPath("ChartConfig.xml"),
    UriKind.RelativeOrAbsolute);
  Stream configData = webClient.OpenRead(uri);
  XmlTextReader xmlText = new XmlTextReader(configData);
  xmlChartConfig.Load(xmlText);

  // I'm hard coding to read in the chart called 'Price History 1'. In a
  // 'real' environment my config would contain multiple charts, and I'd
  // pass the desired chart (along with any parameters) in the request
  // string. But for simplicity I've kept this hard coded.
  XmlNodeList lst =
    xmlChartConfig.SelectNodes("/root/Chart[@Name='PriceHistory1']/Uri/*");

  // The first child contains the root URI
  dataURI.Append(lst.Item(0).InnerText.ToString());

  // The rest of the children of this node contain the parameters
  // the first parameter is prefixed with ?, the rest with &
  // i.e. http://url?firstparam=firstval&secondparam=secondval etc
  for (int lp = 1; lp < lst.Count; lp++)
  {
    if (lp == 1)
      dataURI.Append("?");
    else
      dataURI.Append("&");

    // In this case the desired parameters are hard coded into the XML.
    // in a 'real' server you'd likely accept them as params to this page
    dataURI.Append(lst.Item(lp).Attributes.Item(0).Value.ToString());
    dataURI.Append("=");
    dataURI.Append(lst.Item(lp).InnerText);
  }

  // Now that we have the URI, we can call it and get the XML
  uri = new Uri(dataURI.ToString());
  Stream phData = webClient.OpenRead(uri);
  xmlText = new XmlTextReader(phData);
  xmlData.Load(xmlText);

  // This simple example is hard coded for a particular chart
  // ('PriceHistory1') and assumes only 1 series
  lst = xmlChartConfig.SelectNodes(
    "/root/Chart[@Name='PriceHistory1']/Data/SeriesDefinitions/Series/Data");

  // I'm taking the first series, because I only have 1
  // A 'real' server would iterate through all the matching nodes on the
  // XPath
  string xPath = lst.Item(0).InnerText;

  // I've read the XPath that determines the data location, so I can
  // create a nodelist from that
  XmlNodeList data = xmlData.SelectNodes(xPath);
  Series series = new Series();

  // I'm hard coding for 'Line' here -- the 'real' server should
  // read the chart type from the config
  series.ChartType = SeriesChartType.Line;
  double nCurrent = 0.0;

  // I can now iterate through all the values of the node list, and
  foreach (XmlNode nd in data)
  {
    // .. create a DataPoint from them, which is added to the Series
    DataPoint d = new DataPoint(nCurrent, Convert.ToDouble(nd.
      InnerText));
    series.Points.Add(d);
    nCurrent++;
  }

  // Finally I add the series to my chart
  Chart1.Series.Add(series);
}
```

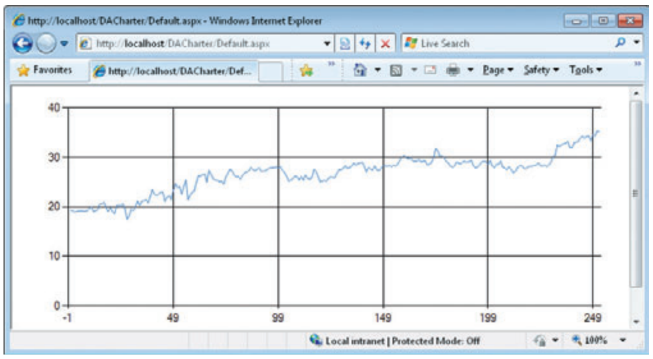


Figure 8 **The Results Generated by the Time-Series Data**

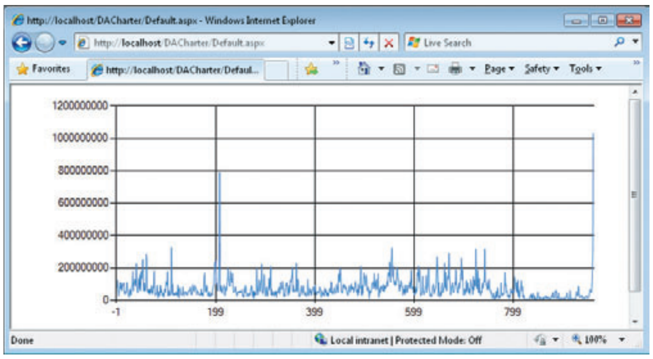


Figure 9 **New Results After Tweaking the Configuration File**

In this case, I've hard-coded the values, but you would usually read them from a database or from a service and then load them into the arrays before using them to generate the chart. Of course, the reuse of this code becomes difficult, and any changes in the data source can break it, so let's take a look at writing something that doesn't need to be bound to the data type.

The nice thing about representing the data in XML is that I can use the XPath language to define where in the XML document the data I want to plot will come from. For the data shown in **Figure 1**, the XPath statement that defines the location of the Close prices looks like this:

```
/NewDataSet/TimeSeries/Close
```

Now, if you think about it, instead of writing code that contains the definitions for your chart, you can externalize it as a configuration. Imagine a configuration file like the one shown in **Figure 6**.

You're now defining a chart called PriceHistory1 that takes its data from the given URL, appending parameters with the given names and given values. The values are hardcoded in this case, but there's nothing to stop you from writing code that uses parameters generated by an end user.

Additionally, the Series Definitions section defines a number of series with an XPath statement indicating where the data comes from and how to draw it. Right now it uses a simple definition of a chart type, but you could include extra parameters here for color or other elements or for defining multiple series (it's XML after all, so it's easy to add extra nodes) as well as categories, labels, or other such metadata. For this example I've kept it simple.

Now your charting-engine code will look vastly different. Instead of writing code that reads the data, parses the data, and loads it directly into the chart, you can write code that reads the configuration, builds the service call URI from the configuration data, calls the service, gets the returned XML, and uses the XPath variables in the configuration to get the data series you want.

Under these conditions, your architecture can be much more robust. Consider, for example, if the data source value changed its XML tag from Close to Closing Price. You wouldn't have to edit or recompile your code; you'd simply edit the XPath variable in the chart definition.

It's not much of a stretch to think about how you would edit this to connect to different types of data sources, such as database connections or Web services. **Figure 7** shows the code that plots the time-series data on an ASP.NET chart.

The results are shown in **Figure 8**. No configuration has been done on the chart, and it's using the default configuration values, but the data is being read and being plotted.

A small tweak to the configuration file to give me volume and a different set of dates (1-1-1980 to 1-1-1990) provides the view in **Figure 9**—without changing a line of code in the charting service because it is data-agnostic.

Documenting how to build a data-agnostic server would take an entire book in its own right, and here I've just skimmed the surface. The principles you've seen will apply to most APIs you work with, and the majority of the code you write should be for managing the external chart configuration to get the data, making what you've seen here very portable.

Next Time

In this article, I looked at one of the main principles of building data visualizations—providing a way to render your data in a data-agnostic manner. In a future article, I will explore using rich technologies on the client side to provide the ability to interact with your data and to smartly aggregate disparate data sources. The power of the .NET platform is now available in the browser using Microsoft Silverlight, so we will use it to demonstrate these principles. ■

LAURENCE MORONEY *is a senior technology evangelist with Microsoft, specializing in Silverlight. He is the author of many books on computing topics, including Silverlight, AJAX, interoperability, and security. You can find Laurence's blog at blogs.msdn.com/webnext.*

N-Tier Application Patterns

Danny Simmons

In my previous article, I described a foundation on which you can build successful *n*-tier applications, focusing mainly on anti-patterns to avoid. There are many issues to consider before making decisions about the design of an *n*-tier application. In this article, I examine *n*-tier patterns for success and some of the key APIs and issues specific to the Entity Framework. I also provide a sneak peak at features coming in the Microsoft .NET Framework 4 that should make *n*-tier development significantly easier.

Change Set

The idea behind the change set pattern is to create a serializable container that can keep the data needed for a unit of work together and, ideally, perform change tracking automatically on the client. The container glues together the parts of the unit of work in a custom format, so this approach also tends to be quite full-featured and is easy to use on the mid-tier and on the client.

DataSet is the most common example of this pattern, but other examples exist, such as the EntityBag sample I wrote some time ago as an exploration of this technique with the Entity Framework. Both examples exhibit some of the downsides of this pattern. First, the change set pattern places significant constraints on the client because the wire format tends to be very specific to the change set and hard to make interoperable. In practice, the client must use .NET with the same change set implementation used on the mid-tier. Second, the wire format is usually quite inefficient. Among other things, change sets are designed to handle arbitrary schemas, so overhead is required to track the instance schema. Another issue with change set implementations such as DataSet, but not necessarily endemic to the pattern, is the ease with which you can end up tightly coupling two or more of the tiers, which causes problems if you have different rates of change. Finally, and probably of most concern, is how easy it is to abuse the change set.

In some ways, this pattern automates and submerges critical concerns that should be at the forefront of your mind when designing your solution. Precisely because it is so easy to put data into the change set, send it to the mid-tier, and then persist, you can do so without verifying on the mid-tier that the changes you are persisting are only of the type that you expect. Imagine that you have a service intended to add an expense report to your accounting system that ends up also modifying someone's salary.

The change set pattern is best used in cases where you have full control over client deployment so that you can address the coupling and technology requirement issues. It is also the right choice if you want to optimize for developer efficiency rather than runtime efficiency. If you do adopt this pattern, be sure to exercise the discipline to validate any changes on the mid-tier rather than blindly persisting whatever changes arrive.

This article discusses:

- *N*-tier design patterns
- Entity Framework
- Microsoft .NET Framework 4

Technologies discussed:

Entity Framework, Windows Communication Foundation

DTOs

At the opposite end of the spectrum from change sets are Data Transfer Objects, or DTOs. The intent of this pattern is to separate the client and the mid-tier by using different types to hold the data on the mid-tier and the data on the client and in the messages sent between them.

The DTO approach requires the most effort to implement, but when implemented correctly, it can achieve the most architectural benefits. You can develop and evolve your mid-tier and your client on completely separate schedules because you can keep the data that travels between the two tiers in a stable format regardless of changes made on either end. Naturally, at times you'll need to add some functionality to both ends, but you can manage the rollout of that functionality by building versioning plus backward and forward compatibility into the code that maps the data to and from the transfer objects. Because you explicitly design the format of the data for when it transfers between the tiers, you can use an approach that interoperates nicely with clients that use technologies other than .NET. If necessary, you can use a format that is very efficient to send across the wire, or you can choose, for instance, to exchange only a subset of an entity's data for security reasons.

The downside to implementing DTOs is the extra effort required to design three different sets of types for essentially the same data and to map the information between the types. You can consider a variety of shortcuts, however, like using DTOs as the types on the client so that you have to design only two types instead of three; using LINQ to Objects to reduce the code that must be written to move data between the types; or using an automatic mapping library, which can further reduce the code for copying data by detecting patterns such as properties with the same name on more than one type. But there is no way around the fact that this pattern involves more effort than any of the other options—at least for initial implementation.

This is the pattern to consider when your solution becomes very large with very sophisticated requirements for interoperability, long-term maintenance, and the like. The longer the life of a project, the more likely that DTOs will pay off. For many projects, however, you might be able to achieve your goals with a pattern that requires less effort.

Simple Entities

Like the change set pattern, the simple entities pattern reuses the mid-tier entity types on the client, but unlike change sets, which wrap those entities in a complex data structure for communication between tiers, simple entities strives to keep the complexity of the data structure to a minimum and passes entity instances directly to service methods. The simple entities pattern allows only simple property modification to entity instances on the client. If more complex operations are required, such as changing the relationships between entities or accomplishing a combination of inserts, updates, and deletes, those operations should be represented in the structure of the service methods.

The beauty of the simple entities approach is that no extra types are required and no effort has to be put into mapping data from one type to another. If you can control deployment of the client, you can reuse the same entity structures (either the same assemblies or proxies), and even if you have to work with a client technology other than .NET, the data structures are simple and therefore easy to make interoperable. The client implementation is typically straightforward because minimal tracking is required. When properties must be modified, the client can change them directly on an entity instance. When operations involving multiple entities or relationships are required, special service methods do the work.

The primary disadvantage of this pattern is that more methods are usually required on the service if you need to accomplish complex scenarios that touch multiple entities. This leads to either chatty network traffic, where the client has to make many service calls to accomplish a scenario or special-purpose service methods with many arguments.

The simple entities approach is especially effective when you have relatively simple clients or when the scenarios are such that operations are homogenous. Consider, for example, the implementation of an e-commerce system in which the vast majority of operations involve creating new orders. You can design your application-interaction patterns so that modifications to information like customer data are performed in separate operations from creating new orders. Then the service methods you need are generally either queries for read-only data, modifications to one entity at a time without changing much in the way of relationships, or inserting a set of related entities all at once for a new order. The simple entities pattern works fairly well with this kind of scenario. When the overall complexity of a solution goes up, when your client becomes more sophisticated, or when network performance is so critical that you need to carefully tune your wire format, other patterns are more appropriate.

Self-Tracking Entities

The self-tracking entities pattern is designed to build on the simple entities pattern and achieve a good balance between the various concerns to create a single pattern that works in many scenarios. The idea is to create smart entity objects that keep track of their own changes and changes to related entities. To reduce constraints on the client, these entities are plain-old CLR objects (POCO) that are not tied to any particular persistence technology—they just represent the entities and some information about whether they are unchanged, modified, new, or marked for deletion.

Because the entities are self-tracking, they have many of the ease-of-use characteristics of a change set, but because the tracking information is built into the entities themselves and is specific to their schema, the wire format can be more efficient than with a change set. In addition, because they are POCO, they make few demands on the client and interoperate well. Finally, because validation logic can be built into the entities themselves, you can more easily remain disciplined about enforcing the intended operations for a particular service method.

There are two primary disadvantages for self-tracking entities compared to change sets. First, a change set can be implemented in a way that allows multiple change sets to be merged if the client needs to call more than one service method to retrieve the data it needs. While such an implementation can be accomplished with self-tracking entities, it is harder than with a change set. Second, the entity definitions themselves are complicated somewhat because they include the tracking information directly instead of keeping that information in a separate structure outside the entities. Often this information can be kept to a minimum, however, so it usually does not have much effect on the usability or maintainability of the entities.

Naturally, self-tracking entities are not as thoroughly decoupled as DTOs, and there are times when more efficient wire formats can be created with DTOs than with self-tracking entities. Nothing prevents you from using a mix of DTOs and self-tracking entities, and, in fact, as long as the structure of the tracking information is kept as simple as possible, it is not difficult to evolve self-tracking entities into DTOs at some later date if that becomes necessary.

Implementing *N*-Tier with the Entity Framework

Having reviewed your options and decided that you need an *n*-tier application, you can select a pattern and a client technology knowing what pitfalls to avoid. Now you're ready to get rolling. But where does the Entity Framework (EF) fit into all this?

The EF provides a foundation for addressing persistence concerns. This foundation includes a declarative mapping between the database and your conceptual entities, which decouples your mid-tier from the database structure; automatic concurrency checks on updates as long as appropriate change-tracking information is supplied; and transparent change tracking on the mid-tier. In addition, the EF is a LINQ provider, which means that it is relatively easy to create sophisticated queries that can help with mapping entities to DTOs.

The EF can be used to implement any of the four patterns described earlier, but various limitations in the first release of the framework (shipped as part of Visual Studio 2008 SP1/.NET 3.5 SP1) make patterns other than the simple entities pattern very difficult to implement. In the upcoming release of the EF in Visual Studio 2010/.NET 4, a number of features have been added to make implementing the other patterns easier. Before we look at the future release, though, let's look at what you can do with the EF now by using the simple entities pattern.

Concurrency Tokens

The first step you need to take before looking at any aspects of *n*-tier development is to create your model and make sure that you have concurrency tokens. You can read about the basics of building a model elsewhere. There are some great tutorials, for instance, available in the Entity Framework section of the MSDN Data Platform Developer Center at msdn.microsoft.com/data/.

The most important point for this discussion, however, is to make sure that you have specified concurrency tokens for each entity. The best option is to use a row version number or an equivalent concept. A row's version automatically changes whenever any part of the row changes in the database. If you cannot use a row version, the next best option is to use something like a time stamp and add a trigger to the database so that the time stamp is updated whenever a row is modified. You can also perform this sort of operation on the client, but that is prone to causing subtle data corruption problems because multiple clients could inadvertently come up with the same new value for the concurrency token. Once you have an appropriate property configured in the database, open the Entity Designer with your model, select the property, and set its Concurrency Mode in the Properties pane to Fixed instead of the default value None. This setting tells the EF to perform concurrency checks using this property. Remember that you can have more than one property in the same entity with Concurrency Mode set to Fixed, but this is usually not necessary.

Serialization

After you have the prerequisites out of the way, the next topic is serialization. You need a way to move your entities between tiers. If you are using the default entity code generated by the EF and you are building a Windows Communication Foundation (WCF) service, your work is done because the EF automatically generates DataContract attributes on the types and DataMember attributes on the persistable properties of your entities. This includes navigation properties, which means that if you retrieve a graph of related entities into memory, the whole graph is serialized automatically. The generated code also supports binary serialization and XML serialization out of the box, but XML serialization applies only to single entities, not to graphs.

Another important concept to understand is that while the default-generated entities support serialization, their change-tracking information is stored in the ObjectStateManager (a part of the ObjectContext), which does not support serialization. In the simple entities pattern, you typically retrieve unmodified entities from the database on the mid-tier and serialize them to the client, which does not need the change-tracking information. That code might look something like this:

```
public Customer GetCustomerByID(string id)
{
    using (var ctx = new NorthwindEntities())
    {
        return ctx.Customers.Where(c => c.CustomerID == id).First();
    }
}
```

When it comes time to perform an update, however, the change-tracking information must be managed somehow, and that leads to the next important part of the EF you need to understand.

Working with the ObjectStateManager

For two-tier persistence operations, the ObjectStateManager does its job automatically for the most part. You don't have to think about it at all. The state manager keeps track of the existence of each entity under its control; its key value; an EntityState value, which can be unchanged, modified, added, or deleted; a list of modified properties; and the original value of each modified property. When you retrieve an entity from the database, it is added to the list of entities tracked by the state manager, and the entity and the state manager work together to maintain the tracking information. If you set a property on the entity, the state of the entity automatically changes to Modified, the property is added to the list of modified properties, and the original value is saved. Similar information is tracked if you add or delete an entity. When you call SaveChanges on the ObjectContext, this tracking information is used to compute the update statements for the database. If the update completes successfully, deleted entities are removed from the context, and all other entities transition to the unchanged state so that the process can start over again.

When you send entities to another tier, however, this automatic tracking process is interrupted. To implement a service method on the mid-tier that performs an update by using information from the client, you need two special methods that exist on the ObjectContext for just this purpose: Attach and ApplyPropertyChanges.

The Attach method tells the state manager to start tracking an entity. Normally, queries automatically attach entities, but if you have an entity that you retrieved some other way (serialized from the client, for example), then you call Attach to start the tracking process. There are two critical things about Attach to keep in mind.

First, at the end of a successful call to Attach, the entity will always be in the unchanged state. If you want to eventually get the entity into some other state, such as modified or deleted, you need to take additional steps to transition the entity to that state. In effect, Attach tells the EF, "Trust me. At least at some point in the past, this is how this entity looked in the database." The value an entity's property has when you attach it will be considered the original value for that property. So, if you retrieve an entity with a query, serialize it to the client, and then serialize it back to the mid-tier, you can use Attach on it rather than querying again. The value of the concurrency token when you attach the entity will be used for concurrency checks. (For more information about the danger of querying again, see my description of the anti-pattern Mishandled Concurrency in the June issue of *MSDN Magazine* at msdn.microsoft.com/magazine/dd882522.aspx.)

The second thing to know about Attach is that if you attach an entity that is part of a graph of related entities, the Attach method will walk the graph and attach each of the entities it finds. This occurs because the EF never allows a graph to be in a mixed state, where it is partially attached and partially not attached. So if the EF attaches one entity in a graph, it needs to make sure that the rest of the graph becomes attached as well.

The ApplyPropertyChanges method implements the other half of a disconnected entity modification scenario. It looks in the ObjectStateManager for another entity with the same key as its argument and compares each regular property of the two entities. When it finds a property that is different, it sets the property value on the entity in the state manager to match the value from the entity passed as an argument to the method. The effect is the same as if you had performed changes directly on the entity in the state manager when it was being tracked. It is important to note that this method operates only on "regular" properties and not on navigation properties, so it affects only a single entity, not an entire graph. It was designed especially for the simple entities pattern, where a new copy of the entity contains all the information you need in its property values—no extra tracking information is required for it to function.

If you put the Attach and ApplyPropertyChanges methods together to create a simple service method for updating an entity, the method might look something like this:

```
public void UpdateCustomer(Customer original, Customer modified)
{
    using (var ctx = new NorthwindEntities())
    {
        ctx.Attach(original);
        ctx.ApplyPropertyChanges(modified.EntityKey.EntitySetName,
          modified);
        ctx.SaveChanges();
    }
}
```

While these methods make implementation of the service easy, this kind of service contract adds some complication to the client which now needs to copy the entity before modifying it. Many times, this level of complexity is more than you want or need on the client. So, instead of using ApplyPropertyChanges, you can attach the modified entity and use some lower-level APIs on the ObjectStateManager to tell it that the entity should be in the modified state and that every property is modified. This approach has the advantage of reducing the data that must travel from the client to the mid-tier (only one copy of the entity) at the expense of increasing the data that is updated in the database in some scenarios (every property will be updated even if the client modified only some because there is no way to tell which properties were modified and which were not). **Figure 1** shows what the code for this approach would look like.

Figure 1 **Update Service Method**

```
public void UpdateCustomer(Customer modified)
{
    using (var ctx = new NorthwindEntities())
    {
        ctx.Attach(modified);
        var stateEntry = ctx.ObjectStateManager.GetObjectStateEntry(modified);
        foreach (var propertyName in stateEntry.CurrentValues
                                 .DataRecordInfo.FieldMetadata
                                 .Select(fm => fm.FieldType.Name))
        {
            stateEntry.SetModifiedProperty(propertyName);
        }
    }
    ctx.SaveChanges();
}
```

Expanding the service to include methods for adding new customers and deleting customers is also straightforward. **Figure 2** shows an example of this code.

This approach can be extended to methods that change relationships between entities or perform other operations. The key concept to remember is that you need to first get the state manager into something like the state it would have been in originally if you had queried the database, then make changes to the entities for the effect you want, and then call SaveChanges.

Patterns Other Than Simple Entities in .NET 3.5 SP1

If you decide to use the first release of the EF to implement one of the other patterns, my first suggestion is to read the next section, which explains how .NET 4 will make things much easier. If your project needs one of the other patterns before .NET 4 is released, however, here are a few things to think about.

The change set pattern can certainly be implemented. You can see a sample of this pattern that was written to work with one of the prerelease betas of the EF at code.msdn.com/entitybag/. This sample has not been updated to work with the 3.5 SP1 version of the EF, but the work required to do that is fairly easily. One key step you might want to adopt even if you choose to build a change set implementation from scratch is to create an ObjectContext on the client with only the conceptual model metadata (no mapping, storage model, or real connection to the database is needed) and use that as a client-side change tracker.

DTOs are also possible. In fact, implementing DTOs is not that much more difficult with the first release of the EF than it will be in later releases. In either case, you have to write your own code or use an automatic mapper to move data between your entities and the DTOs. One idea to consider is to use LINQ projections to copy data from queries directly into your DTOs. For example, if I created a CustomerDTO class that has just name and phone

Figure 2 **Add and Delete Service Methods**

```
public void AddCustomer(Customer customer)
{
    using (var ctx = new NorthwindEntities())
    {
        ctx.AddObject("Customers", customer);
        ctx.SaveChanges();
    }
}

public void DeleteCustomer(Customer customer)
{
    using (var ctx = new NorthwindEntities())
    {
        ctx.Attach(customer);
        ctx.DeleteObject(customer);
        ctx.SaveChanges();
    }
}
```

properties, I could then create a service method that returns a set of CustomerDTOs like this:

```
public List<CustomerDTO> GetCustomerDTOs()
{
    using (var ctx = new NorthwindEntities())
    {
        var query = from c in ctx.Customers
                    select new CustomerDTO()
                    {
                        Name = c.ContactName,
                        Phone = c.Phone
                    };
        return query.ToList();
    }
}
```

Unfortunately, self-tracking entities is the hardest pattern to implement in the SP1 release for two reasons. First, the EF in .NET 3.5 SP1 does not support POCO, so any self-tracking entities that you implement have a dependency on the 3.5 SP1 version of .NET, and the serialization format will not be as suitable for interoperability. You can address this by hand writing proxies for the client, but they will be tricky to implement correctly. Second, one of the nice features of self-tracking entities is that you can create a single graph of related entities with a mix of operations—some entities can be modified, others new, and still others marked for deletion—but implementing a method on the mid-tier to handle such a mixed graph is quite difficult. If you call the Attach method, it will walk the whole graph, attaching everything it can reach. Similarly, if you call the AddObject method, it will walk the whole graph and add everything it can reach. After either of those operations occurs, you will encounter cases in which you cannot easily transition some entities to their intended final state because the state manager allows only certain state transitions. You can move an entity from unchanged to modified, for instance, but you cannot move it from unchanged to added. To attach a mixed graph to the context, you need to shred your graph into individual entities, add or attach each one separately, and then reconnect the relationships. This code is very difficult.

API Improvements in .NET 4

In the upcoming release of the EF, which will ship with Visual Studio 2010 and .NET 4, we have made a number of improvements to ease the pain of implementing *n*-tier patterns—especially self-tracking entities. I'll touch on some of the most important features in the following sections.

POCO

The EF will support complete persistence ignorance for entity classes. This means that you can create entities that have no dependencies on the EF or other persistence-related DLLs. A single entity class used for persisting data with the EF will also work on Silverlight or earlier versions of .NET. Also, POCO helps isolate the business logic in your entities from persistence concerns and makes it possible to create classes with a very clean, interoperable serialization format.

Improved *N*-Tier Support APIs

Working with the ObjectStateManager will be easier because we have relaxed the state transition constraints. It will be possible to first add or attach an entire graph and then walk over that graph changing entities to the right state. You will be able to set the original values of entities, change the state of an entity to any value, and change the state of a relationship.

Foreign Key Property Support

The first release of the EF supports modeling relationships only as completely separate from entities, which means that the only way to change relationships is through the navigation properties or the RelationshipManager. In the upcoming release, you'll be able to build a model in which an entity exposes a foreign key property that can be manipulated directly.

T4-Based Code Generation

The final important change to the EF will be the use of the T4 template engine to allow easy, complete control over the code that is generated for entities. This is important because it means Microsoft can create and release templates that generate code for a variety of scenarios and usage patterns, and you can customize those templates or even write your own. One of the templates we will release will produce classes that implement the self-tracking entities pattern with no custom coding required on your part. The resulting classes allow the creation of very simple clients and services.

More to Learn

I hope this article has given you a good survey of the design issues involved in creating *n*-tier applications and some specific hints for implementing those designs with the Entity Framework. There is certainly a lot more to learn, so I encourage you to take a look at the Application Architecture Guide from the patterns & practices group (codeplex.com/AppArchGuide/) and the Entity Framework FAQ at blogs.msdn.com/dsimmons/pages/entity-framework-faq.aspx. ■

DANNY SIMMONS *is dev manager for the Entity Framework team at Microsoft. You can read more of his thoughts on the Entity Framework and other subjects at blogs.msdn.com/dsimmons.*

Employing The Domain Model Pattern

Udi Dahan

If you had come to me a few years ago and asked me if I ever used the domain model pattern, I would have responded with an absolute "yes." I was sure of my understanding of the pattern. I had supporting technologies that made it work.

But, I would have been completely wrong.

My understanding has evolved over the years, and I've gained an appreciation for how an application can benefit from aligning itself with those same domain-driven principles.

In this article, we'll go through why we'd even want to consider employing the domain model pattern (as well as why not), the benefits it is supposed to bring, how it interacts with other parts of an application, and the features we'd want to be provided by supporting technologies, and discuss some practical tips on keeping the overall solution as simple as possible.

What Is It?

The author of the domain model pattern, Martin Fowler, provides this definition (Fowler, 2003):

An object model of the domain that incorporates both behavior and data.

To tell you the truth, this definition can be interpreted to fit almost any piece of code—a fairly good reason why I thought I was using the pattern when in fact I wasn't being true to its original intent.

Let's dig deeper.

Reasons Not to Use the Domain Model

In the text following the original description, I had originally blown past this innocuous passage, but it turns out that many important decisions hinge on understanding it.

Since the behavior of the business is subject to a lot of change, it's important to be able to modify, build, and test this layer easily. As a result you'll want the minimum of coupling from the Domain Model to other layers in the system.

So one reason not to make use of the domain model pattern is if the business your software is automating does not change much. That's not to say it does not change *at all*—but rather that the underlying

In this article, we'll go through the reasons to (and not to) employ the domain model pattern, the benefits it brings, as well as provide some practical tips on keeping the overall solution as simple as possible.

This article discusses:

- Domain model pattern
- Scenarios for using the domain model pattern
- Domain events
- Keeping the business in the domain

Technologies discussed:

Domain Model Pattern

rules dictating how business is done aren't very dynamic. While other technological and environmental factors may change, that is not the context of this pattern.

Technology

Some examples of this include supporting multiple databases (such as SQL Server and Oracle) or multiple UI technologies (Windows, Web, Mobile, and so on). If the behavior of the business hasn't changed, these do not justify the use of the domain model pattern. That is not to say one could not get a great deal of value from using technologies that support the pattern, but we need to be honest about which rules we break and why.

Reasons to Use the Domain Model

In those cases where the behavior of the business is subject to a lot of change, having a domain model will decrease the total cost of those changes. Having all the behavior of the business that is likely to change encapsulated in a single part of our software decreases the amount of time we need to perform a change because it will all be performed in one place. By isolating that code as much as possible, we decrease the likelihood of changes in other places causing it to break, thus decreasing the time it takes to stabilize the system.

Scenarios for Not Using the Domain Model

This leads us to the No. 1 most common fallacy about employing domain models. I myself was guilty of making this false assumption for a number of years and see now where it led me astray.

Fallacy: Any persistent object model is a domain model

First of all, a persistent object model does not inherently encapsulate all the behaviors of the business that are likely to change. Second, a persistent object model may include functionality that is not likely to change.

The nature of this fallacy is similar to stating that any screwdriver is a hammer. While you can (try to) hammer in nails with a screwdriver, you won't be very effective doing it that way. One could hardly say you were being true to the hammer pattern.

Bringing this back to concrete scenarios we all know and love, let's consider the ever-present requirement that a user's e-mail address should be unique.

For a while, I thought that the whole point of having a domain model was that requirements like this would be implemented there. However, when we consider the guidance that the domain model is about capturing those business behaviors that are subject to change, we can see that this requirement doesn't fit that mold. It is likely that this requirement will never change.

Therefore, choosing to implement such a requirement in the part of the system that is about encapsulating the volatile parts of the business makes little sense, may be difficult to implement, and might not perform that well. Bringing all e-mail addresses into memory would probably get you locked up by the performance police. Even having the domain model call some service, which calls the database, to see if the e-mail address is there is unnecessary. A unique constraint in the database would suffice.

This pragmatic thinking is very much at the core of the domain model pattern and domain-driven design and is what will keep things simple even as we tackle requirements more involved than simple e-mail uniqueness.

Scenarios for Using the Domain Model

Business rules that indicate when certain actions are allowed are good candidates for being implemented in a domain model.

For example, in an e-commerce system a rule stating that a customer may have no more than $1,000 in unpaid orders would likely belong in the domain model. Notice that this rule involves multiple entities and would need to be evaluated in a variety of use cases.

Of course, in a given domain model we'd expect to see many of these kinds of rules, including cases where some rules override others. In our example above, if the user performing a change to an order is the account manager for the account the customer belongs to, then the previous rule does not apply.

It may appear unnecessary to spend time going through which rules need to apply in which use cases and quickly come up with a list of entities and relationships between them—eschewing the "big design up front" that agile practices rail against. However, the business rules and use cases are the very reasons we're applying the domain model pattern in the first place.

When solving these kinds of problems in the past, I wouldn't have thought twice and would have quickly designed a Customer class with a collection of Order objects. But our rules so far indicate only a single property on Customer instead—UnpaidOrdersAmount. We could go through several rules and never actually run into something that clearly pointed to a collection of Orders. In which case, the agile maxim "you aren't gonna need it" (YAGNI) should prevent us from creating that collection.

When looking at how to persist this graph of objects, we may find it expedient to add supporting objects and collections underneath. We need to clearly differentiate between implementation details and core business behaviors that are the responsibility of the domain model.

More Complex Interactions

Consider the requirement that when a customer has made more than $10,000 worth of purchases with our company, they become a "preferred" customer. When a customer becomes a preferred customer, the system should send them an e-mail notifying them of the benefits of our preferred customer program.

What makes this scenario different from the unique e-mail address requirement described previously is that this interaction does necessarily involve the domain model. One option is to implement this logic in the code that calls the domain model as follows:

```
public void SubmitOrder(OrderData data)
{
  bool wasPreferredBefore = customer.IsPreferred;
  // call the domain model for regular order submit logic
  if (customer.IsPreferred && !wasPreferredBefore)
    // send email
}
```

One pitfall that the sample code avoids is that of checking the amount that constitutes when a customer becomes preferred. That logic is appropriately entrusted to the domain model.

Unfortunately, we can see that the sample code is liable to become bloated as more rules are added to the system that needs to be evaluated when orders are submitted. Even if we were to move this code into the domain model, we'd still be left with the following issues.

Cross-Cutting Business Rules

There may be other use cases that result in the customer becoming preferred. We wouldn't want to have to duplicate that logic in multiple places (whether it's in the domain model or not), especially because refactoring to an extracted method would still require capturing the customer's original preferred state.

We may even need to go so far as to include some kind of interception/aspect-oriented programming (AOP) method to avoid the duplication.

It looks like we'd better rethink our approach before we cut ourselves on Occam's razor. Looking at our requirements again may give us some direction.

When a customer has become a [something] the system should [do something].

We seem to be missing a good way of representing this requirement pattern, although this *does* sound like something that an event-based model could handle well. That way, if we're required to do more in the "should do something" part, we could easily implement that as an additional event handler.

Domain Events and Their Callers

Domain events are the way we explicitly represent the first part of the requirement described:

When a [something] has become a [something] ...

While we can implement these events on the entities themselves, it may be advantageous to have them be accessible at the level of the whole domain. Let's compare how the service layer behaves in either case:

```
public void SubmitOrder(OrderData data)
{
   var customer = GetCustomer(data.CustomerId);
   var sendEmail = delegate { /* send email */ };
   customer.BecamePreferred += sendEmail;
   // call the domain model for the rest of the regular order submit logic
   customer.BecamePreferred -= sendEmail; // to avoid leaking memory
}
```

While it's nice not having to check the state before and after the call, we've traded that complexity with that of subscribing and removing subscriptions from the domain event. Also, code that calls the domain model in any use case shouldn't have to know if a customer can become preferred there. When the code is directly interacting with the customer, this isn't such a big deal. But consider that when submitting an order, we may bring the inventory of one of the order products below its replenishment threshold—we wouldn't want to handle that event in the code, too.

It would be better if we could have each event be handled by a dedicated class that didn't deal with any specific use case but could be generically activated as needed in all use cases. Here's what such a class would look like:

```
public class CustomerBecamePreferredHandler : Handles<CustomerBecamePreferred>
{
   public void Handle(CustomerBecamePreferred args)
   {
      // send email to args.Customer
   }
}
```

We'll talk about what kind of infrastructure will make this class magically get called when needed, but let's see what's left of the original submit order code:

```
public void SubmitOrder(OrderData data)
{
   // call the domain model for regular order submit logic
}
```

That's as clean and straightforward as one could hope—our code doesn't need to know anything about events.

Explicit Domain Events

In the CustomerBecamePreferredHandler class we see the reference to a type called CustomerBecamePreferred—an explicit representation in code of the occurrence mentioned in the requirement. This class can be as simple as this:

```
public class CustomerBecamePreferred : IDomainEvent
{
   public Customer Customer { get; set; }
}
```

The next step is to have the ability for any class within our domain model to raise such an event, which is easily accomplished with the following static class that makes use of a container like Unity, Castle, or Spring.NET:

```
public static class DomainEvents
{
   public IContainer Container { get; set; }
   public static void Raise<T>(T args) where T : IDomainEvent
   {
      foreach(var handler in Container.ResolveAll<Handles<T>>())
         handler.Handle(args);
   }
}
```

Now, any class in our domain model can raise a domain event, with entity classes usually raising the events like so:

```
public class Customer
{
   public void DoSomething()
   {
      // regular logic (that also makes IsPreferred = true)
      DomainEvents.Raise(new CustomerBecamePreferred() { Customer = this });
   }
}
```

Testability

While the DomainEvents class shown is functional, it can make unit testing a domain model somewhat cumbersome as we'd need to make use of a container to check that domain events were raised. Some additions to the DomainEvents class can sidestep the issue, as shown in **Figure 1**.

Now a unit test could be entirely self-contained without needing a container, as **Figure 2** shows.

Commands and Queries

The use cases we've been examining so far have all dealt with changing data and the rules around them. Yet in many systems,

Figure 1 **Additions to the DomainEvents Class**

```
public static class DomainEvents
{
    [ThreadStatic] //so that each thread has its own callbacks
    private static List<Delegate> actions;

    public IContainer Container { get; set; } //as before

    //Registers a callback for the given domain event
    public static void Register<T>(Action<T> callback) where T : IDomainEvent
    {
       if (actions == null)
          actions = new List<Delegate>();

       actions.Add(callback);
   }

   //Clears callbacks passed to Register on the current thread
   public static void ClearCallbacks ()
   {
       actions = null;
   }

   //Raises the given domain event
   public static void Raise<T>(T args) where T : IDomainEvent
   {
      foreach(var handler in Container.ResolveAll<Handles<T>>())
         handler.Handle(args);

      if (actions != null)
          foreach (var action in actions)
              if (action is Action<T>)
                  ((Action<T>)action)(args);
   }
}
```

users will also need to be able to view this data, as well as perform all sorts of searches, sorts, and filters.

I had originally thought that the same entity classes that were in the domain model should be used for showing data to the user. Over the years, I've been getting used to understanding that my original thinking often turns out to be wrong. The domain model is all about encapsulating data with business behaviors.

Showing user information involves no business behavior and is all about opening up that data. Even when we have certain security-related requirements around which users can see what information, that often can be represented as a mandatory filtering of the data.

While I was "successful" in the past in creating a single persistent object model that handled both commands and queries, it was often very difficult to scale it, as each part of the system tugged the model in a different direction.

Figure 2 **Unit Test Without Container**

```
public class UnitTest
{
     public void DoSomethingShouldMakeCustomerPreferred()
     {
         var c = new Customer();
         Customer preferred = null;

         DomainEvents.Register<CustomerBecamePreferred>(
             p => preferred = p.Customer
                 );

         c.DoSomething();
         Assert(preferred == c && c.IsPreferred);
     }
}
```

It turns out that developers often take on more strenuous requirements than the business actually needs. The decision to use the domain model entities for showing information to the user is just such an example.

You see, in a multi-user system, changes made by one user don't necessarily have to be immediately visible to all other users. We all implicitly understand this when we introduce caching to improve performance—but the deeper questions remain: If you don't need the most up-to-date data, why go through the domain model that necessarily works on that data? If you don't need the behavior found on those domain model classes, why plough through them to get at their data?

For those old enough to remember, the best practices around using COM+ guided us to create separate components for read-only and for read-write logic. Here we are, a decade later, with new technologies like the Entity Framework, yet those same principles continue to hold.

Getting data from a database and showing it to a user is a fairly trivial problem to solve these days. This can be as simple as using an ADO.NET data reader or data set.

Figure 3 shows what our "new" architecture might look like.

One thing that is different in this model from common approaches based on two-way data binding, is that the structure that is used to show the data isn't used for changes. This makes things like change-tracking not entirely necessary.

In this architecture, data flows up the right side from the database to the user in the form of queries and down the left side from the user back to the database in the form of commands. Choosing to go to a fully separate database used for those queries is a compelling

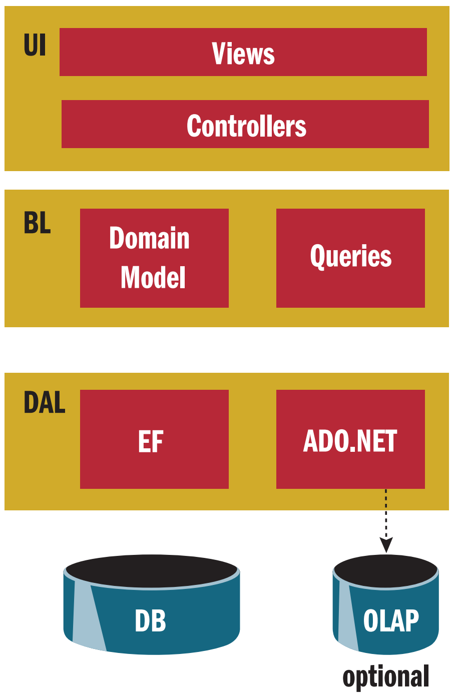


Figure 3 **Model for Getting Data from a Database**

option in terms of performance and scalability, as reads don't interfere with writes in the database (including which pages of data are kept in memory in the database), yet an explicit synchronization mechanism between the two is required. Options for this include ADO.NET Sync Services, SQL Server Integration Services (SSIS), and publish/subscribe messaging. Choosing one of these options is beyond the scope of this article.

Keeping the Business in the Domain

One of the challenges facing developers when designing a domain model is how to ensure that business logic doesn't bleed out of the domain model. There is no silver-bullet solution to this, but one style of working does manage to find a delicate balance between concurrency, correctness, and domain encapsulation that can even be tested for with static analysis tools like FxCop.

Here is an example of the kind of code we wouldn't want to see interacting with a domain model:

```
public void SubmitOrder(OrderData data)
{
   var customer = GetCustomer(data.CustomerId);
   var shoppingCart = GetShoppingCart(data.CartId);
   if (customer.UnpaidOrdersAmount + shoppingCart.Total > Max)
      // fail (no discussion of exceptions vs returns codes here)
   else
      customer.Purchase(shoppingCart);
}
```

Although this code is quite object-oriented, we can see that a certain amount of business logic is being performed here rather than in the domain model. A preferable approach would be this:

```
public void SubmitOrder(OrderData data)
{
   var customer = GetCustomer(data.CustomerId);
   var shoppingCart = GetShoppingCart(data.CartId);
   customer.Purchase(shoppingCart);
}
```

In the case of the new order exceeding the limit of unpaid orders, that would be represented by a domain event, handled by a separate class as demonstrated previously. The purchase method would not cause any data changes in that case, resulting in a technically successful transaction without any business effect.

When inspecting the difference between the two code samples, we can see that calling only a single method on the domain model necessarily means that all business logic has to be encapsulated there. The more focused API of the domain model often further improves testability.

While this is a good step in the right direction, it does open up some questions about concurrency.

Concurrency

You see, in between the time where we get the customer and the time we ask it to perform the purchase, another transaction can come in and change the customer in such a way that its unpaid order amount is updated. That may cause our transaction to perform the purchase (based on previously retrieved data), although it doesn't comply with the updated state.

The simplest way to solve this issue is for us to cause the customer record to be locked when we originally read it—performed by indicating a transaction isolation level of at least repeatable-read (or serializable—which is the default) as follows:

```
public void SubmitOrder(OrderData data)
{
   using (var scope = new TransactionScope(
         TransactionScopeOption.Required,
         new TransactionOptions() { IsolationLevel = IsolationLevel.
            RepeatableRead }
      ))
   {
      // regular code
   }
}
```

Although this does take a slightly more expensive lock than the read-committed isolation level some high-performance environments have settled on, performance can be maintained at similar levels when the entities involved in a given use case are eagerly loaded and are connected by indexed columns. This is often largely offset by the much simpler applicative coding model, because no code is required for identifying or resolving concurrency issues. When employing a separate database for the query portions of the system and all reads are offloaded from the OLTP database serving the domain model, performance and scalability can be almost identical to read-committed-based solutions.

Finding a Comprehensive Solution

The domain model pattern is indeed a powerful tool in the hands of a skilled craftsman. Like many other developers, the first time I picked up this tool, I over-used it and may even have abused it with less than stellar results. When designing a domain model, spend more time looking at the specifics found in various use cases rather than jumping directly into modeling entity relationships—especially be careful of setting up these relationships for the purposes of showing the user data. That is better served with simple and straightforward database querying, with possibly a thin layer of facade on top of it for some database-provider independence.

When looking at how code outside the domain model interacts with it, look for the agile "simplest thing that could possibly work"—a single method call on a single object from the domain, even in the case when you're working on multiple objects. Domain events can help round out your solution for handling more complex interactions and technological integrations, without introducing any complications.

When starting down this path, it took me some time to adjust my thinking, but the benefits of each pattern were quickly felt. When I began employing all of these patterns together, I found they provided a comprehensive solution to even the most demanding business domains, while keeping all the code in each part of the system small, focused, and testable—everything a developer could want.

Works Cited

Fowler, M. *Patterns of Enterprise Application Architecture*, (Addison Wesley, 2003). ■

UDI DAHAN *Recognized as an MVP, an IASA Master Architect, and Dr. Dobb's SOA Guru, Udi Dahan is The Software Simplist, an independent consultant, speaker, author, and trainer providing high-end services in service-oriented, scalable, and secure enterprise architecture and design. Contact Udi via his blog: UdiDahan.com.*

Applying Entity Framework 4.0 to Your Application Architecture

Tim Mallalieu

David Hill, in his preface to the latest patterns & practices Architecture Guidance, jokes that the key to being a good architect is learning to answer "It depends" to most questions. In this article, I'll take that joke to heart. How can you use the Entity Framework with your application architecture? Well, it depends.

Developers deploy a wide variety of development philosophies and architecture styles. This article explores three common perspectives on application development and describes how the Entity Framework can be employed in each. Specifically, I'll look at the forms-centric, model-centric, and code-centric development styles and their relationship to the Entity Framework.

This article uses beta and CTP software. Some features discussed are not currently available in Visual Studio 2010 beta but will be available in upcoming releases of Visual Studio 2010.

This article uses the following technologies:

- Microsoft Visual Studio 2010, ADO.NET Entity Framework Feature CT1

This article discusses:

- Application development styles
- Design patterns

Application Development Styles

I'll start with a discussion of the various development styles. This discussion does not make strong assumptions about particular methodologies that can be applied within these development styles, and I should note that I've used stereotypes for the purpose of this article. Most development styles blend elements of the models I describe. **Figure 1** shows the relative characteristics of the models I'll discuss.

Forms-Centric. In the forms-centric (or "forms-over-data") style of development, the focus is largely on the construction of the top-level user interface (UI) elements that bind to data. The Microsoft development experience for this style is often a drag-and-drop experience in which you define a data source and then systematically construct a series of forms that can perform create, read, update, and delete (CRUD) operations on the underlying data source. This experience tends to be highly productive and intuitive for a developer. The cost is often that the developer accepts a fairly high degree of prescription from the tools and frameworks being used.

Model-Centric. Model-centric development is a step beyond the forms-centric approach. In model-centric development, a developer defines a model in a visual tool or some textual domain-specific language (DSL) and uses this model as the source for generating classes to program against and a database for persistence. This experience is often handy for tools developers who want to build on existing infrastructure to deliver added value. It is also often useful

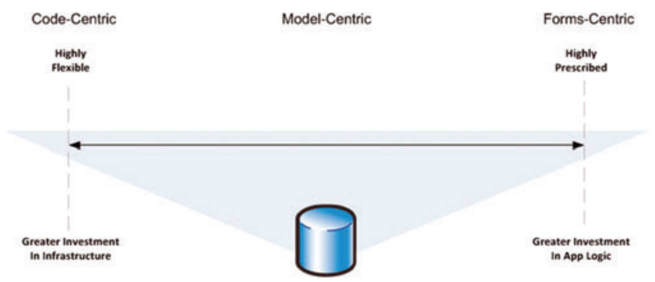


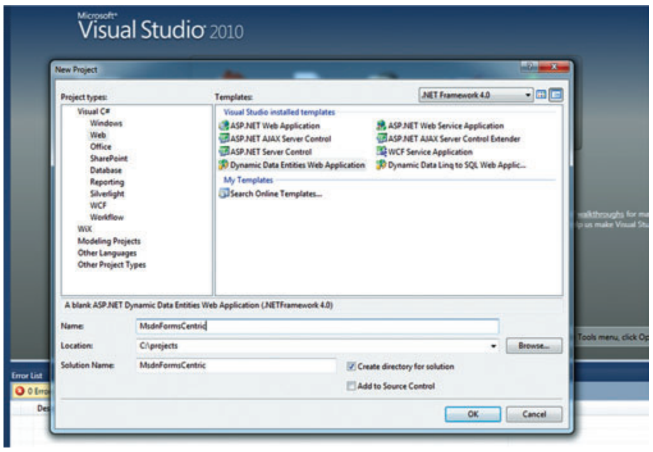
Figure 1 **Development Styles and Their Associated Tradeoffs**

for organizations that want to prescribe their own standards for their application architectures and databases. The cost of this path has historically been in the investment required to enable a complete experience. As with the forms-centric experience, a developer leveraging a model-centric experience tends to give up some flexibility as a consequence of operating in a more prescribed world.

Code-Centric. In the code-centric application development style, the truth is the code. Developers define persistent classes on their own. They elect to write their own data access layer to support these classes, or they use some available persistence offering to do this for them. The main benefit of the code-centric option is that developers get the utmost flexibility. The cost consideration tends to fall on the approach chosen for persistence. If a developer selects a solution that allows her to focus on the business domain instead of the persistence infrastructure, the overall benefit of this approach can be very high.

Building a Forms-Centric Application

In this section, I'll walk through how to build a very simple application using the forms-centric approach with the Entity Framework. The first step is to create a Visual Studio project. For this example, I created a Dynamic Data application. In Visual Studio 2010, you select the Dynamic Data Entities Web Application, as shown in **Figure 2**.



A blank ASP.NET Dynamic Data Entities Web Application (.NETFramework 4.0)

Name: MsdnFormsCentric

Location: C:\projects

Solution Name: MsdnFormsCentric

Figure 2 **Visual Studio 2010 New Project dialog box with Dynamic Data Entities Web Application project template selected.**

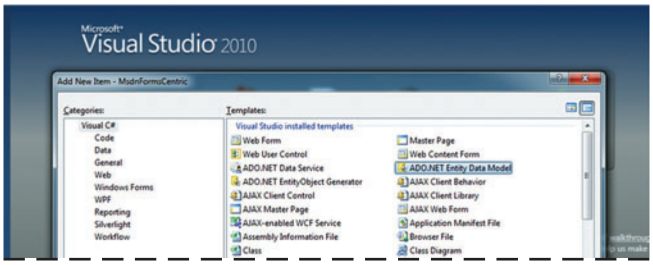


Figure 3 **Add New Item Dialog Box with ADO.NET Entity Data Model Project Item Selected**

The next step is to specify the Entity Framework as a data source for the application. You do this by adding a new ADO.NET Entity Data Model project item to the project, as you can see in **Figure 3**.

After selecting this project item, you perform the following three steps:

1. Choose to start from a database.
2. Choose the database to target.
3. Select the tables to import.

At this point, you click Finish and see the model that is generated from the database, as shown in **Figure 4**.

Now that the model has been generated, using it in the Dynamic Data application is as simple as configuring the form to register the object context that was created in the steps performed earlier. In Global.asax.cs, you can modify the following code to point to the context:

```
DefaultModel.RegisterContext(typeof(NorthwindEntities), new
  ContextConfiguration()
    { ScaffoldAllTables = true});
```

You should now be able to run your application and have a functional set of forms over your persistent data, as shown in **Figure 5**.

This exercise illustrates the most straightforward forms-driven experience. You can now start working on how the presentation should look and what behaviors you need. The ASP.NET Dynamic Data framework uses CLR attributes in the System.

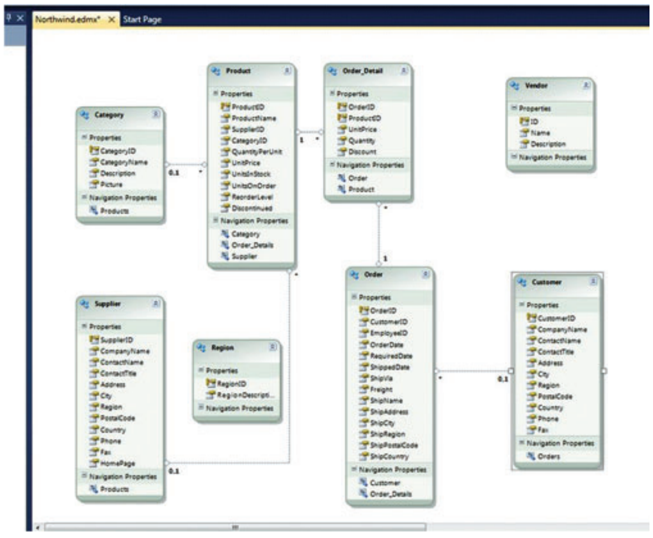
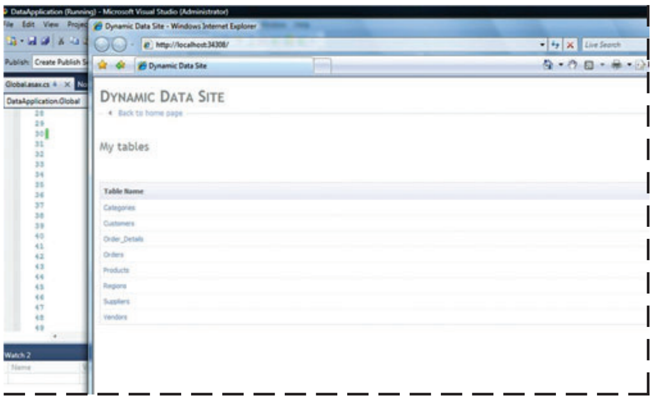


Figure 4 **Default Entity Data Model Created from the Northwind Database**


DataApplication (Running) - Microsoft Visual Studio (Administrator)
Dynamic Data Site - Windows Internet Explorer
http://localhost:34308/
Live Search
Dynamic Data Site
Publish: Create Publish S
Global.asax.cs
DataApplication.Global
Back to home page
Watch 2
Name

DYNAMIC DATA SITE

My tables

Table Name
Categories
Customers
Order_Details
Orders
Products
Regions
Suppliers
Vendors

Figure 5 **Default Dynamic Data Site Using the Entity Framework**

ComponentModel.DataAnnotations namespace to provide guidance on how data can be rendered. For example, you can change how the form is rendered by adding an annotation that hides a particluar column. The attribute is as follows:

```
[ScaffoldColumn(false)]
```

The ScaffoldColumn attribute indicates whether the Dynamic Data framework renders the column. In a case where a table is to be rendered, you can use the ScaffoldColumn attribute to opt out of rendering a specific column. The interesting challenge in the current scenario is where and when do you attribute a column? In this example, the CLR classes, which are used by Dynamic Data, were generated from the Entity Data Model. You can attribute the generated classes, but then any changes to the model will cause the loss of the attributes. Dynamic Data also allows you to apply attributes by using partial classes associated with your entities class, but then you lose some readability and discoverability because of the loss of encapsulation.

Entity Framework 4.0 will provide an extensibility model that allows developers to extend the Entity Framework Designer's tooling surface and add additional metadata that can then be used in code or database generation; however, this functionality is not available in Visual Studio 2010 beta 1.

The Entity Framework developer who wants to work with Dynamic Data can have a very productive experience. He can start with a database and annotate a model with the appropriate metadata to drive much of that experience. After the model is in good shape, the developer can focus on the UI. For more information on using Dynamic Data with the Entity Framework, please take a look at the official Dynamic Data Web site, asp.net/dynamicdata.

Thoughts on Forms-Centric Applications

Using ASP.NET Dynamic Data and the Entity Framework provides a highly productive experience for developing data-centric applications. However, forms-centric applications are not local to Dynamic Data. Many UI-first development experiences that allow developers to build an application by creating a set of screens over a data source tend to share the same characteristics. The developer experience generally relies on some combination of design-time and run-time experiences that prescribe a given architectural style. The data model often reflects the shape of the persistent store (the underlying tables), and there is often a fair bit of UI metadata (such as DataAnnotations in the case of Dynamic Data) that help to define the UI.

The role of the Entity Framework within a forms-centric experience is primarily as the abstraction over the underlying data source. The extensibility capabilities give a developer one true place to define all the model metadata that they need to express. The mapping capabilities allow a developer to reshape the mid-tier domain classes declaratively without having to dive down into infrastructure code.

Building a Model-Centric Application

The promise of model-driven development is that developers can declaratively express a model that is closer to the conceptual business domain than the run-time concerns of a given application architecture. For the purpose of this article, I've focused on the experience of having a single design surface on which you define the domain and related metadata and from which you provision classes and storage.

In the Microsoft .NET Framework 4, there are a number of innovations in Entity Framework tooling that enable a model-centric experience. Entity Framework tooling provides a basic experience plus the capabilities for framework developers, ISVs, and IT organizations to extend those capabilities. To illustrate the experience, I'll walk through a simple application.

I'll start with a new Dynamic Data project again and add an ADO.NET Entity Data Model project item. This time, however, I'll start with a blank model rather than create the model from a database. By starting with a blank surface, you can build out the model you want. I'll build a very simple Fitness application with just two entity types, Workout and WorkoutType. The data models for the types are shown in **Figure 6**.

When you define a model like this in the Entity Framework Designer, there is no mapping or store definition created. However, the Entity Framework Designer now allows developers to create a database script from this model. By right-clicking the designer surface, you can choose Generate Database Script From Model, as shown in **Figure 7,** and the Entity Framework Designer generates a default database from the entity model. For this simple model, two tables are defined. The names of the tables match the EntitySets that are defined in the designer. In the default generation, the database created will build join tables for many-to-many relationships and employ a Table Per Type (TPT) scheme for building tables that must support an inheritance hierarchy.

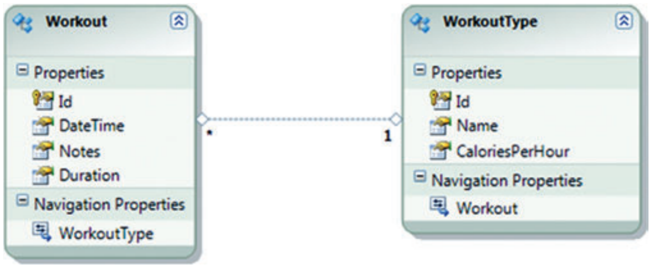


Figure 6 **Simple Entity Data Model**

When you invoke Generate Database Script from Model, a new T-SQL file is added to the project and the Entity Data Model you've created provides the Entity Framework metadata with valid mapping and store descriptions. You can see these in **Figure 8.**

If a developer is using Visual Studio Team Architect or Team Suite, she can deploy and execute the T-SQL script within Visual Studio merely by clicking in the T-SQL file to give it focus and then pressing F5. You are prompted to select the target database, and then the script executes.

At the same time, the Entity Framework Designer runs the default code generation to create classes based on the model, the Entity Framework artifacts required to describe the mapping between the model and the database, and a description of the data store that was created. As a result, you now have a strongly typed data access layer that can be used in the context of your application.

At this point, you've seen only the default experience. The Entity Framework Designer's extensibility allows you to customize many aspects of the model-driven experience. The database-generation and code-generation steps use T4 templates that can be customized to tailor the database schema and the code that is produced. The overall generation process is a Windows Workflow Foundation (WF) workflow that can also be customized, and you have already seen how you can add extensions to the tools surface by using Managed Extensibility Framework–based Visual Studio extensibility. As an example of this extensibility, let's look at how you can change the code-generation step in the project.

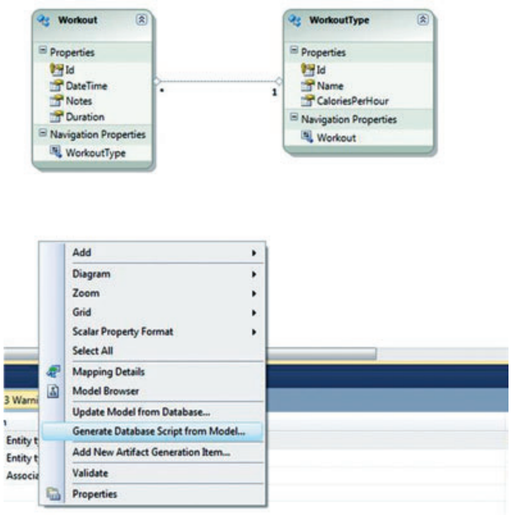
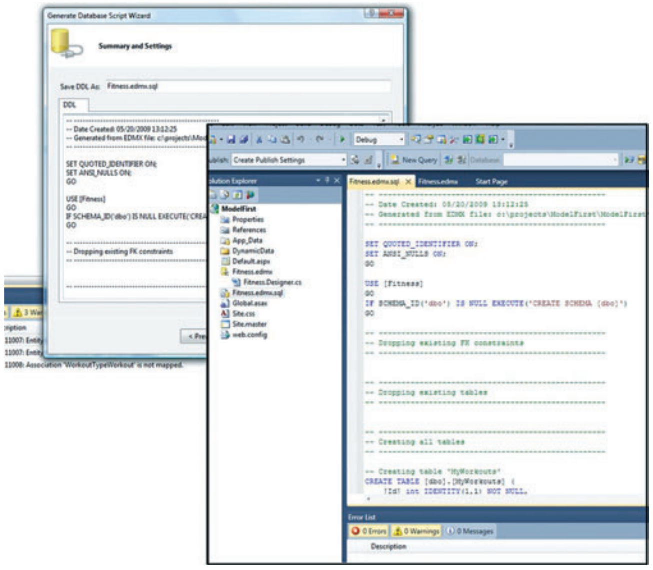


Figure 7 **Generating a Database Script from the Model**

By right-clicking the design surface, you can choose Add New Artifact Generation Item. Choosing this command opens a dialog box in which you can select any of the installed templates to add to the project. In the example shown in **Figure 9,** I selected the Entity Framework POCO Code Generator template (Note: The POCO template does not work with Dynamic Data in Visual Studio 2010 beta 1, but it will work in upcoming releases.) POCO (Plain Old CLR Objects) classes allow developers to define only the items they care about in their classes and avoid polluting them with implementation details from the persistence framework. With .NET 4.0, we have introduced POCO support within the Entity Framework, and one way of creating POCO classes when you are using a model-centric or data-centric development style is with the use of the POCO template. The POCO template is currently available in the ADO.NET Entity Framework Feature CTP 1, which can be downloaded from msdn.microsoft.com/data and used with Visual Studio 2010 beta 1.

By selecting the ADO.NET EF POCO Code Generator template, you get a different set of generated classes. Specifically, you get a set of POCO classes generated as a single file per class, a helper class to use for changes to related items, and a separate context class. Note that you did not do anything to the model. You merely changed the code-generation template.

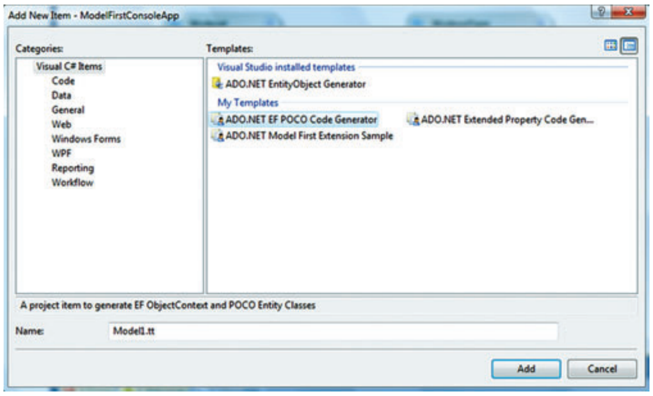
One interesting capability added in .NET 4.0 is the capability to define functions in terms of the Entity Data Model. These functions are expressed in the model and can be referenced in queries. Think about trying to provide a method to determine how many calories are burned in a given workout. There is no property defined on the type that captures the calories burned. You could query the existing types and then enumerate the results, calculating the calories burned in memory; however, by using model-defined functions,



Save DDL As: Fitness.edmx.sql

```
-- --------------------------------------------------
-- Date Created: 05/20/2009 13:12:25
-- Generated from EDMX file: c:\projects\ModelFirst\ModelFirst
-- --------------------------------------------------

SET QUOTED_IDENTIFIER ON;
SET ANSI_NULLS ON;
GO

USE [Fitness]
GO
IF SCHEMA_ID('dbo') IS NULL EXECUTE('CREATE SCHEMA [dbo]')
GO

-- --------------------------------------------------
-- Dropping existing FK constraints
-- --------------------------------------------------


-- --------------------------------------------------
-- Dropping existing tables
-- --------------------------------------------------


-- --------------------------------------------------
-- Creating all tables
-- --------------------------------------------------

-- Creating table 'MyWorkouts'
CREATE TABLE [dbo].[MyWorkouts] (
    [Id] int IDENTITY(1,1) NOT NULL,
```

Figure 8 **The T-SQL File Generated from the Model**


Add New Item - ModelFirstConsoleApp
Categories:
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
Workflow
Templates:
Visual Studio installed templates
ADO.NET EntityObject Generator
My Templates
ADO.NET EF POCO Code Generator
ADO.NET Extended Property Code Gen...
ADO.NET Model First Extension Sample
Add
Cancel

A project item to generate EF ObjectContext and POCO Entity Classes

Name: Model1.tt

Figure 9 **Add New Item Dialog Box**

you can fold this query into the database query that is sent to the store, thus yielding a more efficient operation. You can define the function in the EDMX (XML) as follows:

```
<Function Name="CaloriesBurned" ReturnType="Edm.Int32">
    <Parameter Name="workout" Type="Fitness.Workout" />
    <DefiningExpression>
        workout.Duration * workout.WorkoutType.CaloriesPerHour / 60
    </DefiningExpression>
</Function>
```

To allow this function to be used in a LINQ query, you need to provide a function in code that can be leveraged. You annotate this method to indicate which model function you intend to use. If you want the function to work when directly invoked, you should implement the body. For the purpose of this exercise, we will throw an unsupported exception because we expect to use this function in the form of LINQ queries that will be pushed to the store:

```
[EdmFunction("Fitness", "CaloriesBurned")]
public int CaloriesBurned(Workout workout)
       { throw new NotSupportedException(); }
```

If you want to then build a query to retrieve all high-calorie workouts, where a high-calorie workout is greater than 1,000 calories, you can write the following query:

```
var highCalWorkouts = from w in context.MyWorkouts
                      where
                      context.CaloriesBurned(w) > 1000
                      select w;
```

This LINQ query is a valid query that can now leverage the CaloriesBurned function and be translated to native T-SQL that will be executed in the database.

Thoughts on Model-Centric Application Development

In the degenerate case, where a developer uses the model-first experience and does not customize any of the steps, the model-centric experience is very much like the forms-centric experience. The model the developer is working with is a higher-level model than the logical data model, but it is still a fairly data-centric view of the application.

Developers who extend their Entity Data Model to express more metadata about their domain and who customize the code and/or database generation can come to a place where the experience approaches one in which you define all the metadata for your runtime. This is great for IT organizations that want to prescribe a strict architecture and set of coding standards. It is also very useful for ISVs or framework developers who want to use the Entity Framework Designer as a starting point for describing the model and then generate a broader end-to-end experience from it.

Code-Centric Application Development

The best way to describe code-centric application development is to cite the cliché "the code is the truth." In the forms-centric approach, the focus is on building a data source and UI model for the application. In the model-centric approach, the model is the truth: you define a model, and then generation takes place on both sides (storage and the application). In the code-centric approach, all your intent is captured in code.

One of the challenges of code-centric approaches is the tradeoff between domain logic and infrastructure logic. Object Relational Mapping (ORM) solutions tend to help with code-centric approaches because developers can focus on expressing their domain model in classes and let the ORM take care of the persistence.

As we saw in the model-centric approach, POCO classes can be used with an existing EDM model (in either the model-first or database-first approaches). In the code-centric approach, we use something called Code Only, where we start with just POCO classes and no other artifacts. Code Only is currently available in the ADO.NET Entity Framework Feature CTP 1 (msdn.microsoft.com/data/aa937695.aspx), which can be downloaded from msdn.microsoft.com/data and used with Visual Studio 2010 Beta 1.

Consider replicating the Fitness application using only code. Ideally, you would define the domain classes in code such as shown in **Figure 10.**

To make the domain classes work with the Entity Framework, you need to define a specialized ObjectContext that represents the entry point into the Entity Framework (much like a session or connection abstraction for your interaction with the underlying database). The ObjectContext class must define the EntitySets that you can create LINQ queries on top of. Here's an example of the code:

```
public class FitnessContext : ObjectContext
{
    public FitnessContext(EntityConnection connection)
        : base(connection, "FitnessContext")
    {
    }

    public IObjectSet<Workout> Workouts {
        get { return this.CreateObjectSet<Workout>(); } }

    public IObjectSet<WorkoutType> WorkoutTypes {
        get { return this.CreateObjectSet<WorkoutType>(); } }
}
```

In the code-only experience, a factory class is used to retrieve an instance of the context. This context class reflects over the context and builds up the requisite metadata for the run-time execution. The factory signature is as follows:

```
ContextBuilder.Create<T>(SqlConnection conn)
```

For convenience, you can add a factory method to the generated context. You provide a static field for the connection string and a static factory method to return instances of a FitnessContext. First

Figure 10 **Workout and WorkoutType Domain Classes**

```
public class Workout
{
    public int Id { get; set; }
    public DateTime DateTime { get; set; }
    public string Notes { get; set; }
    public int Duration { get; set; }
    public virtual WorkoutType WorkoutType { get; set; }
}
public class WorkoutType
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int CaloriesPerHour { get; set; }
}
```

the connection string:

```
static readonly string connString = new SqlConnectionStringBuilder
{
    IntegratedSecurity = true,
    DataSource = ".\\sqlexpress",
    InitialCatalog = "FitnessExpress",
}.ConnectionString;
```

And here is the factory method:

```
public static FitnessContext CreateContext()
{
    return ContextBuilder.Create<FitnessContext>(
                           new SqlConnection(connString));
}
```

With this, you have enough to be able to use the context. For example, you could write a method such as the following to query all workout types:

```
public List<WorkoutType> AllWorkoutTypes()
{
    FitnessContext context = FitnessContext.CreateContext();
    return (from w in context.WorkoutTypes select w).ToList();
}
```

As with the model-first experience, it is handy to be able to deploy a database from the code-only experience. The ContextBuilder provides some helper methods that can check whether a database exists, drop it if you want to, and create it.

You can write code like the following to bootstrap a simple set of demo functionality using the code-only approach:

```
public void CreateDatabase()
{
    using (FitnessContext context = FitnessContext.CreateContext())
    {
        if (context.DatabaseExists())
        {
            context.DropDatabase();
        }
        context.CreateDatabase();
    }
}
```

At this point, you can use the Repository pattern from domain-driven design (DDD) to elaborate a bit in what we have seen so far. The use of DDD principles is a common trend in application development today, but I won't attempt to define or evangelize domain driven design here. (For more information, read content from experts such as Eric Evans (*Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003) and Jimmy Nilsson (*Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*, Addison-Wesley, 2006).

Currently, we have a handwritten set of domain classes and a specialized ObjectContext. When we used Dynamic Data, we just pointed the framework at the ObjectContext. But if we want to consider a stronger abstraction of our underlying persistence layer, and if we want to truly constrain the contract of operations to just the meaningful domain operations that one should do, we can leverage the Repository pattern.

For this example, I'll define two repositories: one for WorkoutTypes and one for Workouts. When you follow DDD principles, you should think hard about the aggregate root(s) and then think about modeling the repositories appropriately. In this very simple example, I've used two repositories to illustrate high-level concepts. **Figure 11** shows the WorkoutType repository, and **Figure 12** shows the Workout repository.

One interesting thing to note is that the return types are not IQueryable<T>; they are List<T>. There are debates about whether you should expose IQueryable past the boundaries of the persistence layer. My opinion is that exposing IQueryable breaks the encapsulation of the persistence layer and compromises the boundary between explicit operations that happen in memory and operations that happen in the database. If you

Figure 11 **The WorkoutType Repository**

```
public class WorkoutTypeRepository
{
     public WorkoutTypeRepository()
     {
          _context = FitnessContext.CreateContext();
     }
     public List<WorkoutType> AllWorkoutTypes()
     {
          return _context.WorkoutTypes.ToList();
     }
     public WorkoutType WorkoutTypeForName(string name)
     {
          return (from w in _context.WorkoutTypes
          where w.Name == name
          select w).FirstOrDefault();
     }
     public void AddWorkoutType(WorkoutType workoutType)
     {
          _context.WorkoutTypes.AddObject(workoutType);
     }
     public void Save()
     {
          this._context.SaveChanges();
     }
     private FitnessContext _context;
}
```

Figure 12 **The Workout Repository**

```
public class WorkoutRepository
{
    public WorkoutRepository()
    {
        _context = FitnessContext.CreateContext();
    }

    public Workout WorkoutForId(int Id)
    {
        return (from w in _context.Workouts where w.Id == Id select
          w).FirstOrDefault();
    }

    public List<Workout> WorkoutsForDate(DateTime date)
    {
        return (from w in _context.Workouts where w.DateTime == date
          select w).ToList();
    }

    public Workout CreateWorkout(int id, DateTime dateTime, int
      duration, string notes, WorkoutType workoutType)
    {
        _context.WorkoutTypes.Attach(workoutType);
        Workout workout = new Workout() { Id = id, DateTime =
          dateTime, Duration = duration,
          Notes = notes, WorkoutType = workoutType };
        _context.Workouts.AddObject(workout);
        return workout;
    }

    public void Save()
    {
        _context.SaveChanges();
    }

    private FitnessContext _context;
}
```

Figure 13 **Methods for Building Sample Data**

```
public void AddWorkouts()
{
     Console.WriteLine("--- adding workouts ---");
     WorkoutRepository repository = new WorkoutRepository();
     WorkoutTypeRepository typeRepository = new WorkoutTypeRepository();

     WorkoutType squash = typeRepository.WorkoutTypeForName("Squash");
     WorkoutType running = typeRepository.WorkoutTypeForName("Running");

     repository.CreateWorkout(0,new DateTime(2009, 4, 20, 7, 0, 0),
       60, "nice squash workout", squash);
     repository.CreateWorkout(1, new DateTime(2009, 4, 21, 7, 0, 0),
       180, "long run", running);
     repository.CreateWorkout(2, new DateTime(2009, 4, 22, 7, 0, 0),
       45, "short squash match", squash);
     repository.CreateWorkout(3, new DateTime(2009, 4, 23, 7, 0, 0),
       120, "really long squash", squash);
     repository.Save();
}
```

expose an IQueryable<T> from the repository, you have no idea who will end up composing a database query in LINQ higher up the stack.

You can now use these repositories to add some data in the store. **Figure 13** shows two methods that could be used to create some sample data.

In the model-first scenario, we used model-defined functions to provide a method to determine how many calories are burned in a given workout, even though there is no property defined on the type that captures the calories burned. With the code-only approach, you do not have the option to define model-defined functions here. You can, however, compose on top of the existing Workout EntitySet to define a query that already encapsulates the high-calorie filter, as shown here:

```
public IQueryable<Workout> HighCalorieWorkouts()
{
    return (
      from w in Workouts
      where (w.Duration * w.WorkoutType.CaloriesPerHour / 60) > 1000
         select w);
}
```

If we define this method on the FitnessContext, we can then leverage it in the Workout Repository as follows:

```
public List<Workout> HighCalorieWorkouts()
{
    return _context.HighCalorieWorkouts().ToList();
}
```

Because the method on the context returned an IQueryable, you could have further composed on top of it, but I chose, for symmetry, to just return the results as a List.

Thoughts on Code-Centric Development

The code-centric experience is highly compelling for developers who want to express their domain logic in code. The code-centric experience lends itself well to providing a level of flexibility and clarity needed to work with other frameworks. Using abstractions like the Repository pattern, this approach lets developers provide a high degree of isolation for the persistence layer, which allows the application to remain ignorant of the persistence layer.

Final Thoughts on the Application Development Styles

These are the three application development styles that we often see. As mentioned earlier, there is no single, true classification of these development styles. They lie more on a continuum from highly prescriptive, very data-centric and CRUD-centric experiences that focus on productivity, to highly expressive code-centric experiences.

For all of these, the Entity Framework can be leveraged to provide the persistence layer. As you move toward the form-centric and model-centric side of the spectrum, the explicit model and the ability to extend the model and tool chain can help the Entity Framework improve overall developer productivity. On the code-centric side, the improvements in the Entity Framework allow the runtime to get out of the way and be merely an implementation detail for persistence services. ■

TIM MALLALIEU *is the product unit manager for the Entity Framework and LINQ to SQL. He can be reached at blogs.msdn.com/adonet.*

The Relational Database of the Azure Services Platform

David Robinson

In March of 2008 at the annual MIX conference, Microsoft announced SQL Data Services (SDS), its first data store for the cloud. SDS was an Entity-Attribute-Value (EAV) store that could be accessed using industry standard Internet protocols. It included all the features you would expect from a cloud-based offering, including high availability, fault tolerance, and disaster recovery; all powered by the Microsoft SQL Server engine. Though the initial data model was EAV-based, the more relational features promised at MIX began to be delivered at the Professional Developers Conference in October 2008.

Over the months that followed, the SDS team gathered essential feedback from the user community, most importantly that while the current SDS offering provided a valuable data storage utility, it wasn't SQL Server. What customers wanted was a relational database offered as a service. In March 2009, the SQL Server team announced it was accelerating its plans to offer exactly that, and this was met by overwhelmingly positive feedback from the community. Microsoft has always provided a comprehensive data platform and the new relational capabilities of SDS continue that tradition. With SDS, Microsoft SQL Server now extends from handheld devices with SQL Server CE, to the desktop with SQL Server Express, to the enterprise with SQL Server (both standard and enterprise editions), and now, to the cloud. SDS is the relational database of the Azure Services Platform.

This article is based on a prerelease version of SQL Data Services. All information herein is subject to change.

This article discusses:

- SQL Data Platform
- SQL Data Services Architecture
- Building Applications that Consume SQL Data Services

Technologies discussed:

SQL Data Services

Extending the SQL Data Platform to the Cloud

SDS is the relational database of the Azure Services Platform in the same way that SQL Server is the database of the Windows Server Platform. In the initial offering, only the core relational database features are provided. The research that the product

The TDS Protocol

The native protocol used by clients to communicate with Microsoft SQL Server is called Tabular Data Stream, or TDS. TDS is a well-documented protocol that is used by the underlying Microsoft client components to exchange data with the SQL Server engine. There are even General Public License (GPL) implementations of TDS that can be found on thse Internet.

team has done shows that the current feature set addresses about 95 percent of Web and departmental workloads. When you look at the SQL Server brand, the database engine is only one piece of a larger suite of products. Since SDS uses the same network protocol as the on-premises SQL Server product, all the existing ancillary products continue to work. But though the products will function, they must be run on-premises on your own network. The SQL Server team plans to enable the rest of the SQL Server stack, in the future, to function in the cloud. The end result will be a consistent development experience, whether your solution targets Windows Server or Windows Azure. In fact, the same code will continue to work. All that will be required is a connection string change.

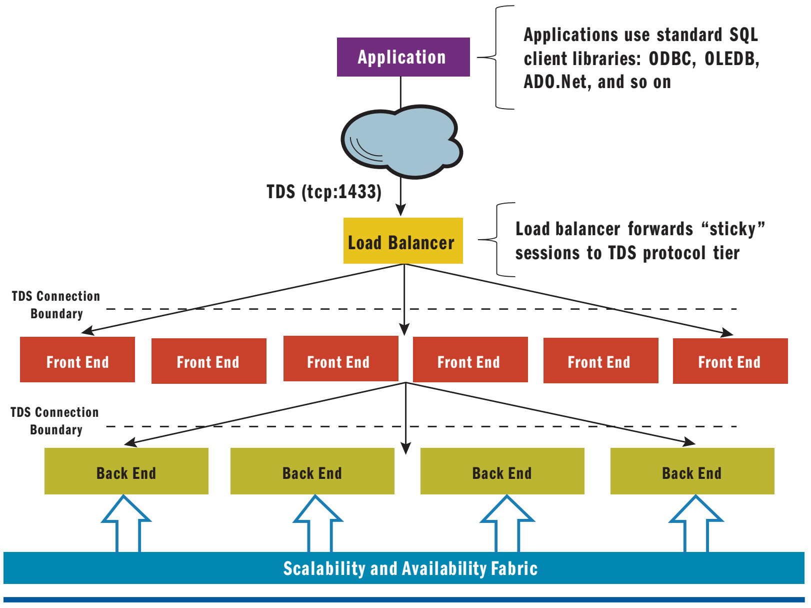

Application
Applications use standard SQL client libraries: ODBC, OLEDB, ADO.Net, and so on
TDS (tcp:1433)
Load Balancer
Load balancer forwards "sticky" sessions to TDS protocol tier
TDS Connection Boundary
Front End
Front End
Front End
Front End
Front End
Front End
TDS Connection Boundary
Back End
Back End
Back End
Back End
Scalability and Availability Fabric

Figure 1 **SQL Data Services Architecture**

SDS Architecture

As mentioned earlier, SDS provides a SQL Server database as a utility service. Features like high availability, fault tolerance and disaster recovery are built in. **Figure 1** provides a view of the SDS architecture. Let's take a look.

SQL Data Services Front End

The SDS front-end servers are the Internet-facing machines that expose the TDS protocol over port 1433. In addition to acting as the gateway to the service, these servers also provide some necessary customer features, such as account provisioning, billing, and usage monitoring. Most importantly, the servers are in charge of routing requests to the appropriate back-end server. SDS maintains a directory that keeps track of where on the SDS back-end servers your primary data and all the backup replicas are located. When you connect to SDS, the front end looks in the directory to see where your database is located and forwards the request to that specific back-end node.

SQL Data Services Back End

The SDS back-end servers, or data nodes, are where the SQL Server engine lives, and it is in charge of providing all the relational database services that an application will consume. The product team is often asked why SDS provides only a subset of the features found in the on-premises SQL Server product. The reason for this is that the feature surface area of SQL Server is extremely large. A significant amount of engineering and testing goes into each feature area that is exposed in SDS, to ensure that the feature is hardened and that a customer's data is completely siloed from all the other SDS customer data. By providing the core relational features that address 95 percent of Web and departmental applications, the team could get the product to market sooner. And, because SDS is an Internet service, we are able to be much more agile and provide new features at a faster pace. Over time, you can expect to see most of the features in the on-premises product available in SDS.

The SDS back end receives the TDS connection from the front end and processes all CRUD (Create, Retrieve, Update, Delete) operations. What features are currently supported? Everything you have come to expect from a relational database, as listed in "Supported Features."

SQL Data Services Fabric

The SDS fabric is in charge of maintaining the fault tolerance and high availability of the system. The fabric plays a key role in the SDS system of automatic failure detection, self-healing and load balancing across all the SDS back-end data nodes. Earlier on, we discussed how SDS maintains a primary copy of your data as well as a series of backup replicas. The fabric provides SDS automatic failure detection. If the node where the primary copy of your data exists experiences a failure, the fabric automatically promotes one of the backup replicas to primary and reroutes the requests. Once the Fabric sees that the failover has occurred, it automatically

Supported Features

In version 1, SDS will support
- Tables, indexes and views
- Stored procedures
- Triggers
- Constraints
- Table variables, session temp tables (#t)

The following are out of scope for SDS v1
- Distributed transactions
- Distributed query
- CLR
- Service Broker
- Spatial data types
- Physical server or catalog DDL and views

rebuilds the backup replica in case another failure should occur.

Connecting to SQL Data Services

This is the part of the article where the SDS team hopes I put you to sleep. The fact of the matter is that because SDS exposes the TDS protocol, all the existing clients like ADO.Net and ODBC just work. Take, for example, the following ADO.Net connection string:

```
SqlConnection conn = new SqlConnection("Data
  Source=testserver; Database=northwind; encrypt=true;
  User ID=david; Password=M5DNROck5");
```

To connect to SDS, that string would look like this:

```
SqlConnection conn = new SqlConnection("Data
  Source=testserver.database.windows.net;
  Database=northwind; encrypt=true; User ID=david;
  Password=M5DNROck5");
```

All that's changed is where the server is located. Note that the string includes the optional parameter encrypt=true. This parameter is not optional for SDS, which requires that all communication be over an encrypted SSL channel. If you try to connect without encryption, the SDS front end will terminate the connection. Because of the TDS protocol, all your existing knowledge, tools and techniques developing against SQL Server still apply. The only thing you need to be concerned about is where your application will run and its proximity to the data.

Building Applications that Consume SQL Data Services

As previously mentioned, one of the main things you need to be concerned with when storing data in SDS is where your application code will run—whether your application follows a "Code Near" architecture or a "Code Far" architecture.

Code Near A Code Near application typically means that your data and your data access components are located on the same network segment, for example when you have your application running on your corporate network. In the case of the Azure Services Platform, it would mean having your application running in Windows Azure and your data residing in SDS. When the Azure platform goes live later this year, you will have the option of picking the region where your application will be hosted as well as the region where your data will be hosted. As long as you choose the same region for both, your application code will be accessing data within the same datacenter, as shown in **Figure 2**.

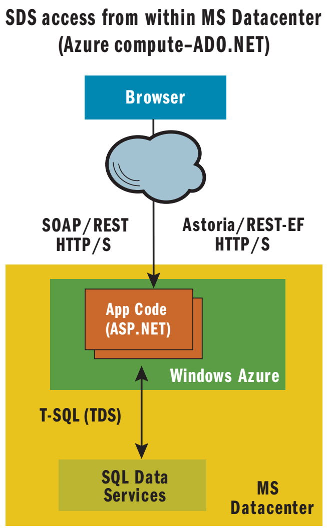


Figure 2 **Code Near Application**

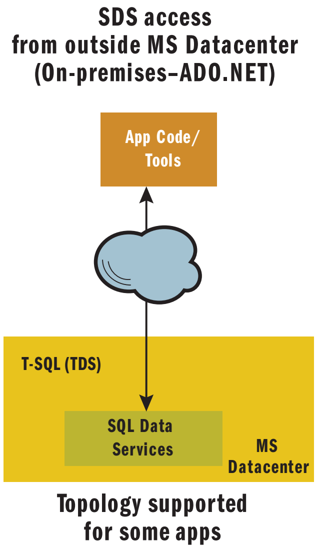


Figure 3 **Code Far Application**

Code Far When your application is a Code Far application, this typically means having your data and data access components on separate networks as shown in **Figure 3**, often with the Internet in between. The Internet has been an incredible enabler for business and technology, but from a data-access perspective, it does pose some interesting challenges, depending on your application and its architecture.

Suppose, for example, that your application provided some sort of data archival service to your customers. In this scenario, the typical pattern is write once, read seldom (or never), and latency would not be too much of a concern.

On the flip side, suppose your application was highly transactional with many reads and writes per second. The performance of this type of application would be poor if it was running on your corporate network and the data was located in SDS. Some sort of data cache, perhaps the project code-named "Velocity" might help, but as application architects and developers, we need to look at each application on a case-by-case basis to identify the best architecture for the application's purposes.

New Face of SQL Data Services

SDS is the relational database of the Azure Services Platform, which will be commercially available at PDC 09 this November. SDS currently provides the key relational features you have come to know and love from SQL Server. Over time, additional features will be enabled, as well as support for additional products in the SQL Server family, such as SQL Server Reporting Services and SQL Server Analysis Services. Because SDS is accessed over TDS—the same protocol as SQL Server—all the existing tools, client libraries and development techniques continue to work. I hope that by reading this article you have been given a glimpse of the new face of SDS, and that you can see that it is truly an extension of SQL Server in the cloud. ■

DAVID ROBINSON *is a Senior Program Manager on the SQL Server Data Services team at Microsoft. He spends his time driving new and compelling features into the product. He also enjoys doing presentations at community events and getting feedback from customers on SDS.*

Cryptographic Agility

Throughout history, people have used various forms of ciphers to conceal information from their adversaries. Julius Caesar used a three-place shift cipher (the letter A is converted to D, B is converted to E, and so on) to communicate battle plans. During World War II, the German navy used a significantly more advanced system—the Enigma machine—to encrypt messages sent to their U-boats. Today, we use even more sophisticated encryption mechanisms as part of the public key infrastructure that helps us perform secure transactions on the Internet.

But for as long as cryptographers have been making secret codes, cryptanalysts have been trying to break them and steal information, and sometimes the code breakers succeed. Cryptographic algorithms once considered secure are broken and rendered useless. Sometimes subtle flaws are found in the algorithms, and sometimes it is simply a matter of attackers having access to more computing power to perform brute-force attacks.

Recently, security researchers have demonstrated weaknesses in the MD5 hash algorithm as the result of collisions; that is, they have shown that two messages can have the same computed MD5 hash value. They have created a proof-of-concept attack against this weakness targeted at the public key infrastructures that protect e-commerce transactions on the Web. By purchasing a specially crafted Web site certificate from a certificate authority (CA) that uses MD5 to sign its certificates, the researchers were able to create a rogue CA certificate that could effectively be used to impersonate potentially any site on the Internet. They concluded that MD5 is not appropriate for signing digital certificates and that a stronger alternative, such as one of the SHA-2 algorithms, should be used. (If you're interested in learning more about this research, you can read the white paper at win.tue.nl/hashclash/rogue-ca/.)

These findings are certainly cause for concern, but they are not a huge surprise. Theoretical MD5 weaknesses have been demonstrated for years, and the use of MD5 in Microsoft products has been banned by the Microsoft SDL cryptographic standards since 2005. Other once-popular algorithms, such as SHA-1 and RC2, have been similarly banned. **Figure 1** shows a complete list of the cryptographic algorithms banned or approved by the SDL. The list of SDL-approved algorithms is current as of this writing, but this list is reviewed and updated annually as part of the SDL update process.

Rather than hard-code specific cryptographic algorithms, use one of the crypto-agility features built into .NET.

Even if you follow these standards in your own code, using only the most secure algorithms and the longest key lengths, there's no guarantee that the code you write today will remain secure. In fact, it will probably not remain secure if history is any guide.

Planning for Future Exploits

You can address this unpleasant scenario reactively by going through your old applications' code bases, picking out instantiations of vulnerable algorithms, replacing them with new algorithms, rebuilding the applications, running them through regression tests, and then issuing patches or service packs to your users. This is not only a lot of work for you, but it still leaves your users at risk until you can get the fixes shipped.

A better alternative is to plan for this scenario from the beginning. Rather than hard-coding specific cryptographic algorithms into your code, use one of the crypto-agility features built into the Microsoft .NET Framework. Let's take a look at a few C# code snippets, starting with the least agile example:

```
private static byte[] computeHash(byte[] buffer)
{
   using (MD5CryptoServiceProvider md5 = new MD5CryptoServiceProvider())
   {
      return md5.ComputeHash(buffer);
   }
}
```

This code is completely nonagile. It is tied to a specific algorithm (MD5) as well as a specific implementation of that algorithm, the MD5CryptoServiceProvider class. Modifying this application to use a secure hashing algorithm would require changing code and issuing a patch. Here's a little better example:

```
private static byte[] computeHash(byte[] buffer)
{
   string md5Impl = ConfigurationManager.AppSettings["md5Impl"];
   if (md5Impl == null)
      md5Impl = String.Empty;

   using (MD5 md5 = MD5.Create(md5Impl))
   {
      return md5.ComputeHash(buffer);
   }
}
```

Send your questions and comments to briefs@microsoft.com.

Figure 1 **SDL-Approved Cryptographic Algorithms**

Algorithm Type	Banned (algorithms to be replaced in existing code or used only for decryption)	Acceptable (algorithms acceptable for existing code, except sensitive data)	Recommended (algorithms for new code)
Symmetric Block	DES, DESX, RC2, SKIPJACK	3DES (2 or 3 key)	AES (>=128 bit)
Symmetric Stream	SEAL, CYLINK_MEK, RC4 (<128bit)	RC4 (>= 128bit)	None, block cipher is preferred
Asymmetric	RSA (<2048 bit), Diffie-Hellman (<2048 bit)	RSA (>=2048bit), Diffie-Hellman (>=2048bit)	RSA (>=2048bit), Diffie-Hellman (>=2048bit), ECC (>=256bit)
Hash (includes HMAC usage)	SHA-0 (SHA), SHA-1, MD2, MD4, MD5	SHA-2	SHA-2 (includes: SHA-256, SHA-384, SHA-512)
HMAC Key Lengths	<112bit	>= 112bit	>= 128bit

This function uses the System.Configuration.ConfigurationManager class to retrieve a custom app setting (the "md5Impl" setting) from the application's configuration file. In this case, the setting is used to store the strong name of the algorithm implementation class you want to use. The code passes the retrieved value of this setting to the static function MD5.Create to create an instance of the desired class. (System.Security.Cryptography.MD5 is an abstract base class from which all implementations of the MD5 algorithm must derive.) For example, if the application setting for md5Impl was set to the string "System.Security.Cryptography.MD5Cng, System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089", MD5.Create would create an instance of the MD5Cng class.

This approach solves only half of our crypto-agility problem, so it really is no solution at all. We can now specify an implementation of the MD5 algorithm without having to change any source code, which might prove useful if a flaw is discovered in a specific implementation, like MD5Cng, but we're still tied to the use of MD5 in general. To solve this problem, we keep abstracting upward:

```
private static byte[] computeHash(byte[] buffer)
{
   using (HashAlgorithm hash = HashAlgorithm.Create("MD5"))
   {
      return hash.ComputeHash(buffer);
   }
}
```

At first glance, this code snippet does not look substantially different from the first example. It looks like we've once again hard-coded the MD5 algorithm into the application via the call to HashAlgorithm.Create("MD5"). Surprisingly though, this code is substantially more cryptographically agile than both of the other examples. While the default behavior of the method call HashAlgorithm.Create("MD5")—as of .NET 3.5—is to create an instance of the MD5CryptoServiceProvider class, the runtime behavior can be customized by making a change to the machine.config file.

Let's change the behavior of this code to create an instance of the SHA512algorithm instead of MD5. To do this, we need to add two elements to the machine.config file: a <cryptoClass> element to map a friendly algorithm name to the algorithm implementation class we want; and a <nameEntry> element to map the old, deprecated algorithm's friendly name to the new friendly name.

```
<configuration>
  <mscorlib>
    <cryptographySettings>
      <cryptoNameMapping>
        <cryptoClasses>
          <cryptoClass MyPreferredHash="SHA512CryptoServiceProvider,
            System.Core, Version=3.5.0.0, Culture=neutral,
            PublicKeyToken=b77a5c561934e089"/>
        </cryptoClasses>
        <nameEntry name="MD5" class="MyPreferredHash"/>
      </cryptoNameMapping>
    </cryptographySettings>
  </mscorlib>
</configuration>
```

Now, when our code makes its call to HashAlgorithm.Create("MD5"), the CLR looks in the machine.config file and sees that the string "MD5" should map to the friendly algorithm name "MyPreferredHash". It then sees that "MyPreferredHash" maps to the class SHA512CryptoServiceProvider (as defined in the assembly System.Core, with the specified version, culture, and public key token) and creates an instance of that class.

It's important to note that the algorithm remapping takes place not at compile time but at run time: it's the user's machine.config that controls the remapping, not the developer's. As a result, this technique solves our dilemma of being tied to a particular algorithm that might be broken at some time in the future. By avoiding hard-coding the cryptographic algorithm class into the application—coding only the abstract type of cryptographic algorithm, HashAlgorithm, instead—we create an application in which the end user (more specifically, someone with administrative rights to edit the machine.config file on the machine where the application is installed) can determine exactly which algorithm and implementation the application will use. An administrator might choose to replace an algorithm that was recently broken with one still considered secure (for example, replace MD5 with SHA-256) or to proactively replace a secure algorithm with an alternative with a longer bit length (replace SHA-256 with SHA-512).

Potential Problems

Modifying the machine.config file to remap the default algorithm-type strings (like "MD5" and "SHA1") might solve crypto-agility problems, but it can create compatibility problems at the same time. Making changes to machine.config affects every .NET application on the machine. Other applications installed on the machine might rely on MD5 specifically, and changing the algorithms used by these applications might break them in unexpected ways that are difficult to diagnose. As an alternative to forcing blanket changes to the entire machine, it's better to use custom, application-specific friendly names in your code and map those name entries to preferred classes in the machine.config. For example, we can change "MD5" to "MyApplicationHash" in our example:

```
private static byte[] computeHash(byte[] buffer)
{
   using (HashAlgorithm hash = HashAlgorithm.Create("MyApplicationHash"))
   {
      return hash.ComputeHash(buffer);
   }
}
```

We then add an entry to the machine.config file to map "MyApplicationHash" to the "MyPreferredHash" class:

```
…
<cryptoClasses>
   <cryptoClass MyPreferredHash="SHA512CryptoServiceProvider,
     System.Core, Version=3.5.0.0, Culture=neutral,
     PublicKeyToken=b77a5c561934e089"/>
</cryptoClasses>
<nameEntry name="MyApplicationHash" class="MyPreferredHash"/>
…
```

You can also map multiple friendly names to the same class; for example, you could have one friendly name for each of your applications, and in this way change the behavior of specific applications without affecting every other application on the machine:

```
…
<cryptoClasses>
   <cryptoClass MyPreferredHash="SHA512CryptoServiceProvider,
     System.Core, Version=3.5.0.0, Culture=neutral,
     PublicKeyToken=b77a5c561934e089"/>
</cryptoClasses>
<nameEntry name="MyApplicationHash" class="MyPreferredHash"/>
<nameEntry name="MyApplication2Hash" class="MyPreferredHash"/>
<nameEntry name="MyApplication3Hash" class="MyPreferredHash"/>
…
```

However, we're still not out of the woods with regard to compatibility problems in our own applications. You need to plan ahead regarding storage size, for both local variables (transient storage) and database and XML schemas (persistent storage). For example, MD5 hashes are always 128 bits in length. If you budget exactly 128 bits in your code or schema to store hash output, you will not be able to upgrade to SHA-256 (256 bit-length output) or SHA-512 (512 bit-length output).

This does beg the question of how much storage is enough. Is 512 bits enough, or should you use 1,024, 2,048, or more? I can't provide a hard rule here because every application has different requirements, but as a rule of thumb I recommend that you budget twice as much space for hashes as you currently use. For symmetric- and asymmetric-encrypted data, you might reserve an extra 10 percent of space at most. It's unlikely that new algorithms with output sizes significantly larger than existing algorithms will be widely accepted.

However, applications that store hash values or encrypted data in a persistent state (for example, in a database or file) have bigger problems than reserving enough space. If you persist data using one algorithm and then try to operate on that data later using a different algorithm, you will not get the results you expect. For example, it's a good idea to store hashes of passwords rather than the full plaintext versions. When the user tries to log on, the code can compare the hash of the password supplied by the user to the stored hash in the database. If the hashes match, the user is authentic. However, if a hash is stored in one format (say, MD5) and an application is upgraded to use another algorithm (say, SHA-256), users will never be able to log on because the SHA-256 hash value of the passwords will always be different from the MD5 hash value of those same passwords.

You can get around this issue in some cases by storing the original algorithm as metadata along with the actual data. Then, when operating on stored data, use the agility methods (or reflection) to instantiate the algorithm originally used instead of the current algorithm:

```
private static bool checkPassword(string password, byte[] storedHash,
   string storedHashAlgorithm)
{
   using (HashAlgorithm hash = HashAlgorithm.Create(storedHashAlgorithm))
   {
      byte[] newHash =
         hash.ComputeHash(System.Text.Encoding.Default.GetBytes(password));
      if (newHash.Length != storedHash.Length)
         return false;
      for (int i = 0; i < newHash.Length; i++)
         if (newHash[i] != storedHash[i])
            return false;
      return true;
   }
}
```

You need to plan ahead regarding storage size for both local variables and database and XML schemas.

Unfortunately, if you ever need to compare two stored hashes, they have to have been created using the same algorithm. There is simply no way to compare an MD5 hash to a SHA-256 hash and determine if they were both created from the same original data. There is no good crypto-agility solution for this problem, and the best advice I can offer is that you should choose the most secure algorithm currently available and then develop an upgrade plan in case that algorithm is broken later. In general, crypto agility tends to work much better for transient data than for persistent data.

Alternative Usage and Syntax

Assuming that your application design allows the use of crypto agility, let's continue to look at some alternative uses and syntaxes for this technique. We've focused almost entirely on cryptographic hashing algorithms to this point in the article, but crypto agility also works for other cryptographic algorithm types. Just call the static Create method of the appropriate abstract base class: SymmetricAlgorithm for symmetric (secret-key) cryptography algorithms such as AES; AsymmetricAlgorithm for asymmetric (public key) cryptography algorithms such as RSA; KeyedHashAlgorithm for keyed hashes; and HMAC for hash-based message authentication codes.

You can also use crypto agility to replace one of the standard .NET cryptographic algorithm classes with a custom algorithm class, such as one of the algorithms developed by the CLR security team and uploaded to CodePlex (clrsecurity.codeplex.com/). However, writing your own custom crypto libraries is highly discouraged. Your homemade algorithm consisting of an ROT13 followed by a bitwise left shift and an XOR against your cat's name might seem secure, but it will pose little challenge to an expert code breaker. Unless you are an expert in cryptography, leave algorithm design to the professionals.

Also resist the temptation to develop your own algorithms—or to revive long-dead, obscure ones, like the Vigenère cipher—even in situations where you don't need cryptographically strong protection. The issue isn't so much what you do with your cipher, but what developers who come after you will do with it. A new developer who finds your custom algorithm class in the code base years later might decide that it's just what he needs for the new product activation key generation logic.

So far we've seen one of the syntaxes for implementing cryptographically agile code, AlgorithmType.Create(algorithmName), but two other approaches are built into the .NET Framework. The first is to use the System.Security.Cryptography.CryptoConfig class:

```
private static byte[] computeHash(byte[] buffer)
{
   using (HashAlgorithm hash = (HashAlgorithm)CryptoConfig.CreateFromName
     ("MyApplicationHash"))
   {
      return hash.ComputeHash(buffer);
   }
}
```

This code performs the same operations as our previous example using HashAlgorithm.Create("MyApplicationHash"): the CLR looks in the machine.config file for a remapping of the string "MyApplicationHash" and uses the remapped algorithm class if it finds one. Notice that we have to cast the result of CryptoConfig.CreateFromName because it has a return type of System.Object and can be used to create SymmetricAlgorithms, AsymmetricAlgorithms, or any other kind of object.

The second alternative syntax is to call the static algorithm Create method in our original example but with no parameters, like this:

```
private static byte[] computeHash(byte[] buffer)
{
   using (HashAlgorithm hash = HashAlgorithm.Create())
   {
      return hash.ComputeHash(buffer);
   }
}
```

In this code, we simply ask the framework to provide an instance of whatever the default hash algorithm implementation is. You can find the list of defaults for each of the System.Security.Cryptography abstract base classes (as of .NET 3.5) in **Figure 2**.

For HashAlgorithm, you can see that the default algorithm is SHA-1 and the default implementation class is SHA1CryptoServiceProvider. However, we know that SHA-1 is banned by the SDL cryptographic standards. For the moment, let's ignore the fact that potential compatibility problems make it generally unwise to remap inherent algorithm names like "SHA1" and alter our machine.config to remap "SHA1" to SHA512CryptoServiceProvider:

```
…
<cryptoClasses>
   <cryptoClass MyPreferredHash="SHA512CryptoServiceProvider,
     System.Core, Version=3.5.0.0, Culture=neutral,
     PublicKeyToken=b77a5c561934e089"/>
</cryptoClasses>
<nameEntry name="SHA1" class="MyPreferredHash"/>
…
```

Figure 2 **Default Algorithms and Implementations in the .NET Framework 3.5**

Abstract Base Class	Default Algorithm	Default Implementation
HashAlgorithm	SHA-1	SHA1CryptoServiceProvider
SymmetricAlgorithm	AES (Rijndael)	RijndaelManaged
AsymmetricAlgorithm	RSA	RSACryptoServiceProvider
KeyedHashAlgorithm	SHA-1	HMACSHA1
HMAC	SHA-1	HMACSHA1

Now let's insert a debug line in the computeHash function to confirm that the algorithm was remapped correctly and then run the application:

```
private static byte[] computeHash(byte[] buffer)
{
   using (HashAlgorithm hash = HashAlgorithm.Create())
   {
      Debug.WriteLine(hash.GetType());
      return hash.ComputeHash(buffer);
   }
}
```

The debug output from this method is:

```
System.Security.Cryptography.SHA1CryptoServiceProvider
```

What happened? Didn't we remap SHA1 to SHA-512? Actually, no, we didn't. We remapped only the *string* "SHA1" to the class SHA512CryptoServiceProvider, and we did not pass the string "SHA1" as a parameter to the call to HashAlgorithm.Create.

Even though Create appears to have no string parameters to remap, it is still possible to change the type of object that is created. You can do this because HashAlgorithm.Create() is just shortcut syntax for HashAlgorithm.Create("System.Security.Cryptography.HashAlgorithm"). Now let's add another line to the machine.config file to remap "System.Security.Cryptography.HashAlgorithm" to SHA512CryptoServiceProvider and then run the app again:

```
…
<cryptoClasses>
   <cryptoClass MyPreferredHash="SHA512CryptoServiceProvider,
     System.Core, Version=3.5.0.0, Culture=neutral,
     PublicKeyToken=b77a5c561934e089"/>
</cryptoClasses>
<nameEntry name="SHA1" class="MyPreferredHash"/>
<nameEntry name="System.Security.Cryptography.HashAlgorithm"
class="MyPreferredHash"/>
…
```

The debug output from computeHash is now exactly what we expected:

```
System.Security.Cryptography.SHA512CryptoServiceProvider
```

However, remember that remapping classes in this way can create unexpected and difficult-to-debug compatibility issues. It's preferable to use application-specific friendly names that can be remapped with less chance of causing problems.

Another Benefit of Crypto Agility

In addition to letting you replace broken algorithms on the fly without having to recompile, crypto agility can be used to improve performance. If you've ever looked at the System.Security.Cryptography namespace, you might have noticed that often several different implementation classes exist for a given algorithm. For example, there are three different implementations of SHA-512: SHA512Cng, SHA512CryptoServiceProvider, and SHA512Managed.

Of these classes, SHA512Cng usually offers the best performance. A quick test on my laptop (running Windows 7 release candidate) shows that the –Cng classes in general are about 10 percent faster than the -CryptoServiceProvider and -Managed classes. My colleagues in the Core Architecture group inform me that in some circumstances the –Cng classes can actually run 10 times faster than the others!

Clearly, using the –Cng classes is preferable, and we could set up our machine.config file to remap algorithm implementations to use those classes, but the -Cng classes are not available on every operating system. Only Windows Vista, Windows Server 2008, and Windows 7 (and later versions, presumably) support –Cng. Trying to instantiate a –Cng class on any other operating system will throw an exception.

Similarly, the –Managed family of crypto classes (AesManaged, RijndaelManaged, SHA256Managed, and so on) are not always available, but for a completely different reason. The Federal Information Processing Standard 140 (FIPS) specifies standards for cryptographic algorithms and implementations. As of this writing, both the –Cng and –CryptoServiceProvider implementation classes are FIPS-certified, but –Managed classes are not. Furthermore, you can configure a Group Policy setting that allows only FIPS-compliant algorithms to be used. Some U.S. and Canadian government agencies mandate this policy setting. If you'd like to check your machine, open the Local Group Policy Editor (gpedit.msc), navigate to the Computer Configuration/Windows Settings/Security Settings/Local Policies/Security Options node, and check the value of the setting "System Cryptography: Use FIPS compliant algorithms for encryption, hashing, and signing". If this policy is set to Enabled, attempting to instantiate a –Managed class on that machine will throw an exception.

This leaves the –CryptoServiceProvider family of classes as the lowest common denominator guaranteed to work on all platforms, but these classes also generally have the worst performance. You can overcome this problem by implementing one of the three crypto-agility syntaxes mentioned earlier in this article and customizing the machine.config file remapping for deployed machines based on their operating system and settings. For machines running Windows Vista or later, we can remap the machine.config to prefer the –Cng implementation classes:

```
…
<cryptoClasses>
   <cryptoClass MyPreferredHash="SHA512Cng, System.Core,
     Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
</cryptoClasses>
<nameEntry name="MyApplicationHash" class="MyPreferredHash"/>
…
```

For machines running operating systems earlier than Windows Vista with FIPS compliance disabled, we can remap machine.config to prefer the –Managed classes:

```
…
<cryptoClasses>
   <cryptoClass MyPreferredHash="SHA512Managed"/>
</cryptoClasses>
<nameEntry name="MyApplicationHash" class="MyPreferredHash"/>
…
```

For all other machines, we remap to the –CryptoServiceProvider classes:

```
…
<cryptoClasses>
   <cryptoClass MyPreferredHash="SHA512CryptoServiceProvider,
     System.Core, Version=3.5.0.0, Culture=neutral,
     PublicKeyToken=b77a5c561934e089"/>
</cryptoClasses>
<nameEntry name="MyApplicationHash" class="MyPreferredHash"/>
…
```

Any call to HashAlgorithm.Create("MyApplicationHash") now creates the highest-performing implementation class available for that machine. Furthermore, since the algorithms are identical, you don't need to worry about compatibility or interoperability issues. A hash created for a given input value on one machine will be the same as a hash created for that same input value on another machine, even if the implementation classes are different. This holds true for the other algorithm types as well: you can encrypt an input on one machine by using AesManaged and decrypt it successfully on a different machine by using AesCryptoServiceProvider.

Wrapping Up

Given the time and expense of recoding your application in response to a broken cryptographic algorithm, not to mention the danger to your users until you can get a new version deployed, it is wise to plan for this occurrence and write your application in a cryptographically agile fashion. The fact that you can also obtain a performance benefit from coding this way is icing on the cake.

Never hardcode specific algorithms or implementations of those algorithms into your application. Always declare cryptographic algorithms as one of the following abstract algorithm type classes: HashAlgorithm, SymmetricAlgorithm, AsymmetricAlgorithm, KeyedHashAlgorithm, or HMAC.

I believe that an FxCop rule that would verify cryptographic agility would be extremely useful. If someone writes such a rule and posts it to Codeplex or another public code repository, I will be more than happy to give them full credit in this space and on the SDL blog (blogs.msdn.com/sdl/).

Finally, I would like to acknowledge Shawn Hernan from the SQL Server security team and Shawn Farkas from the CLR security team for their expert feedback and help in producing this article. ■

BRYAN SULLIVAN *is a security program manager for the Microsoft Security Development Lifecycle team, specializing in Web application security issues. He is the author of* Ajax Security *(Addison-Wesley, 2007).*

How Data Access Code Affects Database Performance

There's been a consistent debate over whether query tuning, and database application performance tuning in general, is the province of the database administrator, the application developer, or both. The database administrator usually has access to more tools than the developer. The DBA can look at the performance monitor counters and dynamic management views, run SQL Profiler, decide where to place the database, and create indexes to make queries perform better. The application developer usually writes the queries and stored procedures that access the database. The developer can use most of the same tools in a test system, and based on knowledge of the application's design and use cases, the developer can recommend useful indexes. But an often overlooked point is that the application developer writes the database API code that accesses the database. Code that accesses the database, such as ADO.NET, OLE DB, or ODBC, can have an effect on database performance. This is especially important when attempting to write a generalized data access framework or choosing an existing framework. In this article, we'll delve into some typical approaches to writing data access code and look at the effect they can have on performance.

Query Plan Reuse

Let's start by going over the lifetime of a query. When a query is submitted through the query processor, the query processor parses the text for syntactic correctness. Queries in stored procedures are syntax-checked during the CREATE PROCEDURE statement. Before the query or stored procedure is executed, the processor checks the plan cache for a query plan that matches the text. If the text matches, the query plan is reused; if no match occurs, a query plan is created for the query text. After the query is executed, the plan is returned to the cache to be reused. Query plan creation is an expensive operation, and query plan reuse is almost always a good idea. The query text that's being compared against text of the plan in the cache must match using a case-sensitive string comparison.

So the query

```
SELECT a.au_id, ta.title_id FROM authors a
JOIN titleauthor ta ON a.au_id = ta.au_id
WHERE au_fname = 'Innes';
```

will not match

```
SELECT a.au_id, ta.title_id FROM authors a
JOIN titleauthor ta ON a.au_id = ta.au_id
WHERE au_fname = 'Johnson';
```

Note that it also will not match this text

```
SELECT a.au_id, ta.title_id
FROM authors a JOIN titleauthor ta ON a.au_id = ta.au_id
WHERE au_fname = 'Innes';
```

This is because the line feed characters are in different places in the statement. To help with query plan reuse, the SQL Server query processor can perform a process known as autoparameterization. Autoparameterization will change a statement like

```
SELECT * FROM authors WHERE au_fname = 'Innes'
```

To a parameterized statement and a parameter declaration:

```
(@0 varchar(8000))SELECT * FROM authors WHERE au_fname = @0
```

These statements can be observed in the plan cache, using either sys.dm_exec_query_stats or sys.dm_exec_cache_plans, with a CROSS APPLY using sys.dm_exec_sql_text(handle) to correlate the text with the other information. Autoparameterization assists in query plan reuse, but it's not perfect.

For example, the statement

```
SELECT * FROM titles WHERE price > 9.99
```

is parameterized to

```
(@0 decimal(3,2))SELECT * FROM titles WHERE price > @0
```

while

```
SELECT * FROM titles WHERE price > 19.99
```

is parameterized to use a different data type

```
(@0 decimal(4,2))SELECT * FROM titles WHERE price > @0
```

and

```
SELECT * FROM titles WHERE price > $19.99
```

is parameterized to

```
(@0 money)SELECT * FROM titles WHERE price > @0
```

Having multiple query plans for similar queries that could use the same plan is known as plan cache pollution. Not only does it fill up the plan caches with redundant plans, but it also causes time (and CPU and I/O) to be consumed creating the redundant plans. Notice that in autoparameterization, the query processor must "guess" the parameter type based on the parameter value. Autoparameterization helps, but it does not completely eliminate plan cache pollution. In addition, the text of parameterized queries is normalized so that plans are reused even if the original text uses different formatting. Autoparameterization is used only for a subset of queries, based on the complexity of the query. Although a complete discussion of all the autoparameterization rules is beyond the scope of this article, realize that SQL Server uses one of two sets of rules: SIMPLE and FORCED parameterization. An example of the difference is that simple parameterization will not autoparameterize a multitable query, but forced parameterization will.

Send your questions and comments for Bob to mmdbdev@microsoft.com.

Stored Procedures and Parameterized Queries

A much better choice is to use parameterized queries or stored procedures. Not only do these help with query plan reuse, but, if you define your parameters properly, data type guessing is never done. Using stored procedures is the best choice, because the parameter data type is specified exactly in the stored procedure definition. Bear in mind that stored procedures are not perfect either. One difficulty is that database object name resolution is not done at CREATE PROCEDURE time; a table or column name that does not exist causes an execution-time error. Also, although a stored procedure's parameters constitute a "contract" between application code and procedure code, stored procedures can also return resultsets. No metadata definition, and therefore no contract, exists on the number of resultsets and the number and data types of resultset columns.

Stored procedures can be called in at least two ways in database API code. Here's an example using ADO.NET:

```
SqlCommand cmd = new SqlCommand("EXECUTE myproc 100", conn);
int i = cmd.ExecuteNonQuery();
```

Or:

```
SqlCommand cmd = new SqlCommand("myproc", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add("@a", SqlDbType.Int); cmd.Parameters.Value = 100;
int i = cmd.ExecuteNonQuery();
```

Executing a stored procedure as a command string (CommandType.Text) without using ADO.NET's ParameterCollection uses a SQL Server language statement, while using CommandType.StoredProcedure results in a lower-overhead remote procedure call (RPC). This difference can be observed in SQL Profiler. How you pass your parameters is also important because of when the query plans are created, but I'll get back to that later.

Parameterized queries use parameters in APIs the way stored procedures do, with a few important exceptions. ADO.NET's SqlParameter class contains properties not only for parameter name and value, but also for parameter data type, length, precision, and scale. It's important to avoid plan cache pollution by specifying the correct values for all relevant parameters in parameterized queries. Otherwise, because there isn't a parameter contract as there is with stored procedures, ADO.NET must guess at these properties. This is similar to the way that autoparameterization guesses, but the implementation is different in a few areas. The charts below, **Figure 1** and **Figure 2**, show the current implementation in SQL Server 2008 of autoparameterization and ADO.NET 3.5 SP1's SqlParameterCollection's AddWithValue method.

When you're using parameterized queries, it's a bad idea to use Parameters.AddWithValue. In this case, ADO.NET must guess the data type, and there's a special hazard when using strings and

Figure 1 **Parameter Data Types Produced by Autoparameterization**

Literal Type	Parameter Produced
Non-Unicode String	VARCHAR(8000)
Unicode String	NVARCHAR(4000)
Whole Number	Smallest fit: TINYINT, SMALLINT, INT, or BIGINT
Fractional Number	DECIMAL(p,s) with precision and scale matching the literal
Number with Currency Symbol	MONEY

Figure 2 **Parameter Data Types Produced by ADO.NET's AddWithValue and Parameterized Queries**

Literal Type	Parameter Produced
String	NVARCHAR(x) where x is the length of the string
Whole Number	Smallest fit: INT or BIGINT
Fractional Number	FLOAT(53) Note: this is double-precision floating point

AddWithValue. First of all, the .NET string class is a Unicode string, whereas in T-SQL, a string constant can either be specified as Unicode or non-Unicode. ADO.NET will pass in a Unicode string parameter (NVARCHAR in SQL). This can have negative repercussions on the query plan itself, if the column that's being used in the predicate is non-Unicode. For example, suppose you have a table with a VARCHAR column as the clustered primary key:

```
CREATE TABLE sample (
  thekey varchar(7) primary key,
  name varchar(20) -- other columns omitted
)
```

In my ADO.NET code, I want to do a lookup by primary key:

```
cmd.CommandText = "SELECT * FROM sample when thekey = @keyvalue;"
```

And I specify the parameter using this:

```
cmd.Parameters.AddWithValue("@keyvalue", "ABCDEFG");
```

ADO.NET will decide on a parameter data type of NVARCHAR(7). Because the conversion from NVARCHAR to VARCHAR happens in the query execution step that retrieves the rows, the parameter value cannot be used as a search argument. Rather than perform a Clustered Index Seek of one row, the query will perform a Clustered Index Scan of the entire table. Now, imagine this with a table with 5 million rows. Since you've submitted a parameterized query, there's nothing that SQL Server autoparameterization can do, and nothing that a database administrator can change in the server to fix this behavior. Using the FORCESEEK query hint as a last resort fails to produce a plan at all. When the parameter type is specified as SqlDbType.VarChar rather than making ADO.NET guess the data type, the response of such a query drops from multiple seconds (at best) to milliseconds.

Parameter Length

Another good habit to get into for string-based data types is to always specify the length of the parameter. This value should be the length of the field in the SQL predicate that uses the parameter, or the maximum string length (4,000 for NVARCHAR, 8,000 for VARCHAR), not the length of the string itself. SQL Server autoparameterization always assumes the maximum string length, but SqlParameterCollection.AddWithValue makes the parameter length equal to the length of the string. So, using the following calls produces different parameter data types and therefore different plans:

```
// produces an nvarchar(5) parameter
cmd.Parameters.AddWithValue(
"SELECT * FROM authors WHERE au_fname = @name", "@name", "Innes");
// produces an nvarchar(7) parameter
cmd.Parameters.AddWithValue(
"SELECT * FROM authors WHERE au_fname = @name", "@name", "Johnson");
```

By not specifying the length when you're using ParameterCollection.AddWithValue, you can have as many different queries in the plan cache as you have different string lengths. Now that's plan cache pollution in a big way. Although I mention ADO.NET in conjunction with this behavior, note that other database APIs share the problem of string parameter plan cache pollution. The current versions of both LINQ to SQL and ADO.NET Entity Framework exhibit a variant of this behavior. With vanilla ADO.NET, you have the option of specifying a string parameter's length; with the frameworks, the conversion to API calls is done by LINQ to SQL or Entity Framework itself, so you can't do anything about their string parameter plan cache pollution. Both LINQ to SQL and Entity Framework will address this problem in the upcoming .NET Framework 4release. So if you're using your own parameterized queries, don't forget to specify the proper SqlDbType, the length of string parameters, and the precision and scale of decimal parameters. Performance here is absolutely the realm of the programmer, and most DBAs won't check your ADO.NET code if they're concerned about performance. If you're using stored procedures, the explicit parameter contract will ensure that you always use the correct parameter type and length.

Although you should always use parameterized SQL statements inside and output stored procedures if possible, there are a few cases when parameterization cannot be used. You cannot parameterize the names of columns or names of tables in your SQL statements. This includes DDL (Data Definition Language statements) as well as DML (Data Manipulation Language statements). So although parameterization helps performance and is the best safeguard against SQL injection (using string concatenation rather than parameters can allow nefarious users to inject addition SQL into your code), it's not always possible to parameterize everything.

Where you set the value of your parameters is also significant. If you've used SQL Profiler to observe the SQL generated by ADO.NET when you use parameterized queries, you'll notice that it doesn't look like this:

```
(@0 VARCHAR(40))SELECT * FROM authors WHERE au_fname = @0
```

Instead you'll see:

```
sp_executesql N'SELECT * FROM authors WHERE au_fname = @name',
           N'@name VARCHAR(40)', 'Innes'
```

The procedure sp_executesql is a system stored procedure that executes a dynamically built SQL string that can include parameters. One reason why ADO.NET uses it to execute a parameterized query is that this results in use of the lower-overhead RPC call. Another important reason why sp_executesql is used is to enable a SQL Server query processor behavior known as "parameter sniffing." This results in the best performance, because the query processing knows the parameter values at plan-generation time and can make the best use of its statistics.

SQL Server Statistics

SQL Server uses statistics to help generate the best query plan for the job. The two main types of statistics are density statistics (how many unique values exist for a specify column) and cardinality statistics (a histogram of value distribution.) For more information about these statistics, reference the white paper "Statistics Used by the Query Optimizer in Microsoft SQL Server 2005," by Eric N. Hanson and Lubor Kollar (technet.microsoft.com/en-us/library/cc966419.aspx). The key to understanding how SQL Server uses statistics is knowing that SQL Server creates query plans for all queries in a batch or stored procedure at the beginning of the batch or stored procedure. If the query processor knows the value of a parameter at plan-generation time, then the cardinality and density statistics can be used. If the value is unknown at plan-creation time, then only the density

You cannot parameterize the names of columns or names of tables in your SQL statements.

statistics can be used. For example, if the ADO.NET programmer uses parameters like the following, there's no way for the query processor to know you're looking for authors from California and use the cardinality statistics on the state column:

```
cmd.CommandText = "declare @a char(2); set @a = 'CA'; select * from
  authors where state = @a ";
SqlDataReader rdr = cmd.ExecuteReader();
```

The plan is created before the first statement (before the DECLARE statement in this case), when the parameter value hasn't been assigned yet. That's why a parameterized query is translated into sp_executesql. In this case, the plan is created on entry to the sp_executesql stored procedure, and the query processor can sniff the value of the parameter at plan-generation time. The parameter values are specified in the call to sp_executesql. The same concept applies if you write a stored procedure. Say you have a query that retrieves the authors from California if the value passed in is NULL, otherwise the user must specify the state he wants, as follows:

```
CREATE PROCEDURE get_authors_by_state (@state CHAR(2))
AS
BEGIN
IF @state IS NULL THEN @state = 'CA';
SELECT * FROM authors WHERE state = @state;
END
```

Now, in the most common case (no parameter is specified and the state is NULL), the query is optimized for the value NULL, not the value "CA." If CA is a common value of the column, then you'll be potentially getting the wrong plan. So, when you're using parameterized queries in ADO.NET, remember that this means using the SqlParameterCollection, and not specifying the parameter declaration and assignment in the SQL statement itself. If you're writing stored procedures, make sure you keep in mind that setting parameters in the code itself works against parameter sniffing. Note that you won't see different query plans in the example above that uses the authors table in the pubs database; it's too small. In larger tables, this can affect the physical JOIN type that is used and affect other parts of the query plan indirectly. For examples of how parameter sniffing can affect query plans, refer to the white paper "Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005," by Arun Marathe and Shu Scott (technet.microsoft.com/en-us/library/cc966425.aspx).

Parameter sniffing is usually a good idea. The cardinality statistics are calculated to contain an approximately equal number of rows in each histogram step, so that any parameter value is representative of the cardinality as a whole. But, because there is a limited number of cardinality buckets (200 bucket maximum) and because some columns consist of mostly repeating values, this isn't always the case. Imagine that you have a table of customers. Because your business is based in California, most of your customers come from California. Let's imagine 200,000 customers from California and 10 customers from Oregon. A query plan joining your customers table with five other tables might be very different when you're looking for customers in California as opposed to customers in Oregon. If the first query is for customers in Oregon, your cached and reused plan for a California customer would also assume 10 California customers as opposed to the large number of California customers. In this case, using the cardinality statistics isn't a help, but a hindrance. The easiest (but most fragile) way out of this dilemma is to use conditional code—either in the application or in a separate stored procedure, to call two different stored procedures, one for states with many customers and one for states with few customers. SQL Server will not share query plans, even for exactly the same query, if the query occurs in two different stored procedures. The fragile part is determining what constitutes "a state with many customers," and realizing that your distribution of customers can change over time. SQL Server also provides some query hints that can help out. If you decide that having everyone use the plan for California customers is OK (because you have only a small number of rows to process in other states anyway), then you can use the query hint OPTION (OPTIMIZE FOR parameter_name=value). That ensures the plan in cache will always be the plan for states with many customers. As an alternative, you can use SQL Server 2008's new OPTION (OPTIMIZE FOR UNKNOWN) hint, which makes SQL Server ignore cardinality statistics and come up with an intermediate plan, perhaps not optimized for either a big or small state. In addition, if you have a query that uses many parameters, but only uses a value for them one at a time (imagine a system where someone can search from one to ten different conditions defined by parameters in the same query,) then your best bet might be to produce a different query plan each time on purpose. This is specified with an OPTION RECOMPILE query hint.

Right Plan for the Job

To summarize, using parameters guards against plan cache pollution and SQL injection. Always use parameterized queries or parameterized stored procedures. Always specifying the right data type, length, precision, and scale will ensure that you're not doing data type coercion at execution time. Making the values available at query plan creation time ensures that the optimizer can have access to all the statistics that it needs. And if parameter sniffing is a problem (too much caching,) don't go back to a plan for every query that pollutes the cache. Instead, use query hints or stored procedures to ensure that you get the right plan for the job. Remember that the data access and stored procedure code that you, the application programmer, write can make a big difference in performance. ■

BOB BEAUCHEMIN *is a database-centric application practitioner and architect, course author and instructor, writer, and developer skills partner at SQLskills. He's written books and articles on SQL Server, data access and integration technologies, and database security. You can reach him at bobb@sqlskills.com.*

Workflow Design Patterns

Design patterns provide a common, repeatable approach to solving software development tasks, and many different patterns can describe how to accomplish a certain goal in code. When developers begin working with Windows Workflow Foundation (WF), they often ask about how to accomplish common tasks with the technology. This month I discuss several design patterns used in WF.

Doing Work for N Items of Data

Often, workflows are not driven purely by logic but also by data, such as a list of people in an organization or a list of orders, where a set of steps in the workflow needs to execute once for each item. Although perhaps not a pattern in itself, this simple, reusable bit of logic is an important component of the other patterns I discuss in this article. The key to this scenario is using the Replicator activity to iterate over a collection of data and execute the same activities for each item in the collection.

The Replicator activity provides a property for the collection of data items that drives the iterations, events for initializing the child activities for each data item, and conditions to enable you to break out of the execution. Essentially, the Replicator activity provides you with ForEach semantics coupled with DoWhile-style conditional execution.

For example, given a workflow with a property of type List<string> containing employee e-mail addresses, you can iterate over the list and send a message to each employee, as shown in **Figure 1**.

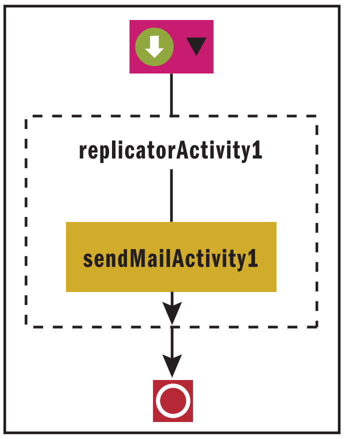


Figure 1 **Replicator with SendMail Activity**

In this scenario, the Replicator activity must have the InitialChildData property bound to a collection implementing the IEnumerable interface that contains the e-mail addresses to be used. These addresses are used to set the recipient's address for each iteration. By handling the ChildInitialized event, you gain access to the data item and the dynamic activity instance that is executed. **Figure 2** shows how the e-mail address from the collection is passed to the event and can be used to set the RecipientAddress property on the related e-mail activity instance.

Figure 2 **Initializing the Child Activity**

```
public List<string> emails = new List<string>
  {"matt@contoso.com","msdnmag@example.com"};

private void InitChildSendMail(object sender, ReplicatorChildEventArgs e)
{
    SendMailActivity sendMail = e.Activity as SendMailActivity;
    sendMail.RecipientAddress = e.InstanceData.ToString();
}
```

The Replicator activity can execute either sequentially or in parallel. In sequential mode, Replicator waits for each iteration to complete before beginning a new iteration. In parallel mode, all activities are initialized and scheduled at the same time, and the execution is much like the Parallel activity, except with the same definition in each branch. Being able to iterate over data items, invoke some actions in parallel, and wait for responses for each item is critical in many design patterns, including several discussed in this article.

Listen with Timeout

In the Listen with Timeout scenario, you have a requirement to wait for some input, but only for a certain amount of time. For example, you might have notified a manager with an e-mail message and need to wait for a reply, but if the manager does not respond within a certain period of time, your workflow should take further action, such as sending a reminder.

The heart of any implementation of this pattern is the Listen activity. The Listen activity allows a workflow to pause and wait for many different events or inputs at the same time. This capability can also be accomplished with the Parallel activity, but the difference is that the Listen activity reacts when the first event occurs and stops listening for all other events, whereas the Parallel activity waits for all events. Combining this functionality with the ability to wait for a designated amount of time, provided by the Delay activity, lets a workflow wait for an event but timeout if the event does not occur. **Figure 3** shows a Listen activity waiting for messages to arrive via Windows Communication Foundation (WCF) or for the timeout to expire. Notice that the Listen activity can have multiple branches and can therefore be listening for many different events at the same time.

Send your questions and comments to mmnet30@microsoft.com.
Code download available at code.msdn.microsoft.com/mag200908Foundations.

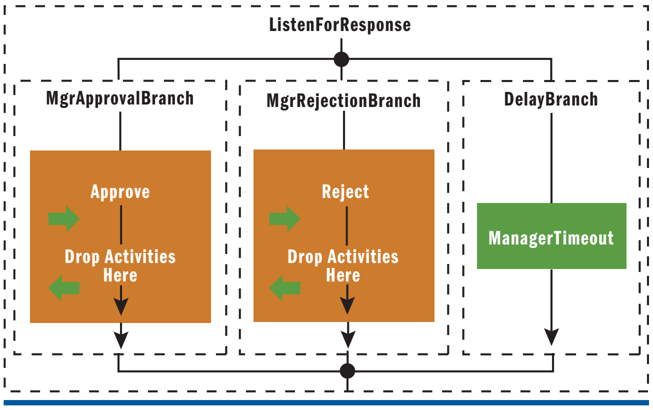


Figure 3 **Listen Activity with Multiple Branches**

An implementation like this enables a workflow to wait for a certain amount of time for a response. Typically, if the timeout occurs, the workflow is designed to take appropriate actions. To expand on the manager approval example, once the timeout occurs, the manager should be reminded that she has an outstanding request she needs to approve. After the manager is reminded, the workflow needs to be restored to a state of waiting for the response and the timeout. Surrounding a Listen with a While activity enables the workflow to continue waiting until a certain condition is met. Inside the branches of the Listen activity, the condition is manipulated appropriately to continue waiting or to move on after the response that is wanted is received. In a simple case, a flag can be used to manage the condition, causing the While activity to loop until the flag is set. Thus, when the manager sends a response, the flag can be set and the While activity closes, allowing the workflow to move on to the next activity. In the branch with the Delay activity, after the delay occurs, activities are used to send a reminder to the manager and to ensure that the condition is still set to force the While activity to schedule the child activities again, as shown in **Figure 4**.

In this example, the trigger condition to stop waiting is simply a response from the manager, but, of course, any level of complex evaluation can be done on the input data received. One option is to use a rule set and the Policy activity to determine whether all conditions have been met to move to the next step in the workflow.

Variation: State Machine

One variation on the Listen with Timeout pattern occurs when you develop a State Machine workflow instead of a Sequential workflow. In this case, the State activity takes the place of the Listen activity and provides the ability to listen for multiple events at the same time, including using the Delay activity. In a given state, say, Waiting For Approval, you can model the same scenario as before, where you wait for a response or the timeout. **Figure 5** shows a sample workflow implementing the same logic as before but using a State Machine workflow.

It is actually simpler to manage the conditions here because there is no need for a While activity. Instead, when the delay occurs, you can send the reminder or take other actions and then transition back to the current state by using the SetState activity, which causes the Delay activity to execute again, resetting the timeout. If the response is received and meets the conditions for continuing, you use the SetState activity to move to the next state. Both branches are shown in **Figure 6**.

Scatter Gather

When you have the need to start many child workflows to do some work, use the Scatter Gather pattern. These workflows might all be doing the same work over different data, or each might be doing different work. The goal is to start all the work, optimize the use of multiple threads to accomplish the tasks faster if possible, and then notify the parent workflow when each task is complete to collect the results.

You can start multiple workflows simply by using the Replicator activity and the InvokeWorkflow activity. The workflows are started

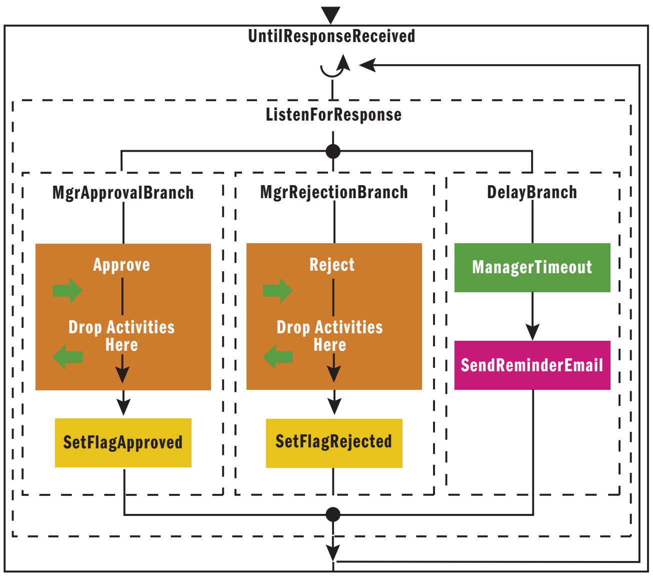


Figure 4 **Listen with While to Send Reminders**

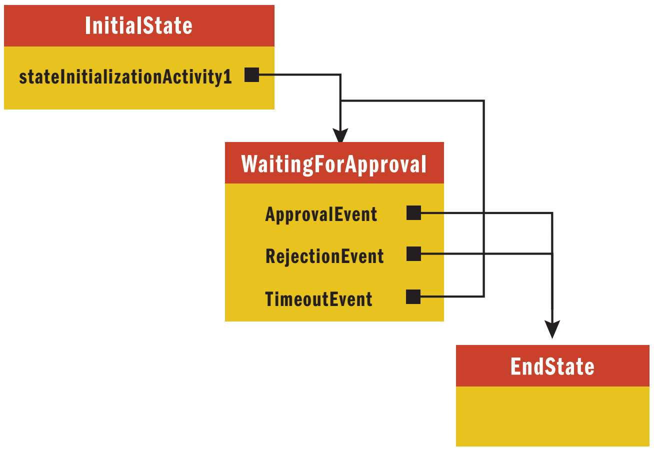


Figure 5 **State Machine Listening**

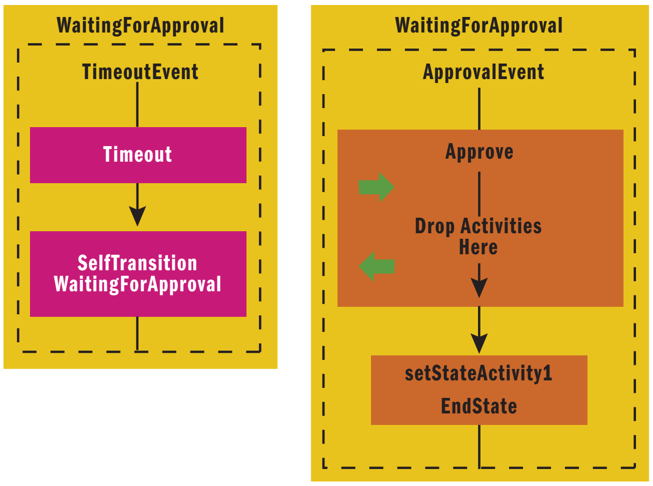


Figure 6 **Listening in a State Activity**

asynchronously, which is what you want, but it makes waiting in the parent workflow more challenging because you need a blocking activity that can receive the data back from the child workflows. Using the Receive activity, the parent workflow can wait for each child activity to finish and receive any results back from each workflow that was started. The high-level view of this pattern in the parent workflow is shown in **Figure 7.**

The figure makes this pattern look simple to implement, but several key steps are needed to ensure that the child workflows can correctly call back to the parent workflow using WCF. Context information needs to be passed from the parent workflow to each child to enable the child to send data back to the parent workflow, including the workflow instance identifier and the conversation

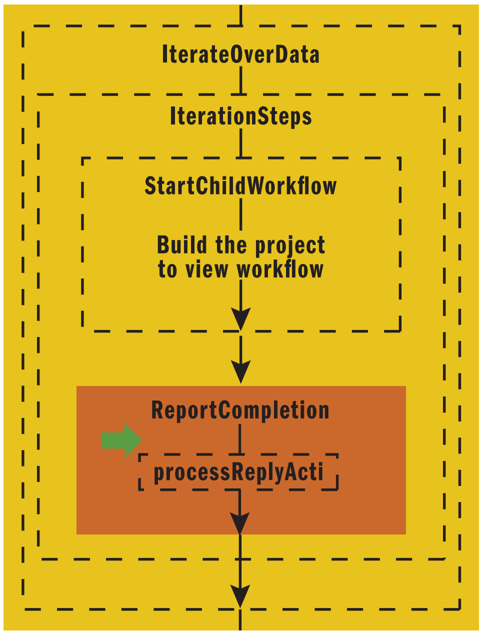


Figure 7 **Replicator with InvokeWorkflow and Receive Activities**

Figure 8 **Starting a Workflow That Must Be Hosted as a WCF Service**

```
WorkflowServiceHost host = new WorkflowServiceHost(typeof(MSDN.Workflows.
  ParentWorkflow));

try
{
    host.Open();
    WorkflowRuntime runtime = host.Description.Behaviors.
      Find<WorkflowRuntimeBehavior>().WorkflowRuntime;
    WorkflowInstance instance = runtime.CreateWorkflow(
      typeof(MSDN.Workflows.ParentWorkflow));
    instance.Start();
    Console.ReadLine();
}
catch (Exception ex)
{
    Console.WriteLine(ex);
    Console.ReadLine();
}
finally
{
    if (host != null && host.State == CommunicationState.Opened)
        host.Close();
    else
        host.Abort();
}
```

identifier to select the correct Receive activity. Additionally, the parent workflow must be hosted as a WCF service to enable the communication, but it needs to be started using WorkflowRuntime, as shown in **Figure 8**.

Each child workflow needs to have input parameters for at least the parent workflow ID and the receive activity ID, in addition to any business data that needs to be processed in the child workflow. Parameters to the workflow are defined as public properties on the workflow definition.

The InvokeWorkflow activity allows you to pass parameters to the child workflow and surfaces those properties in the property dialog box. The parameters on the InvokeWorkflow activity can be bound to a property or to a field in the workflow. However, when using the Replicator activity to invoke many workflows, the parameters need to be set in code because each invocation requires unique values; for each iteration, the property or field can be set with the current inputs. Therefore, the parameters on the InvokeWorkflow activity should be bound to fields in the workflow, and those fields will be updated in your code before the child workflow is created.

Your initial inclination might be to set the property during the ChildInitialized event for the Replicator, as I showed with the SendMail example earlier, and this is a good place to start. However, when executing the Replicator activity in parallel mode, all the children are initialized before any instances begin to execute. Therefore, if you set the property in the ChildInitialized event, by the time the InvokeWorkflow activity executes, all instances of the activity would use a single set of data. However, the ChildInitialized event does provide access to the activity instance and the data item driving the iteration. One approach is to collect the data item and store it with a unique identifier so that it can be related to the correct activity instance during execution. **Figure 9** shows the ChildInitialized event handler for the Replicator activity where the instance data is stored in a dictionary keyed on the unique identifier for the ActivityExecutionContext.

Figure 9 **Storing Data During Iterations**

```
private void InitChild(object sender, ReplicatorChildEventArgs e)
{
    InvokeWorkflowActivity startWF =
      (InvokeWorkflowActivity)e.Activity.GetActivityByName("StartChild
        Workflow");
    InputValueCollection[(Guid)e.Activity.GetValue(
    Activity.ActivityContextGuidProperty)] = e.InstanceData.ToString();
}
```

Next, to initialize the InvokeWorkflow activity, you use the Invoking event to set up the parameters. At this point in the execution, all the values needed for input to the child workflow are available. The workflow identifier can be retrieved from the WorkflowEnvironment, and the conversation identifier can be retrieved from the context property on the Receive activity instance. Finally, the business data can be retrieved using the identifier for the current execution context. **Figure 10** shows the code to initialize the parameter to be passed to the workflow.

After the child workflow is started, it can begin to execute the work to be done and, on completion, use the Send activity to notify the parent workflow. Before sending the message, the context must be set on the Send activity to ensure that the message gets sent to the correct Receive activity in the parent workflow. Using the values passed from the parent, the context can be correctly set using the BeforeSend event, as shown here.

```
e.SendActivity.Context = new Dictionary<string, string>{
                {"instanceId", InputValues.WFInstanceID},
                {"conversationId", InputValues.ConversationID}};
```

With all these parts in place, the parent workflow starts, and the Replicator activity iterates over the collection of data, starting one child workflow for each item and waiting for a message back from each in parallel. Then, as the child workflows finish, they send a message back to the parent, which can continue processing after all the child workflows have reported back their results. Using this approach, the child workflows can be running at the same time, each with its own thread, providing truly asynchronous processing.

Starting Workflow Services with Multiple Operations

In many samples, workflow services start with a single Receive activity modeling a single operation. In the scenario I've discussed here, you have a need to start the workflow service with more than one method call. That is, the client application might not always invoke the same operation to begin interacting with your workflow, and you need to be able to design the workflow so that it can be started on the basis of multiple operations.

There are actually two different varieties of this pattern, depending on how you want to handle requests after the first request. The first option is to enable the workflow to start with one of several operations and, after that operation is complete, move on with the workflow processing until you define another point where operations can be called. To accomplish this goal, you need to return to the Listen activity and use it as the first activity in your workflow definition. Then, in each branch of the activity, add a Receive activity, configure it with the appropriate service operation information, and bind any necessary parameters for processing, as shown in **Figure 11.**

The crucial step is to ensure that all the Receive activities in the Listen activity have the CanCreateInstance property set to True. This instructs the WCF runtime that if no context information is available on the request indicating an existing workflow instance, it is okay to start a new instance based on the configured operation. Although it might seem slightly odd to create the workflow with an activity other than Receive, the runtime creates the instance, then starts, and only then attempts to send the contents of the WCF message to the Receive activity. In this case, once the workflow starts, both Receive activities are executing and waiting for input.

I mentioned that there are two variations of this pattern. When you use the Listen activity as I did in the previous example, one of the operations starts the workflow, but then the Listen activity completes after that single branch is done and the service is no longer able to receive requests for the other operations modeled in the Listen. This

Figure 10 **Initializing the InvokeWorkflow Activity**

```
private void PrepChildParams(object sender, EventArgs e)
{
    InvokeWorkflowActivity startWf = sender as InvokeWorkflowActivity;
    ReceiveActivity receive =
      (ReceiveActivity)startWf.Parent.GetActivityByName(
        "ReceiveChildCompletion");
    Contracts.ChildWFRequest request = new Contracts.ChildWFRequest();
    request.WFInstanceID = WorkflowEnvironment.WorkflowInstanceId.ToString();
    request.ConversationID = receive.Context["conversationId"];
    request.RequestValue =
      InputValueCollection[(Guid)startWf.Parent.GetValue(
    Activity.ActivityContextGuidProperty)];
    StartWF_Input = request;
}
```

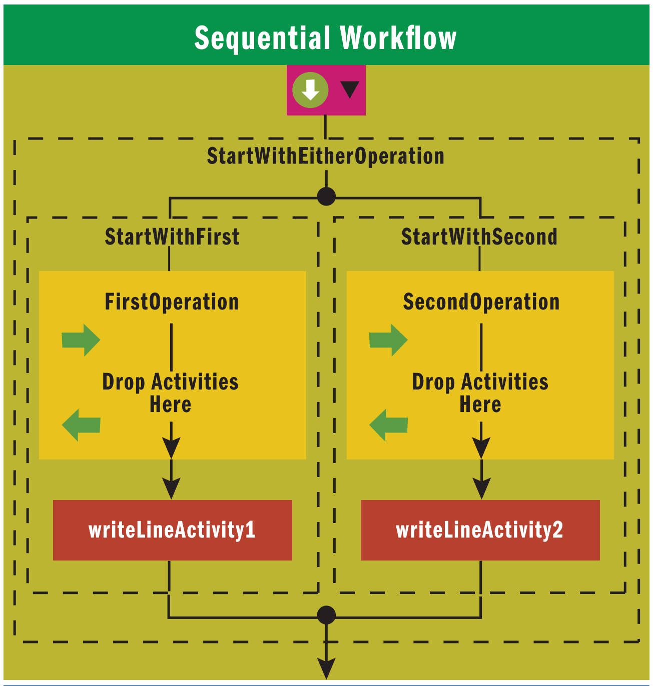


Figure 11 **Multiple Receive Activities in a Listen Activity**

might be exactly what you want in some scenarios, but in others you want the workflow to handle an entire set of operations before it moves on. That is, you know the workflow will receive several requests on different operations in the service contract, but you are not sure which request will be first. In this case, instead of the Listen activity, you can use the Parallel activity with each branch containing a Receive activity with its CanCreateInstance property set to True. This still allows the workflow to start with any operation, but it also keeps the workflow in a state to receive all the other operation calls modeled in the various branches.

Finally, when using a State Machine workflow, you have more flexibility in how the workflow behaves when a particular message is received. Consider a state machine in which the initial state contains several event-driven activities, each with a Receive activity as the starting activity, and each Receive marked to enable creation. Normally, the State activity acts much like the Listen activity, but as the developer, you decide when control moves from the current state to another state. When the Listen activity completes, it closes, and control moves to the next activity in the sequence. With a State activity, after a branch executes, if the workflow does not move to a new state, the workflow remains in the current state and continues to wait for the defined inputs.

To use semantics such as the Listen activity, you must use the SetState activity to move the workflow to the next state when one of the operations is invoked. This usually puts the workflow into a state in which it is waiting for different WCF operations to be invoked. If, on the other hand, you want semantics closer to the Parallel model, where all the operations must be invoked but not in a particular order, then after each Receive activity you have the choice of not changing state or of using the SetState activity to transition back to the same state, a self-transition.

This last option is not entirely like the Parallel activity model in one potentially significant manner. With the Parallel activity, after an operation has been called, it cannot be called again unless modeled somewhere after the Parallel activity. In the state machine model, after the operation is invoked, if the workflow remains in the same state, it can receive messages for the other operations or the original operation. In effect, the state can provide you with semantics similar to a Listen activity in a While activity, waiting for all events, reacting to a single event, and then looping back and waiting for all those same events again. ■

MATT MILNER *is a member of the technical staff at Pluralsight, where he focuses on connected systems technologies (WCF, Windows Workflow, BizTalk, "Dublin," and the Azure Services Platform). Matt is also an independent consultant specializing in Microsoft .NET application design and development. Matt regularly shares his love of technology by speaking at local, regional, and international conferences such as Tech. Ed. Microsoft has recognized Matt as an MVP for his community contributions around connected systems technology. Contact Matt via his blog: pluralsight.com/community/blogs/matt/.*

Aggregating Exceptions

Exceptions in .NET are the fundamental mechanism by which errors and other exceptional conditions are communicated. Exceptions are used not only to store information about such issues but also to propagate that information in object-instance form through a call stack. Based on the Windows structured exception handling (SEH) model, only one .NET exception can be "in flight" at any particular time on any particular thread, a restriction that is usually nary a thought in developers' minds. After all, one operation typically yields only one exception, and thus in the sequential code we write most of the time, we need to be concerned about only one exception at a time. However, there are a variety of scenarios in which multiple exceptions might result from one operation. This includes (but is not limited to) scenarios involving parallelism and concurrency.

Only one .NET exception can be "in flight" at any particular time on any particular thread.

Consider the raising of an event in .NET:

```
public event EventHandler MyEvent;

protected void OnMyEvent() {
    EventHandler handler = MyEvent;
    if (handler != null) handler(this, EventArgs.Empty);
}
```

Multiple delegates can be registered with MyEvent, and when the handler delegate in the previous code snippet is invoked, the operation is equivalent to code like the following:

```
foreach(var d in handler.GetInvocationList()) {
    ((EventHandler)d)(this, EventArgs.Empty);
}
```

Each delegate that makes up the handler multicast delegate is invoked one after the other. However, if any exception is thrown from any of the invocations, the foreach loop ceases processing, which means that some delegates might not be executed in the case of an exception. As an example, consider the following code:

```
MyEvent += (s, e) => Console.WriteLine("1");
MyEvent += (s, e) => Console.WriteLine("2");
MyEvent += (s, e) => { throw new Exception("uh oh"); };
MyEvent += (s, e) => Console.WriteLine("3");
MyEvent += (s, e) => Console.WriteLine("4");
```

If MyEvent is invoked now, "1" and "2" are output to the console, an exception is thrown, and the delegates that would have output "3" and "4" will not be invoked.

To ensure that all the delegates are invoked even in the face of an exception, we can rewrite our OnMyEvent method as follows:

```
protected void OnMyEvent() {
    EventHandler handler = MyEvent;
    if (handler != null) {
        foreach (var d in handler.GetInvocationList()) {
            try {
                ((EventHandler)d)(this, EventArgs.Empty);
            }
            catch{}
        }
    }
}
```

Now we're catching all the exceptions that escape from a registered handler, allowing delegates that come after an exception to still be invoked. If you rerun the earlier example, you'll see that "1", "2", "3", and "4" are output, even though an exception is thrown from one of the delegates. Unfortunately, this new implementation is also eating the exceptions, a practice that is greatly frowned on. Those exceptions could indicate a serious issue with the application, an issue that is not being handled because the exceptions are ignored.

What we really want is to capture any exceptions that might emerge, and then once we've finished invoking all the event handlers, throw again all the exceptions that escaped from a handler. Of course, as already mentioned, only one exception instance can be thrown on a given thread at a given time. Enter AggregateException.

In the .NET Framework 4, System.AggregateException is a new exception type in mscorlib. While a relatively simple type, it enables a plethora of scenarios by providing central and useful exception-related functionality.

AggregateException is itself an exception (deriving from System. Exception) that contains other exceptions. The base System.Exception class already has the notion of wrapping a single Exception instance, referred to as the "inner exception." The inner exception, exposed through the InnerException property on Exception, represents the cause of the exception and is often used by frameworks that layer functionality and that use exceptions to elevate the information being provided. For example, a component that parses input data from a stream might encounter an IOException while reading from the stream. It might then create a CustomParserException that wraps the IOException instance as the InnerException, providing higher-level details about what went wrong in the parse operation while still providing the IOException for the lower-level and underlying details.

Send your questions and comments for Stephen to netqa@microsoft.com.

Figure 1 **System.AggregateException**

```
[Serializable]
[DebuggerDisplay("Count = {InnerExceptions.Count}")]
public class AggregateException : Exception
{
    public AggregateException();
    public AggregateException(params Exception[] innerExceptions);
    public AggregateException(IEnumerable<Exception> innerExceptions);
    public AggregateException(string message);
    public AggregateException(string message, Exception innerException);
    public AggregateException(string message,
        params Exception[] innerExceptions);
    public AggregateException(string message,
        IEnumerable<Exception> innerExceptions);

    public AggregateException Flatten();
    public void Handle(Func<Exception, bool> predicate);

    public ReadOnlyCollection<Exception> InnerExceptions { get; }
}
```

AggregateException simply extends that support to enable wrapping of inner exceptions—plural. It provides constructors that accept params arrays or enumerables of these inner exceptions (in addition to the standard constructor that accepts a single inner exception), and it exposes the inner exceptions through an InnerExceptions property (in addition to the InnerException property from the base class). See **Figure 1** for an overview of AggregateException's public surface area.

If the AggregateException doesn't have any inner exceptions, InnerException returns null and InnerExceptions returns an empty collection. If the AggregateException is provided with a single exception to wrap, InnerException returns that instance (as you'd expect), and InnerExceptions returns a collection with just that one exception. And if the AggregateException is provided with multiple exceptions to wrap, InnerExceptions returns all those in the collection, and InnerException returns the first item from that collection.

Now, with AggregateException, we can augment our .NET event-raising code as shown in **Figure 2**, and we're able to have our cake and eat it, too. Delegates registered with the event continue to run even if one throws an exception, and yet we don't lose any of the exceptional information because they're all wrapped into an aggregate and thrown again at the end (only if any of the delegates fail, of course).

Figure 2 **Using AggregateException when Raising Events**

```
protected void OnMyEvent() {
    EventHandler handler = MyEvent;
    if (handler != null) {
        List<Exception> exceptions = null;
        foreach (var d in handler.GetInvocationList())
        {
            try {
                ((EventHandler)d)(this, EventArgs.Empty);
            }
            catch (Exception exc) {
                if (exceptions == null)
                    exceptions = new List<Exception>();
                exceptions.Add(exc);
            }
        }
        if (exceptions != null) throw new AggregateException(exceptions);
    }
}
```

Events provide a solid example of where exception aggregation is useful for sequential code. However, AggregateException is also of prime importance for the new parallelism constructs in .NET 4 (and, in fact, even though AggregateException is useful for non-parallel code, the type was created and added to the .NET Framework by the Parallel Computing Platform team at Microsoft).

Consider the new Parallel.For method in .NET 4, which is used for parallelizing a for loop. In a typical for loop, only one iteration of that loop can execute at any one time, which means that only one exception can occur at a time. With a parallel "loop," however, multiple iterations can execute in parallel, and multiple iterations can throw exceptions concurrently. A single thread calls the Parallel.For method, which can logically throw multiple exceptions, and thus we need a mechanism through which those multiple exceptions can be propagated onto a single thread of execution. Parallel.For handles this by gathering the exceptions thrown and propagating them wrapped in an AggregateException. The rest of the methods on Parallel (Parallel.ForEach and Parallel.Invoke) handle things similarly, as does Parallel LINQ (PLINQ), also part of .NET 4. In a LINQ-to-Objects query, only one user delegate is invoked at a time, but with PLINQ, multiple user delegates can be invoked in parallel, those delegates might throw exceptions, and PLINQ deals with that by gathering them into an AggregateException and propagating that aggregate.

As an example of this kind of parallel execution, consider **Figure 3**, which shows a method that uses the ThreadPool to invoke multiple user-provided Action delegates in parallel. (A more robust and scalable implementation of this functionality exists in .NET 4 on the Parallel class.) The code uses QueueUserWorkItem to run each Action. If the Action delegate throws an exception, rather than allowing that

Figure 3 **AggregateException in Parallel Invocation**

```
public static void ParallelInvoke(params Action[] actions)
{
    if (actions == null) throw new ArgumentNullException("actions");
    if (actions.Any(a => a == null)) throw new ArgumentException
      ("actions");
    if (actions.Length == 0) return;

    using (ManualResetEvent mre = new ManualResetEvent(false)) {
        int remaining = actions.Length;
        var exceptions = new List<Exception>();
        foreach (var action in actions) {
            ThreadPool.QueueUserWorkItem(state => {
                try {
                    ((Action)state)();
                }
                catch (Exception exc) {
                    lock (exceptions) exceptions.Add(exc);
                }
                finally {
                   if (Interlocked.Decrement(ref remaining) == 0) mre.Set();
                }
            }, action);
        }
        mre.WaitOne();
        if (exceptions.Count > 0) throw new AggregateException
          (exceptions);
    }
}
```

exception to propagate and go unhandled (which, by default, results in the process being torn down), the code captures the exception and stores it in a list shared by all the work items. After all the asynchronous invocations have completed (successfully or exceptionally), an AggregateException is thrown with the captured exceptions, if any were captured. (Note that this code could be used in OnMyEvent to run all delegates registered with an event in parallel.)

The new System.Threading.Tasks namespace in .NET 4 also makes liberal use of AggregateExceptions. A Task in .NET 4 is an object that represents an asynchronous operation. Unlike QueueUserWorkItem, which doesn't provide any mechanism to refer back to the queued work, Tasks provides a handle to the asynchronous work, enabling a large number of important operations to be performed, such as waiting for a work item to complete or continuing from it to perform some operation when the work completes. The Parallel methods mentioned earlier are built on top of Tasks, as is PLINQ.

Methods in .NET that need to deal with the potential for multiple exceptions always wrap, even if only one exception occurs.

Furthering the discussion of AggregateException, an easy construct to reason about here is the static Task.WaitAll method. You pass to WaitAll all the Task instances you want to wait on, and WaitAll "blocks" until those Task instances have completed. (I've placed quotation marks around "blocks" because the WaitAll method might actually assist in executing the Tasks so as to minimize resource consumption and provide better efficiency than just blocking a thread.) If the Tasks all complete successfully, the code goes on its merry way. However, multiple Tasks might have thrown exceptions, and WaitAll can propagate only one exception to its calling thread, so it wraps the exceptions into a single AggregateException and throws that aggregate.

Tasks use AggregateExceptions in other places as well. One that might not be as obvious is in parent/child relationships between Tasks. By default, Tasks created during the execution of a Task are parented to that Task, providing a form of structured parallelism. For example, Task A creates Task B and Task C, and in doing so Task A is considered the parent of both Task B and Task C. These relationships come into play primarily in regard to lifetimes. A Task isn't considered completed until all its children have completed, so if you used Wait on Task A, that instance of Wait wouldn't return until both B and C had also completed. These parent/child relationships not only affect execution in that regard, but they're also visible through new debugger tool windows in Visual Studio 2010, greatly simplifying the debugging of certain types of workloads.

Consider code like the following:

```
var a = Task.Factory.StartNew(() => {
    var b = Task.Factory.StartNew(() => {
        throw new Exception("uh");
    });
    var c = Task.Factory.StartNew(() => {
        throw new Exception("oh");
    });
});
...
a.Wait();
```

Here, Task A has two children, which it implicitly waits for before it is considered complete, and both of those children throw unhandled exceptions. To account for this, Task A wraps its children's exceptions into an AggregateException, and it's that aggregate that's returned from A's Exception property and thrown out of a call to Wait on A.

As I've demonstrated, AggregateException can be a very useful tool. For usability and consistency reasons, however, it can also lead to designs that might at first be counterintuitive. To clarify what I mean, consider the following function:

```
public void DoStuff()
{
    var inputNum = Int32.Parse(Console.ReadLine());
    Parallel.For(0, 4, i=> {
        if (i < inputNum) throw new MySpecialException(i.ToString());
    });
}
```

Here, depending on user input, the code contained in the parallel loop might throw 0, 1, or more exceptions. Now consider the code you'd have to write to handle those exceptions. If Parallel.For wrapped exceptions in an AggregateException only when multiple exceptions were thrown, you, as the consumer of DoStuff, would need to write two separate catch handlers: one for the case in which only one MySpecialException occurred, and one for the case in which an AggregateException occurred. The code for handling the AggregateException would likely search the AggregateException's InnerExceptions for a MySpecialException and then run the same handling code for that individual exception that you would have in the catch block dedicated to MySpecialException. As you start dealing with more exceptions, this duplication problem grows. To address this problem as well as to provide consistency, methods in .NET 4 like Parallel.For that need to deal with the potential for multiple exceptions always wrap, even if only one exception occurs. That way, you need to write only one catch block for AggregateException. The exception to this rule is that exceptions that may never occur in a concurrent scope will not be wrapped. So, for example, exceptions that might result from Parallel.For due to it validating its arguments and finding one of them to be null will not be wrapped. That argument validation occurs before Parallel.For spins off any asynchronous work, and thus it's impossible that multiple exceptions could occur.

Of course, having exceptions wrapped in an AggregateException can also lead to some difficulties in that you now have two models to deal with: unwrapped and wrapped exceptions. To ease the transition between the two, AggregateException provides several helper methods to make working with these models easier.

The first helper method is Flatten. As I mentioned, AggregateException is itself an Exception, so it can be thrown. This means, however, that AggregateException instances can wrap other AggregateException instances, and, in fact, this is a likely occurrence, especially when dealing with recursive functions that might throw aggregates. By default, AggregateExceptions retains this hierarchical structure, which can be helpful when debugging because the hierarchical structure of the contained aggregates will likely correspond to the structure of the code that threw those exceptions. However, this can also make aggregates more difficult to work with in some cases. To account for that, the Flatten method removes the layers of contained aggregates by creating a new AggregateException that contains the non-AggregateExceptions from the whole hierarchy. As an example, let's say I had the following structure of exception instances:

- AggregateException
  - InvalidOperationException
  - ArgumentOutOfRangeException
  - AggregateException
    - IOException
    - DivideByZeroException
    - AggregateException
      - FormatException
  - AggregateException
    - TimeZoneException

If I call Flatten on the outer AggregateException instance, I get a new AggregateException with the following structure:

- AggregateException
  - InvalidOperationException
  - ArgumentOutOfRangeException
  - IOException
  - DivideByZeroException
  - FormatException
  - TimeZoneException

At its core, an AggregateException is really just a container for other exceptions.

This makes it much easier for me to loop through and examine the InnerExceptions of the aggregate, without having to worry about recursively traversing contained aggregates.

The second helper method, Handle, makes such traversal easier. Handle has the following signature:

```
public void Handle(Func<Exception,bool> predicate);
```

Here's an approximation of its implementation:

```
public void Handle(Func<Exception,bool> predicate)
{
    if (predicate == null) throw new ArgumentNullException("predicate");
    List<Exception> remaining = null;
    foreach(var exception in InnerExceptions) {
        if (!predicate(exception)) {
            if (remaining == null) remaining = new List<Exception>();
            remaining.Add(exception);
        }
    }
    if (remaining != null) throw new AggregateException(remaining);
}
```

Handle iterates through the InnerExceptions in the AggregateException and evaluates a predicate function for each. If the predicate function returns true for a given exception instance, that exception is considered handled. If, however, the predicate returns false, that exception is thrown out of Handle again as part of a new AggregateException containing all the exceptions that failed to match the predicate. This approach can be used to quickly filter out exceptions you don't care about; for example:

```
try {
    MyOperation();
}
catch(AggregateException ae) {
    ae.Handle(e => e is FormatException);
}
```

That call to Handle filters out any FormatExceptions from the AggregateException that is caught. If there are exceptions besides FormatExceptions, only those exceptions are thrown again as part of the new AggregateException, and if there aren't any non-FormatException exceptions, Handle returns successfully with nothing being thrown again. In some cases, it might also be useful to first flatten the aggregates, as you see here:

```
ae.Flatten().Handle(e => e is FormatException);
```

Of course, at its core an AggregateException is really just a container for other exceptions, and you can write your own helper methods to work with those contained exceptions in a manner that fits your application's needs. For example, maybe you care more about just throwing a single exception than retaining all the exceptions. You could write an extension method like the following:

```
public static void PropagateOne(this AggregateException aggregate)
{
    if (aggregate == null) throw new ArgumentNullException("aggregate");
    if (aggregate.InnerException != null)
        throw aggregate.InnerException; // just throw one
}
```

which you could then use as follows:

```
catch(AggregateException ae) { ae.PropagateOne(); }
```

Or maybe you want to filter to show only those exceptions that match a certain criteria and then aggregate information about those exceptions. For example, you might have an AggregateException containing a whole bunch of ArgumentExceptions, and you want to summarize which parameters caused the problems:

```
AggregateException aggregate = ...;
string [] problemParameters =
    (from exc in aggregate.InnerExceptions
     let argExc = exc as ArgumentException
     where argExc != null && argExc.ParamName != null
     select argExc.ParamName).ToArray();
```

All in all, the new System.AggregateException is a simple but powerful tool, especially for applications that can't afford to let any exception go unnoticed. For debugging purposes, AggregateException's ToString implementation outputs a string rendering all the contained exceptions. And as you can see back in **Figure 1**, there's even a DebuggerDisplayAttribute on AggregateException to help you quickly identify how many exceptions an AggregateException contains. ■

STEPHEN TOUB *is a Senior Program Manager Lead on the Parallel Computing Platform team at Microsoft. He is also a Contributing Editor for* MSDN Magazine.