Forewords by Miguel de Icaza
and Anders Hejlsberg

Second Edition

Framework Design Guidelines

Conventions, Idioms, and Patterns for Reusable .NET Libraries

Microsoft .net Development Series

Krzysztof Cwalina
Brad Abrams

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Praise for *Framework Design Guidelines*

"*Framework Design Guidelines* is one of those rare books that can be read at different reading levels and can be useful to different kinds of developers. Regardless of whether you want to design an effective object model, improve your understanding of the .NET Framework, borrow from the experience of software gurus, stay clear of the most common programming mistakes, or just get an idea of the huge effort that led to the .NET initiative, this book is a must-read."

—Francesco Balena, The VB Migration Partner Team (www.vbmigration.com),
Code Architect, Author, and Microsoft Regional Director, Italy

"Frameworks are valuable but notoriously difficult to construct: your every decision must be geared toward making them easy to be used correctly and difficult to be used incorrectly. This book takes you through a progression of recommendations that will eliminate many of those downstream 'I wish I'd known that earlier' moments. I wish *I'd* read it earlier."

—Paul Besly, Principal Technologist, QA

"Not since Brooks' *The Mythical Man Month* has the major software maker of its time produced a book so full of relevant advice for the modern software developer. This book has a permanent place on my bookshelf and I consult it frequently."

—George Byrkit, Senior Software Engineer, Genomic Solutions

"Updated for the new language features of the .NET Framework 3.0 and 3.5, this book continues to be the definitive resource for .NET developers and architects who are designing class library frameworks. Some of the existing guidelines have been expanded with new annotations and more detail, and new guidance covering such features as extension methods and nullable types has also been included. The guidance will help any developer write clearer and more understandable code, while the annotations provide invaluable insight into some of the design decisions that made the .NET Framework what it is today."

—Scott Dorman, Microsoft MVP and President,
Tampa Bay International Association of Software Architects

"Filled with information useful to developers and architects of all levels, this book provides practical guidelines and expert background information to get behind the rules. *Framework Design Guidelines* takes the already published guidelines to a higher level, and it is needed to write applications that integrate well in the .NET area."

—Cristof Falk, Software Engineer

"This book is an absolute must read for all .NET developers. It gives clear 'do' and 'don't' guidance on how to design class libraries for .NET. It also offers insight into the design and creation of .NET that really helps developers understand the reasons why things are the way they are. This information will aid developers designing their own class libraries and will also allow them to take advantage of the .NET class library more effectively."

—Jeffrey Richter, Author/Trainer/Consultant, Wintellect

"The second edition of *Framework Design Guidelines* gives you new, important insight into designing your own class libraries: Abrams and Cwalina frankly discuss the challenges of adding new features to shipping versions of their products with minimal impact on existing code. You'll find great examples of how to create version N+1 of your software by learning how the .NET class library team created versions 2.0, 3.0, and 3.5 of the .NET library. They were able to add generics, WCF, WPF, WF, and LINQ with minimal impact on the existing APIs, even providing capabilities for customers wanting to use only some of the new features, while still maintaining compatibility with the original library."

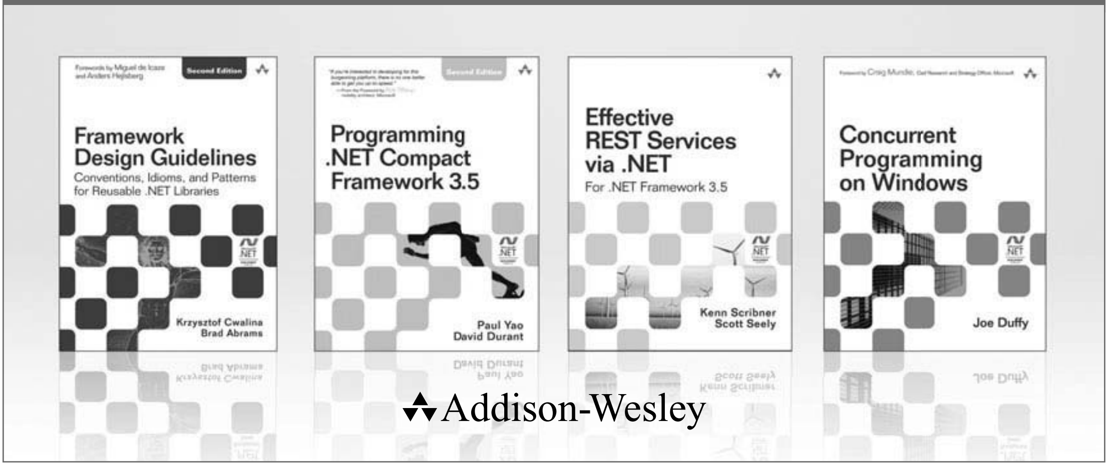
—Bill Wagner, Founder and Consultant, SRT Solutions,
author of *Effective C#* and *More Effective C#*

"This book is a must read for all architects and software developers thinking about frameworks. The book offers insight into some driving factors behind the design of the .NET Framework. It should be considered mandatory reading for anybody tasked with creating application frameworks."

—Peter Winkler, Sr. Software Engineer, Balance Technology Inc.

Framework Design Guidelines
Second Edition

Microsoft® .NET Development Series



Visit **informit.com/msdotnetseries** for a complete list of available products.

The award-winning **Microsoft .NET Development Series** was established in 2002 to provide professional developers with the most comprehensive, practical coverage of the latest .NET technologies. Authors in this series include Microsoft architects, MVPs, and other experts and leaders in the field of Microsoft development technologies. Each book provides developers with the vital information and critical insight they need to write highly effective applications.

Framework Design Guidelines

Conventions, Idioms, and Patterns for Reusable .NET Libraries

Second Edition

■ Krzysztof Cwalina
■ Brad Abrams

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Cwalina, Krzysztof.
Framework design guidelines : conventions, idioms, and patterns for reusable .NET libraries / Krzysztof Cwalina, Brad Abrams. — 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-54561-9 (hardcover : alk. paper)
1. Microsoft .NET Framework. 2. Application program interfaces (Computer software) I. Abrams, Brad. II. Title.

QA76.76.M52C87 2008
006.7'882—dc22

2008034905

ISBN-13: 978-0-321-54561-9
ISBN-10: 0-321-54561-3

Text printed in the United States on recycled paper at Donnelley in Crawfordsville, Indiana.
Third printing, December 2009

To my wife, Ela,
for her support throughout the long process of writing this book,
and to my parents,
Jadwiga and Janusz, for their encouragement.
—Krzysztof Cwalina

To my wife, Tamara:
Your love and patience strengthen me.
—Brad Abrams

This page intentionally left blank

Contents

This page intentionally left blank

Figures

Tables

This page intentionally left blank

Foreword

When the .NET Framework was first published, I was fascinated by the technology. The benefits of the CLR (Common Language Runtime), its extensive APIs, and the C# language were immediately obvious. But underneath all the technology were a common design for the APIs and a set of conventions that were used everywhere. This was the .NET culture. Once you had learned a part of it, it was easy to translate this knowledge into other areas of the Framework.

For the past 16 years, I have been working on open source software. Since contributors span not only multiple backgrounds but multiple years, adhering to the same style and coding conventions has always been very important. Maintainers routinely rewrite or adapt contributions to software to ensure that code adheres to project coding standards and style. It is always better when contributors and people who join a software project follow conventions used in an existing project. The more information that can be conveyed through practices and standards, the simpler it becomes for future contributors to get up-to-speed on a project. This helps the project converge code, both old and new.

As both the .NET Framework and its developer community have grown, new practices, patterns, and conventions have been identified. Brad and Krzysztof have become the curators who turned all of this new knowledge into the present-day guidelines. They typically blog about a new convention, solicit feedback from the community, and keep track of these guidelines. In my opinion, their blogs are must-read documents for everyone who is interested in getting the most out of the .NET Framework.

The first edition of *Framework Design Guidelines* became an instant classic in the Mono community for two valuable reasons. First, it provided us a means of understanding why and how the various .NET APIs had been implemented. Second, we appreciated it for its invaluable guidelines that we too strived to follow in our own programs and libraries. This new edition not only builds on the success of the first but has been updated with new lessons that have since been learned. The annotations to the guidelines are provided by some of the lead .NET architects and great programmers who have helped shape these conventions.

In conclusion, this text goes beyond guidelines. It is a book that you will cherish as the "classic" that helped you become a better programmer, and there are only a select few of those in our industry.

Miguel de Icaza
Boston, MA

Foreword to the First Edition

In the early days of development of the .NET Framework, before it was even called that, I spent countless hours with members of the development teams reviewing designs to ensure that the final result would be a coherent platform. I have always felt that a key characteristic of a framework must be consistency. Once you understand one piece of the framework, the other pieces should be immediately familiar.

As you might expect from a large team of smart people, we had many differences of opinion—there is nothing like coding conventions to spark lively and heated debates. However, in the name of consistency, we gradually worked out our differences and codified the result into a common set of guidelines that allow programmers to understand and use the Framework easily.

Brad Abrams, and later Krzysztof Cwalina, helped capture these guidelines in a living document that has been continuously updated and refined during the past six years. The book you are holding is the result of their work.

The guidelines have served us well through three versions of the .NET Framework and numerous smaller projects, and they are guiding the development of the next generation of APIs for the Microsoft Windows operating system.

With this book, I hope and expect that you will also be successful in making your frameworks, class libraries, and components easy to understand and use.

Good luck and happy designing.

Anders Hejlsberg
Redmond, WA
June 2005

Preface

This book, *Framework Design Guidelines*, presents best practices for designing frameworks, which are reusable object-oriented libraries. The guidelines are applicable to frameworks in various sizes and scales of reuse, including the following:

- Large system frameworks, such as the .NET Framework, usually consisting of thousands of types and used by millions of developers.
- Medium-size reusable layers of large distributed applications or extensions to system frameworks, such as the Web Services Enhancements.
- Small components shared among several applications, such as a grid control library.

It is worth noting that this book focuses on design issues that directly affect the programmability of a framework (publicly accessible APIs[1]). As a result, we generally do not cover much in terms of implementation details. Just as a user interface design book doesn't cover the details of how to implement hit testing, this book does not describe how to implement a binary sort, for example. This scope allows us to provide a definitive guide for framework designers instead of being yet another book about programming.

1. This includes public types, and their public, protected, and explicitly implemented members of these types.

These guidelines were created in the early days of .NET Framework development. They started as a small set of naming and design conventions but have been enhanced, scrutinized, and refined to a point where they are generally considered the canonical way to design frameworks at Microsoft. They carry the experience and cumulative wisdom of thousands of developer hours over three versions of the .NET Framework. We tried to avoid basing the text purely on some idealistic design philosophies, and we think its day-to-day use by development teams at Microsoft has made it an intensely pragmatic book.

The book contains many annotations that explain trade-offs, explain history, amplify, or provide critiquing views on the guidelines. These annotations are written by experienced framework designers, industry experts, and users. They are the stories from the trenches that add color and setting for many of the guidelines presented.

To make them more easily distinguished in text, namespace names, classes, interfaces, methods, properties, and types are set in `monospace` font.

The book assumes basic familiarity with .NET Framework programming. A few guidelines assume familiarity with features introduced in version 3.5 of the Framework. If you are looking for a good introduction to Framework programming, there are some excellent suggestions in the Suggested Reading List at the end of the book.

Guideline Presentation

The guidelines are organized as simple recommendations using **Do**, **Consider**, **Avoid**, and **Do not**. Each guideline describes either a good or bad practice, and all have a consistent presentation. Good practices have a ✓ in front of them, and bad practices have an ✗ in front of them. The wording of each guideline also indicates how strong the recommendation is. For example, a **Do** guideline is one that should always[2] be followed (all examples are from this book):

---

2. *Always* might be a bit too strong a word. There are guidelines that should literally be always followed, but they are extremely rare. On the other hand, you probably need to have a really unusual case for breaking a **Do** guideline and still have it be beneficial to the users of the framework.

✓ **DO** name custom attribute classes with the suffix "Attribute."

```
public class ObsoleteAttribute : Attribute { ... }
```

On the other hand, **Consider** guidelines should generally be followed, but if you fully understand the reasoning behind a guideline and have a good reason to not follow it anyway, you should not feel bad about breaking the rules:

✓ **CONSIDER** defining a struct instead of a class if instances of the type are small and commonly short-lived or are commonly embedded in other objects.

Similarly, **Do not** guidelines indicate something you should almost never do:

✗ **DO NOT** assign instances of mutable types to read-only fields.

Less strong, **Avoid** guidelines indicate that something is generally not a good idea, but there are known cases where breaking the rule makes sense:

✗ **AVOID** using `ICollection<T>` or `ICollection` as a parameter just to access the `Count` property.

Some more complex guidelines are followed by additional background information, illustrative code samples, and rationale:

✓ **DO** implement `IEquatable<T>` on value types.

The `Object.Equals` method on value types causes boxing and its default implementation is not very efficient because it uses reflection. `IEquatable<T>.Equals` can offer much better performance and can be implemented so it does not cause boxing.

```
public struct Int32 : IEquatable<Int32> {
   public bool Equals(Int32 other){ ... }
}
```

Language Choice and Code Examples

One of the goals of the Common Language Runtime (CLR) is to support a variety of programming languages: those with implementations provided by Microsoft, such as C++, VB, C#, F#, Python, and Ruby, as well as third-party languages such as Eiffel, COBOL, Fortran, and others. Therefore, this book was written to be applicable to a broad set of languages that can be used to develop and consume modern frameworks.

To reinforce the message of multilanguage framework design, we considered writing code examples using several different programming languages. However, we decided against this. We felt that using different languages would help to carry the philosophical message, but it could force readers to learn several new languages, which is not the objective of this book.

We decided to choose a single language that is most likely to be readable to the broadest range of developers. We picked C#, because it is a simple language from the C family of languages (C, C++, Java, and C#), a family with a rich history in framework development.

Choice of language is close to the hearts of many developers, and we offer apologies to those who are uncomfortable with our choice.

About This Book

This book offers guidelines for framework design from the top down.

Chapter 1, "Introduction," is a brief orientation to the book, describing the general philosophy of framework design. This is the only chapter without guidelines.

Chapter 2, "Framework Design Fundamentals," offers principles and guidelines that are fundamental to overall framework design.

Chapter 3, "Naming Guidelines," contains common design idioms and naming guidelines for various parts of a framework, such as namespaces, types, and members.

Chapter 4, "Type Design Guidelines," provides guidelines for the general design of types.

Chapter 5, "Member Design," takes a further step and presents guidelines for the design of members of types.

Chapter 6, "Designing for Extensibility," presents issues and guidelines that are important to ensure appropriate extensibility in your framework.

Chapter 7, "Exceptions," presents guidelines for working with exceptions, the preferred error reporting mechanisms.

Chapter 8, "Usage Guidelines," contains guidelines for extending and using types that commonly appear in frameworks.

Chapter 9, "Common Design Patterns," offers guidelines and examples of common framework design patterns.

Appendix A, "C# Coding Style Conventions," contains a short description of coding conventions used in this book.

Appendix B, "Using FxCop to Enforce the Framework Design Guidelines," describes a tool called FxCop. The tool can be used to analyze framework binaries for compliance with the guidelines described in this book. A link to the tool is included on the DVD that accompanies this book.

Appendix C, "Sample API Specification," is a sample of an API specification that framework designers within Microsoft create when designing APIs.

Included with the book is a DVD that contains several hours of video presentations covering topics presented in this book by the authors, a sample API specification, and other useful resources.

This page intentionally left blank

Acknowledgments

This book, by its nature, is the collected wisdom of many hundreds of people, and we are deeply grateful to all of them.

Many people within Microsoft have worked long and hard, over a period of years, proposing, debating, and finally, writing many of these guidelines. Although it is impossible to name everyone who has been involved, a few deserve special mention: Chris Anderson, Erik Christensen, Jason Clark, Joe Duffy, Patrick Dussud, Anders Hejlsberg, Jim Miller, Michael Murray, Lance Olson, Eric Gunnerson, Dare Obasanjo, Steve Starck, Kit George, Mike Hillberg, Greg Schecter, Mark Boulter, Asad Jawahar, Justin Van Patten, and Mircea Trofin.

We'd also like to thank the annotators: Mark Alcazar, Chris Anderson, Christopher Brumme, Pablo Castro, Jason Clark, Steven Clarke, Joe Duffy, Patrick Dussud, Mike Fanning, Kit George, Jan Gray, Brian Grunkemeyer, Eric Gunnerson, Phil Haack, Anders Hejlsberg, David Kean, Rico Mariani, Anthony Moore, Vance Morrison, Christophe Nasarre, Dare Obasanjo, Brian Pepin, Jon Pincus, Jeff Prosise, Brent Rector, Jeffrey Richter, Greg Schechter, Chris Sells, Steve Starck, Herb Sutter, Clemens Szyperski, Mircea Trofin, and Paul Vick.

Their insights provide much needed commentary, color, humor, and history that add tremendous value to this book.

Sheridan Harrison and David Kean actually wrote and edited Appendix B on FxCop, which would not have been done without their skill and passion for this tool.

For all of the help, reviews, and support, both technical and moral, we thank Martin Heller. And for their insightful and helpful comments, we appreciate Pierre Nallet, George Byrkit, Khristof Falk, Paul Besley, Bill Wagner, and Peter Winkler.

We would also like to give special thanks to Susann Ragsdale, who turned this book from a semi-random collection of disconnected thoughts into seamlessly flowing prose. Her flawless writing, patience, and fabulous sense of humor made the process of writing this book so much easier.

About the Authors

Brad Abrams was a founding member of the Common Language Runtime and .NET Framework teams at Microsoft Corporation. He has been designing parts of the .NET Framework since 1998 and is currently Group Program Manager of the .NET Framework team. Brad started his framework design career building the Base Class Library (BCL) that ships as a core part of the .NET Framework. Brad was also the lead editor on the Common Language Specification (CLS), the .NET Framework Design Guidelines, and the libraries in the ECMA\ISO CLI Standard. Brad has authored and coauthored multiple publications, including *Programming in the .NET Environment* and *.NET Framework Standard Library Annotated Reference*, Volumes 1 and 2. Brad graduated from North Carolina State University with a B.S. in computer science. You can find his most recent musings on his blog at http://blogs.msdn.com/BradA.

Krzysztof Cwalina is a program manager on the .NET Framework team at Microsoft. He was a founding member of the .NET Framework team and throughout his career has designed many .NET Framework APIs and framework development tools, such as FxCop. He is currently leading a companywide effort to develop, promote, and apply framework design and architectural guidelines to the .NET Framework. He is also leading the team responsible for delivering core .NET Framework APIs. Krzysztof graduated with a B.S. and an M.S. in computer science from the University of Iowa. You can find his blog at http://blogs.msdn.com/kcwalina.

This page intentionally left blank

About the Annotators

Mark Alcazar wanted to be a famous sportsman. After discovering he had no hand-eye coordination or athletic ability, however, he decided a better career might be computers. Mark has been at Microsoft for the last nine years, where he's worked on the HTML rendering engine in Internet Explorer and has been a member of the Windows Presentation Foundation team since its inception. Mark is a big fan of consistent white space, peach-nectarine Talking Rain, and spicy food. He has a B.Sc. from the University of the West Indies and an M.Sc. from the University of Pennsylvania.

Chris Anderson is an architect at Microsoft in the Connected Systems Division. Chris's primary focus is on the design and architecture of .NET technologies used to implement the next generation of applications and services. From 2002 until recently he was the lead architect of the WPF team. Chris has written numerous articles and white papers, and he has presented and been a keynote speaker at numerous conferences (Microsoft Professional Developers Conference, Microsoft TechEd, WinDev, DevCon, etc.) worldwide. He has a very popular blog at www.simplegeek.com.

Christopher Brumme joined Microsoft in 1997, when the Common Language Runtime (CLR) team was being formed. Since then, he has contributed to the execution engine portions of the codebase and more broadly to the design. He is currently focused on concurrency issues in managed code. Prior to joining the CLR team, Chris was an architect at Borland and Oracle.

Pablo Castro is a technical lead in the SQL Server team. He has contributed extensively to several areas of SQL Server and the .NET Framework, including SQL-CLR integration, type-system extensibility, the TDS client-server protocol, and the ADO.NET API. Pablo is currently involved with the development of the ADO.NET Entity Framework and also leads the ADO.NET Data Services project, which is looking at how to bring data and Web technologies together. Before joining Microsoft, Pablo worked in various companies on a broad set of topics that range from distributed inference systems for credit scoring/risk analysis to collaboration and groupware applications.

Jason Clark works as a software architect for Microsoft. His Microsoft software engineering credits include three versions of Windows, three releases of the .NET Framework, and WCF. In 2000 he published his first book on software development and continues to contribute to magazines and other publications. He is currently responsible for the Visual Studio Team System Database Edition. Jason's only other passions are his wife and kids, with whom he happily lives in the Seattle area.

Steven Clarke has been a user experience researcher in the Developer Division at Microsoft since 1999. His main interests are observing, understanding, and modeling the experiences that developers have with APIs in order to help design APIs that provide an optimal experience to their users.

Joe Duffy is the development lead for parallel extensions to .NET at Microsoft. He codes heavily, manages a team of developers, and defines the team's long-term vision and strategy. Joe previously worked on concurrency in the CLR team and was a software engineer at EMC. While not geeking out, Joe spends his time playing guitar, studying music theory, and blogging at www.bluebytesoftware.com.

Patrick Dussud is a Technical Fellow at Microsoft, where he serves as the chief architect of both the CLR and the .NET Framework architecture groups. He works on .NET Framework issues across the company, helping development teams best utilize the CLR. He specifically focuses on taking advantage of the abstractions the CLR provides to optimize program execution.

Michael Fanning is the current development lead for Expression Web at Microsoft. He was an early member of the team that produced FxCop for internal use and ultimately added it to Visual Studio 2005 for release to the general public.

Kit George is a program manager on the .NET Framework team at Microsoft. He graduated in 1995 with a B.A. in psychology, philosophy, and mathematics from Victoria University of Wellington (New Zealand). Prior to joining Microsoft, he worked as a technical trainer, primarily in Visual Basic. He participated in the design and implementation of the first two releases of the Framework for the last two years.

Jan Gray is a software architect at Microsoft who now works on concurrency programming models and infrastructure. He was previously a CLR performance architect, and in the 1990s he helped write the early MS C++ compilers (e.g., semantics, runtime object model, precompiled headers, PDBs, incremental compilation, and linking) and Microsoft Transaction Server. Jan's interests include building custom multiprocessors in FPGAs.

Brian Grunkemeyer has been a software design engineer on the .NET Framework team at Microsoft since 1998. He implemented a large portion of the Framework Class Libraries and contributed to the details of the classes in the ECMA/ISO CLI standard. Brian is currently working on future versions of the .NET Framework, including areas such as generics, managed code reliability, versioning, contracts in code, and improving the developer experience. He has a B.S. in computer science with a double major in cognitive science from Carnegie Mellon University.

Eric Gunnerson found himself at Microsoft in 1994 after working in the aerospace and going-out-of-business industries. He has worked on the C++ compiler team, as a member of the C# language design team, and as an early thought follower on the DevDiv community effort. He worked on the Windows DVD Maker UI during Vista and joined the Microsoft HealthVault team in early 2007. He spends his free time cycling, skiing, cracking ribs, building decks, blogging, and writing about himself in the third person.

Phil Haack is a program manager with the ASP.NET team working on the ASP.NET MVC Framework, which is being developed in a community-driven transparent manner. The Framework driving goal is to embody and encourage certain principles of good software design: separation of concerns, testability, and the single responsibility principle, among others. Phil is also a code junkie and loves to both write software as well as write about software development on his blog.

Anders Hejlsberg is a technical fellow in the Developer Division at Microsoft. He is the chief designer of the C# programming language and a key participant in the development of the .NET Framework. Before joining Microsoft in 1996, Anders was a principal engineer at Borland International. As one of the first employees of Borland, he was the original author of Turbo Pascal and later worked as the chief architect of the Delphi product line. Anders studied engineering at the Technical University of Denmark.

David Kean is a developer on the .NET Framework team at Microsoft, where he works on the Managed Extensibility Framework (MEF), a set of building blocks for developing extensible and dynamic applications. He worked earlier on the often well-loved but also greatly misunderstood tool FxCop and its related sibling, Visual Studio Code Analysis. He graduated with a B.CS. from Deakin University in Melbourne, Australia, and is now based in Seattle with his wife, Lucy, and two children, Jack and Sarah.

Rico Mariani began his career at Microsoft in 1988, working on language products, beginning with Microsoft C version 6.0, and he contributed there until the release of the Microsoft Visual C++ version 5.0 development system. In 1995, Rico became development manager for what was to become the "Sidewalk" project, which started his seven years of platform work on various MSN technologies. In the summer of 2002, Rico returned to the Developer Division as a performance architect on the CLR team. His performance work led to his most recent assignment as chief architect of Visual Studio. Rico's interests include compilers and language theory, databases, 3D art, and good fiction.

Anthony Moore is a development lead for the Connected Systems Division. He was the development lead for the Base Class Libraries of the CLR from 2001 to 2007, spanning FX V1.0 to FX 3.5. Anthony joined Microsoft in 1999 and initially worked on Visual Basic and ASP.NET. Before that he worked as a corporate developer for eight years in his native Australia, including a three-year period working in the snack food industry.

Vance Morrison is a performance architect for the .NET Runtime at Microsoft. He involves himself with most aspects of runtime performance, with current attention devoted to improving startup time. He has been involved in designs of components of the .NET runtime since its inception. He previously drove the design of the .NET Intermediate Language (IL) and has been the development lead for the JIT compiler for the runtime.

Christophe Nasarre is a software architect and development lead for Business Objects, a multinational software company from SAP that is focused on business intelligence solutions. During his spare time, Christophe writes articles for *MSDN Magazine*, MSDN, and ASPToday. Since 1996, he has also worked as a technical editor on numerous books on Win32, COM, MFC, .NET, and WPF. In 2007, he wrote his first book, *Windows via C/C++* from Microsoft Press.

Dare Obasanjo is a program manager on the MSN Communication Services Platform team at Microsoft. He brings his love of solving problems with XML to building the server infrastructure utilized by the MSN Messenger, MSN Hotmail, and MSN Spaces teams. He was previously a program manager on the XML team responsible for the core XML application programming interfaces and W3C XML Schema-related technologies in the .NET Framework.

Brian Pepin is a software architect at Microsoft and is currently working on the WPF and Silverlight designers for Visual Studio. He's been involved in developer tools and frameworks for 14 years and has provided input on the design of Visual Basic 5, Visual J++, the .NET Framework, WPF, Silverlight, and more than one unfortunate experiment that luckily never made it to market.

Jonathan Pincus was a senior researcher in the Systems and Networking Group at Microsoft Research, where he focused on the security, privacy, and reliability of software and software-based systems. He was previously founder and CTO of Intrinsa and worked in design automation (placement and routing for ICs and CAD frameworks) at GE Calma and EDA Systems.

Jeff Prosise is a cofounder of Wintellect (www.wintellect.com). His most recent book, *Programming Microsoft .NET*, was published by Microsoft Press in 2002, and his writings appear regularly in *MSDN Magazine* and other developer magazines. Jeff's professional life revolves around ASP.NET, ASP.NET AJAX, and Silverlight. A reformed engineer who discovered after college that there's more to life than computing loads on mounting brackets, Jeff is known to go out of his way to get wet in some of the world's best dive spots and to spend way too much time building and flying R/C aircraft.

Brent Rector is a program manager at Microsoft on a technical strategy incubation effort. He has more than 30 years of experience in the software development industry in the production of programming language compilers, operating systems, ISV applications, and other products. Brent is the author and coauthor of numerous Windows software development books, including *ATL Internals*, *Win32 Programming* (both Addison-Wesley), and *Introducing WinFX* (Microsoft Press). Prior to joining Microsoft, Brent was the president and founder of Wise Owl Consulting, Inc. and chief architect of its premier .NET obfuscator, Demeanor for .NET.

Jeffrey Richter is a cofounder of Wintellect (www.Wintellect.com), a training, debugging, and consulting firm dedicated to helping companies build better software faster. He is the author of several best-selling .NET and Win32 programming books, including *Applied Microsoft .NET Framework Programming* (Microsoft Press). Jeffrey is also a contributing editor at *MSDN Magazine*, where he writes the "Concurrent Affairs" column. Jeff has been consulting with Microsoft's .NET Framework team since 1999 and was also a consultant on Microsoft's Web Services and Messaging Team.

Greg Schechter has been working on API implementation and API design for over 20 years, primarily in the 2D and 3D graphics realm, but also in media, imaging, general user interface systems, and asynchronous programming. Greg is currently an architect on the Windows Presentation Foundation and Silverlight teams at Microsoft. Prior to coming to Microsoft in 1994, Greg was at Sun Microsystems for six years. Beyond all of that, Greg also loves to write about himself in the third person.

Chris Sells is a program manager for the Connected Systems Division at Microsoft. He's written several books, including *Programming WPF*, *Windows Forms 2.0 Programming*, and *ATL Internals*. In his free time, Chris hosts various conferences and makes a pest of himself on Microsoft internal product team discussion lists.

Steve Starck is a technical lead on the ADO.NET team at Microsoft, where he has been developing and designing data access technologies, including ODBC, OLE DB, and ADO.NET, for the past ten years.

Herb Sutter is a leading authority on software development. During his career, Herb has been the creator and principal designer of several major commercial technologies, including the PeerDirect peer replication system for heterogeneous distributed databases, the C++/CLI language extensions to C++ for .NET programming, and most recently the Concur concurrent programming model. Currently a software architect at Microsoft, he also serves as chair of the ISO C++ standards committee and is the author of four acclaimed books and hundreds of technical papers and articles on software development topics.

Clemens Szyperski joined Microsoft Research as a software architect in 1999. He focuses on leveraging component software to effectively build new kinds of software. Clemens is cofounder of Oberon Microsystems and its spin-off, Esmertec, and he was an associate professor at the School of Computer Science, Queensland University of Technology, Australia, where he retains an adjunct professorship. He is the author of the Jolt award-winning *Component Software* (Addison-Wesley) and the coauthor of *Software Ecosystem* (MIT Press). He has a Ph.D. in computer science from the Swiss Federal Institute of Technology in Zurich and an M.S. in electrical engineering/computer engineering from the Aachen University of Technology.

Mircea Trofin is a program manager with the .NET Application Framework Core group at Microsoft. He is primarily responsible for driving the effort for ensuring and improving the architecture of the .NET Framework. He is also responsible for a number of upcoming features in .NET in the area of component-based programming. He received his B.A.Sc. in computer engineering from University of Waterloo, and his Ph.D. in computer science from University College Dublin.

Paul Vick is the language architect for Visual Basic, leading the language design team. Paul originally began his career working at Microsoft in 1992 on the Microsoft Access team, shipping versions 1.0 through 97 of Access. In 1998, he moved to the Visual Basic team, participating in the design and implementation of the Visual Basic compiler and driving the redesign of the language for the .NET Framework. He is the author of the Visual Basic .NET Language Specification and the Addison-Wesley book *The Visual Basic .NET Language*. His weblog can be found at www.panopticoncentral.net.

This page intentionally left blank

✓ **CONSIDER** naming factory types by concatenating the name of the type being created and `Factory`. For example, consider naming a factory type that creates `Control` objects `ControlFactory`.

The next section discusses when and how to design abstractions that might or might not support some features.

9.6 LINQ Support

Writing applications that interact with data sources, such as databases, XML documents, or Web Services, was made easier in the .NET Framework 3.5 with the addition of a set of features collectively referred to as LINQ (Language-Integrated Query). The following sections provide a very brief overview of LINQ and list guidelines for designing APIs related to LINQ support, including the so-called Query Pattern.

9.6.1 Overview of LINQ

Quite often, programming requires processing over sets of values. Examples include extracting the list of the most recently added books from a database of products, finding the e-mail address of a person in a directory service such as Active Directory, transforming parts of an XML document to HTML to allow for Web publishing, or something as frequent as looking up a value in a hashtable. LINQ allows for a uniform language-integrated programming model for querying datasets, independent of the technology used to store that data.

> RICO MARIANI Like everything else, there are good and bad ways to use these patterns. The Entity Framework and LINQ to SQL offer good examples of how you can provide rich query semantics and still get very good performance using strong typing and by offering query compilation.
>
> The Pit of Success notion is very important in LINQ implementations. I've seen some cases where the code that runs as a result of using a LINQ pattern is simply terrible in comparison to what you would write the conventional way. That's really not good enough—EF and LINQ to SQL let you write it nicely, and you get high-quality database interactions. That's what to aim for.

In terms of concrete language features and libraries, LINQ is embodied as:

- A specification of the notion of extension methods. These are described in detail in section 5.6.
- Lambda expressions, a language feature for defining anonymous delegates.
- New types representing generic delegates to functions and procedures: `Func<...>` and `Action<...>`.
- Representation of a delay-compiled delegate, the `Expression<...>` family of types.
- A definition of a new interface, `System.Linq.IQueryable<T>`.
- The Query Pattern, a specification of a set of methods a type must provide in order to be considered as a LINQ provider. A reference implementation of the pattern can be found in `System.Linq.Enumerable` class. Details of the pattern will be discussed later in this chapter.
- Query Expressions, an extension to language syntax allowing for queries to be expressed in an alternative, SQL-like format.

```
//using extension methods:
var names = set.Where(x => x.Age>20).Select(x=>x.Name);

//using SQL-like syntax:
var names = from x in set where x.Age>20 select x.Name;
```

MIRCEA TROFIN The interplay between these features is the following: Any `IEnumerable` can be queried upon using the LINQ extension methods, most of which require one or more lambda expressions as parameters; this leads to an in-memory generic evaluation of the queries. For cases where the set of data is not in memory (e.g., in a database) and/or queries may be optimized, the set of data is presented as an `IQueryable`. If lambda expressions are given as parameters, they are transformed by the compiler to `Expression<...>` objects. The implementation of `IQueryable` is responsible for processing said expressions. For example, the implementation of an `IQueryable` representing a database table would translate Expression objects to SQL queries.

9.6.2 Ways of Implementing LINQ Support

There are three ways by which a type can support LINQ queries:

- The type can implement `IEnumerable<T>` (or an interface derived from it).
- The type can implement `IQueryable<T>`.
- The type can implement the Query Pattern.

The following sections will help you choose the right method of supporting LINQ.

9.6.3 Supporting LINQ through `IEnumerable<T>`

✓ **DO** implement `IEnumerable<T>` to enable basic LINQ support.

Such basic support should be sufficient for most in-memory datasets. The basic LINQ support will use the extension methods on `IEnumerable<T>` provided in the .NET Framework. For example, simply define as follows:

```
public class RangeOfInt32s : IEnumerable<int> {
   public IEnumerator<int> GetEnumerator() {...}
   IEnumerator IEnumerable.GetEnumerator() {...}
}
```

Doing so allows for the following code, despite the fact that `RangeOfInt32s` did not implement a `Where` method:

```
var a = new RangeOfInt32s();
var b = a.Where(x => x>10);
```

> **RICO MARIANI** Keeping in mind that you'll get your same enumeration semantics, and putting a LINQ façade on them does not make them execute any faster or use less memory.

✓ **CONSIDER** implementing `ICollection<T>` to improve performance of query operators.

For example, the `System.Linq.Enumerable.Count` method's default implementation simply iterates over the collection. Specific collection types can optimize their implementation of this method, since they often offer an `O(1)` - complexity mechanism for finding the size of the collection.

✓ **CONSIDER** supporting selected methods of `System.Linq.Enumerable` or the Query Pattern (see section 9.6.5) directly on new types implementing `IEnumerable<T>` if it is desirable to override the default `System.Linq.Enumerable` implementation (e.g., for performance optimization reasons).

9.6.4 Supporting LINQ through `IQueryable<T>`

✓ **CONSIDER** implementing `IQueryable<T>` when access to the query expression, passed to members of `IQueryable`, is necessary.

When querying potentially large datasets generated by remote processes or machines, it might be beneficial to execute the query remotely. An example of such a dataset is a database, a directory service, or Web service.

✗ **DO NOT** implement `IQueryable<T>` without understanding the performance implications of doing so.

Building and interpreting expression trees is expensive, and many queries can actually get slower when `IQueryable<T>` is implemented.

The trade-off is acceptable in the LINQ to SQL case, since the alternative overhead of performing queries in memory would have been far greater than the transformation of the expression to an SQL statement and the delegation of the query processing to the database server.

✓ **DO** throw `NotSupportedException` from `IQueryable<T>` methods that cannot be logically supported by your data source.

For example, imagine representing a media stream (e.g., an Internet radio stream) as an `IQueryable<byte>`. The `Count` method is not logically supported—the stream can be considered as infinite, and so the `Count` method should throw `NotSupportedException`.

9.6.5 Supporting LINQ through the Query Pattern

The Query Pattern refers to defining the methods in Figure 9-1 without implementing the `IQueryable<T>` (or any other LINQ interface).

Please note that the notation is not meant to be valid code in any particular language but to simply present the type signature pattern.

The notation uses `S` to indicate a collection type (e.g., `IEnumerable<T>`, `ICollection<T>`), and `T` to indicate the type of elements in that collection. Additionally, we use `O<T>` to represent subtypes of `S<T>` that are ordered. For example, `S<T>` is a notation that could be substituted with `IEnumerable<int>`, `ICollection<Foo>`, or even `MyCollection` (as long as the type is an enumerable type).

The first parameter of all the methods in the pattern (marked with `this`) is the type of the object the method is applied to. The notation uses extension-method-like syntax, but the methods can be implemented as extension methods or as member methods; in the latter case the first parameter should be omitted, of course, and the `this` pointer should be used.

Also, anywhere `Func<...>` is being used, pattern implementations may substitute `Expression<Func<...>>` for it. You can find guidelines later that describe when that is preferable.

```
S<T> Where(this S<T>, Func<T,bool>)

S<T2> Select(this S<T1>, Func<T1,T2>)
S<T3> SelectMany(this S<T1>, Func<T1,S<T2>>, Func<T1,T2,T3>)
S<T2> SelectMany(this S<T1>, Func<T1,S<T2>>)

O<T> OrderBy(this S<T>, Func<T,K>), where K is IComparable
O<T> ThenBy(this O<T>, Func<T,K>), where K is IComparable

S<T> Union(this S<T>, S<T>)
S<T> Take(this S<T>, int)
S<T> Skip(this S<T>, int)
S<T> SkipWhile(this S<T>, Func<T,bool>)

S<T3> Join(this S<T1>, S<T2>, Func<T1,K1>, Func<T2,K2>,
Func<T1,T2,T3>)

T ElementAt(this S<T>,int)
```

FIGURE 9-1: Query Pattern Method Signatures

✓ **DO** implement the Query Pattern as instance members on the new type, if the members make sense on the type even outside of the context of LINQ. Otherwise, implement them as extension methods.

For example, instead of the following:

```
public class MyDataSet<T>:IEnumerable<T>{...}
...
public static class MyDataSetExtensions{
     public static MyDataSet<T> Where(this MyDataSet<T> data, Func<T,bool>
query){...}
}
```

Prefer the following, because it's completely natural for datasets to support `Where` methods:

```
public class MyDataSet<T>:IEnumerable<T>{
     public MyDataSet<T> Where(Func<T,bool> query){...}
       ...
}
```

✓ **DO** implement `IEnumerable<T>` on types implementing the Query Pattern.

✓ **CONSIDER** designing the LINQ operators to return domain-specific enumerable types. Essentially, one is free to return anything from a `Select` query method; however, the expectation is that the query result type should be at least enumerable.

This allows the implementation to control which query methods get executed when they are chained. Otherwise, consider a user-defined type `MyType`, which implements `IEnumerable<T>`. `MyType` has an optimized `Count` method defined, but the return type of the `Where` method is `IEnumerable<T>`. In the example here, the optimization is lost after the `Where` method is called; the method returns `IEnumerable<T>`, and so the built-in `Enumerable.Count` method is called, instead of the optimized one defined on `MyType`.

```
var result = myInstance.Where(query).Count();
```

✗ **AVOID** implementing just a part of the Query Pattern if fallback to the basic `IEnumerable<T>` implementations is undesirable.

For example, consider a user-defined type `MyType`, which implements `IEnumerable<T>`. `MyType` has an optimized `Count` method defined but does not have `Where`. In the example here, the optimization is lost after the `Where` method is called; the method returns `IEnumerable<T>`, and so the built-in `Enumerable.Count` method is called, instead of the optimized one defined on `MyType`.

```
var result = myInstance.Where(query).Count();
```

✓ **DO** represent ordered sequences as a separate type, from its unordered counterpart. Such types should define `ThenBy` method.

This follows the current pattern in the `LINQ to Objects` implementation and allows for early (compile-time) detection of errors such as applying `ThenBy` to an unordered sequence.

For example, the Framework provides the `IOrderedEnumerable<T>` type, which is returned by `OrderBy`. The `ThenBy` extension method is defined for this type, and not for `IEnumerable<T>`.

✓ **DO** defer execution of query operator implementations. The expected behavior of most of the Query Pattern members is that they simply construct a new object which, upon enumeration, produces the elements of the set that match the query.

The following methods are exceptions to this rule: `All`, `Any`, `Average`, `Contains`, `Count`, `ElementAt`, `Empty`, `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Max`, `Min`, `Single`, `Sum`.

In the example here, the expectation is that the time necessary for evaluating the second line will be independent from the size or nature (e.g., in-memory or remote server) of `set1`. The general expectation is that this line simply prepares `set2`, delaying the determination of its composition to the time of its enumeration.

```
var set1 = ...
var set2 = set1.Select(x => x.SomeInt32Property);
foreach(int number in set2){...} // this is when actual work happens
```

✓ **DO** place query extensions methods in a "`Linq`" subnamespace of the main namespace. For example, extension methods for `System.Data` features reside in `System.Data.Linq` namespace.

✓ **DO** use `Expression<Func<...>>` as a parameter instead of `Func<...>` when it is necessary to inspect the query.

As discussed earlier, interacting with an SQL database is already done through `IQueryable<T>` (and therefore expressions) rather than `IEnumerable<T>`, since this gives an opportunity to translate lambda expressions to SQL expressions.

An alternative reason for using expressions is performing optimizations. For example, a sorted list can implement look-up (`Where` clauses) with binary search, which can be much more efficient than the standard `IEnumerable<T>` or `IQueryable<T>` implementations.

9.7 Optional Feature Pattern

When designing an abstraction, you might want to allow cases in which some implementations of the abstraction support a feature or a behavior, whereas other implementations do not. For example, stream implementations can support reading, writing, seeking, or any combination thereof.

One way to model these requirements is to provide a base class with APIs for all nonoptional features and a set of interfaces for the optional features. The interfaces are implemented only if the feature is actually supported by a concrete implementation. The following example shows one of many ways to model the stream abstraction using such an approach.

```
// framework APIs
public abstract class Stream {
   public abstract void Close();
   public abstract int Position { get; }
}
public interface IInputStream {
   byte[] Read(int numberOfBytes);
}
```

```
public interface IOutputStream {
   void Write(byte[] bytes);
}
public interface ISeekableStream {
   void Seek(int position);
}
public interface IFiniteStream {
   int Length { get; }
   bool EndOfStream { get; }
}

// concrete stream
public class FileStream : Stream, IOutputStream, IInputStream,
ISeekableStream, IFiniteStream {
   ...
}

// usage
void OverwriteAt(IOutputStream stream, int position, byte[] bytes){
   // do dynamic cast to see if the stream is seekable
   ISeekableStream seekable = stream as ISeekableStream;
   if(seekable==null){
      throw new NotSupportedException(...);
   }
   seekable.Seek(position);
   stream.Write(bytes);
}
```

You will notice the .NET Framework's `System.IO` namespace does not follow this model, and with good reason. Such factored design requires adding many types to the framework, which increases general complexity. Also, using optional features exposed through interfaces often requires dynamic casts, and that in turn results in usability problems.

> **KRZYSZTOF CWALINA** Sometimes framework designers provide interfaces for common combinations of optional interfaces. For example, the `OverwriteAt` method would not have to use the dynamic cast if the framework design provided `ISeekableOutputStream`. The problem with this approach is that it results in an explosion of the number of different interfaces for all combinations.

Sometimes the benefits of factored design are worth the drawbacks, but often they are not. It is easy to overestimate the benefits and underestimate the drawbacks. For example, the factorization did not help the developer who wrote the `OverwriteAt` method avoid runtime exceptions (the main reason for factorization). It is our experience that many designs incorrectly err on the side of too much factorization.

The Optional Feature Pattern provides an alternative to excessive factorization. It has drawbacks of its own but should be considered as an alternative to the factored design described previously. The pattern provides a mechanism for discovering whether the particular instance supports a feature through a query API and uses the features by accessing optionally supported members directly through the base abstraction.

```
// framework APIs
public abstract class Stream {
   public abstract void Close();
   public abstract int Position { get; }

   public virtual bool CanWrite { get { return false; } }
   public virtual void Write(byte[] bytes){
      throw new NotSupportedException(...);
   }

   public virtual bool CanSeek { get { return false; } }
   public virtual void Seek(int position){
      throw new NotSupportedException(...);
   }
   ... // other options
}

// concrete stream
public class FileStream : Stream {
   public override bool CanSeek { get { return true; } }
   public override void Seek(int position) { ... }
   ...
}

// usage
void OverwriteAt(Stream stream, int position, byte[] bytes){
   if(!stream.CanSeek || !stream.CanWrite){
      throw new NotSupportedException(...);
   }
   stream.Seek(position);
   stream.Write(bytes);
}
```

In fact, the `System.IO.Stream` class uses this design approach. Some abstractions might choose to use a combination of factoring and the Optional Feature Pattern. For example, the Framework collection interfaces are factored into indexable and nonindexable collections (`IList<T>` and `ICollection<T>`), but they use the Optional Feature Pattern to differentiate between read-only and read-write collections (`ICollection<T>.IsReadOnly property`).

✓ **CONSIDER** using the Optional Feature Pattern for optional features in abstractions.

The pattern minimizes the complexity of the framework and improves usability by making dynamic casts unnecessary.

> **STEVE STARCK** If your expectation is that only a very small percentage of classes deriving from the base class or interface would actually implement the optional feature or behavior, using interface-based design might be better. There is no real need to add additional members to all derived classes when only one of them provides the feature or behavior. Also, factored design is preferred in cases when the number of combinations of the optional features is small and the compile-time safety afforded by factorization is important.

✓ **DO** provide a simple Boolean property that clients can use to determine whether an optional feature is supported.

```
public abstract class Stream {
   public virtual bool CanSeek { get { return false; } }
   public virtual void Seek(int position){ ... }
}
```

Code that consumes the abstract base class can query this property at runtime to determine whether it can use the optional feature.

```
if(stream.CanSeek){
   stream.Seek(position);
}
```

✓ **DO** use virtual methods on the base class that throw `NotSupported-Exception` to define optional features.

```
public abstract class Stream {
   public virtual bool CanSeek { get { return false; } }
   public virtual void Seek(int position){
      throw new NotSupportedException(...);
   }
}
```

The method can be overridden by subclasses to provide support for the optional feature. The exception should clearly communicate to the user that the feature is optional and which property the user should query to determine if the feature is supported.

9.8 Simulating Covariance

Different constructed types don't have a common root type. For example, there would not be a common representation of `IEnumerable<string>` and `IEnumerable<object>` if not for a pattern implemented by `IEnumerable<T>` called Simulated Covariance. This section describes the details of the pattern.

Generics is a very powerful type system feature added to the .NET Framework 2.0. It allows creation of so-called parameterized types. For example, `List<T>` is such a type and it represents a list of objects of type `T`. The `T` is specified at the time when the instance of the list is created.

```
var names = new List<string>();
names.Add("John Smith");
names.Add("Mary Johnson");
```

Such generic data structures have many benefits over their nongeneric counterparts. But they also have some—sometimes surprising—limitations. For example, some users expect that a `List<string>` can be cast to `List<object>`, just as a `String` can be cast to `Object`. But unfortunately, the following code won't even compile.

```
List<string> names = new List<string>();
List<object> objects = names; // this won't compile
```

There is a very good reason for this limitation, and that is to allow for full strong typing. For example, if you could cast `List<string>` to a `List<object>` the following incorrect code would compile, but the program would fail at runtime.

```
static void Main(){
   var names = new List<string>();

   // this of course does not compile, but if it did
   // the whole program would compile, but would be incorrect as it
   // attempts to add arbitrary objects to a list of strings.
   AddObjects((List<object>)names);

   string name = names[0]; // how could this work?
}

// this would (and does) compile just fine.
static void AddObjects(List<object> list){
   list.Add(new object()); // it's a list of strings, really. Should we throw?
   list.Add(new Button());
}
```

Unfortunately, this limitation can also be undesired in some scenarios. For example, let's consider the following type:

```
public class CountedReference<T> {
   public CountedReference(T value);
   public T Value { get; }
   public int Count { get; }
   public void AddReference();
   public void ReleaseReference();
}
```

There is nothing wrong with casting a `CountedReference<string>` to `CountedReference<object>`, as in the following example.

```
var reference = new CountedReference<string>(...);
CountedReference<object> obj = reference; // this won't compile
```

In general, having a way to represent any instance of this generic type is very useful.

```
// what type should ??? be?
// CountedReference<object> would be nice but it won't work
static void PrintValue(??? anyCountedReference){
   Console.WriteLine(anyCountedReference.Value);
}
```

KRZYSZTOF CWALINA Of course, `PrintValue` could be a generic method taking `CountedReference<T>` as the parameter.

```
static void PrintValue<T>(CountedReference<T> any){
   Console.WriteLine(any.Value);
}
```

This would be a fine solution in many cases. But it does not work as a general solution and might have negative performance implications. For example, the trick does not work for properties. If a property needed to be typed as "any reference," you could not use `CountedReference<T>` as the type of the property. In addition, generic methods might have undesirable performance implications. If such generic methods are called with many differently sized type arguments, the runtime will generate a new method for every argument size. This might introduce unacceptable memory consumption overhead.

Unfortunately, unless `CountedReference<T>` implemented the Simulated Covariance Pattern described next, the only common representation of all `CountedReference<T>` instances would be `System.Object`. But `System.Object` is too limiting and would not allow the `PrintValue` method to access the `Value` property.

The reason that casting to `CountedReference<object>` is just fine, but casting to `List<object>` can cause all sorts of problems, is that in case of `CountedReference<object>`, the object appears only in the output position (the return type of `Value` property). In the case of `List<object>`, the object represents both output and input types. For example, `object` is the type of the input to the `Add` method.

```
// T does not appear as input to any members except the constructor
public class CountedReference<T> {
   public CountedReference(T value);
   public T Value { get; }
   public int Count { get; }
   public void AddReference();
   public void ReleaseReference();
}

// T does appear as input to members of List<T>
public class List<T> {
   public void Add(T item); // T is an input here
   public T this[int index]{
   get;
   set; // T is actually an input here
}
}
```

In other words, we say that in `CountedReference<T>`, the `T` is at covariant positions (outputs). In `List<T>`, the `T` is at covariant and contravariant (inputs) positions.

To solve the problem of not having a common type representing the root of all constructions of a generic type, you can implement what's called the Simulated Covariance Pattern.

Consider a generic type (class or interface) and its dependencies described in the code fragment that follows.

```
public class Foo<T> {
    public T Property1 { get; }
    public T Property2 { set; }
    public T Property3 { get; set; }
    public void Method1(T arg1);
    public T Method2();
    public T Method3(T arg);
    public Type1<T> GetMethod1();
    public Type2<T> GetMethod2();
}
public class Type1<T> {
   public T Property { get; }
}
public class Type2<T> {
   public T Property { get; set; }
}
```

Create a new interface (root type) with all members containing a `T` at contravariant positions removed. In addition, feel free to remove all members that might not make sense in the context of the trimmed-down type.

```
public interface IFoo<out T> {
   T Property1 { get; }
   T Property3 { get; } // setter removed
   T Method2();
   Type1<T> GetMethod1();
   IType2<T> GetMethod2(); // note that the return type changed
}
public interface IType2<T> {
   T Property { get; } // setter removed
}
```

The generic type should then implement the interface explicitly and "add back" the strongly typed members (using `T` instead of `object`) to its public API surface.

```
public class Foo<T> : IFoo<object> {
   public T Property1 { get; }
   public T Property2 { set; }
   public T Property3 { get; set;}
   public void Method1(T arg1);
   public T Method2();
   public T Method3(T arg);
   public Type1<T> GetMethod1();
   public Type2<T> GetMethod2();

   object IFoo<object>.Property1 { get; }
   object IFoo<object>.Property3 { get; }
   object IFoo<object>.Method2() { return null; }
   Type1<object> IFoo<object>.GetMethod1();
   IType2<object> IFoo<object>.GetMethod2();
}

public class Type2<T> : IType2<object> {
   public T Property { get; set; }
   object IType2<object>.Property { get; }
}
```

Now, all constructed instantiations of `Foo<T>` have a common root type `IFoo<object>`.

```
var foos = new List<IFoo<object>>();
foos.Add(new Foo<int>());
foos.Add(new Foo<string>());
```

```
...
foreach(IFoo<object> foo in foos){
   Console.WriteLine(foo.Property1);
   Console.WriteLine(foo.GetMethod2().Property);
}
```

In the case of the simple `CountedReference<T>`, the code would look like the following:

```
public interface ICountedReference<out T> {
   T Value { get; }
   int Count { get; }
   void AddReference();
   void ReleaseReference();
}

public class CountedReference<T> : ICountedReference<object> {
   public CountedReference(T value) {...}
   public T Value { get { ... } }
   public int Count { get { ... } }
   public void AddReference(){...}
   public void ReleaseReference(){...}

  object ICountedReference<object>.Value { get { return Value; } }
}
```

✓ **CONSIDER** using the Simulated Covariance Pattern if there is a need to have a representation for all instantiations of a generic type.

The pattern should not be used frivolously, because it results in additional types in the framework and can makes the existing types more complex.

✓ **DO** ensure that the implementation of the root's members is equivalent to the implementation of the corresponding generic type members.

There should not be an observable difference between calling a member on the root type and calling the corresponding member on the generic type. In many cases, the members of the root are implemented by calling members on the generic type.

```
public class Foo<T> : IFoo<object> {

  public T Property3 { get { ... } set { ... } }
  object IFoo<object>.Property3 { get { return Property3; } }
...
}
```

✓ **CONSIDER** using an abstract class instead of an interface to represent the root.

This might sometimes be a better option, because interfaces are more difficult to evolve (see section 4.3). On the other hand, there are some problems with using abstract classes for the root. Abstract class members cannot be implemented explicitly and the subtypes need to use the `new` modifier. This makes it tricky to implement the root's members by delegating to the generic type members.

✓ **CONSIDER** using a nongeneric root type if such type is already available.

For example, `List<T>` implements `IEnumerable` for the purpose of simulating covariance.

9.9 Template Method

The Template Method Pattern is a very well-known pattern described in much greater detail in many sources, such as the classic book *Design Patterns* by Gamma et al. Its intent is to outline an algorithm in an operation. The Template Method Pattern allows subclasses to retain the algorithm's structure while permitting redefinition of certain steps of the algorithm. We are including a simple description of this pattern here, because it is one of the most commonly used patterns in API frameworks.

The most common variation of the pattern consists of one or more nonvirtual (usually public) members that are implemented by calling one or more protected virtual members.

```
public Control{
   public void SetBounds(int x, int y, int width, int height){
      ...
      SetBoundsCore (...);
   }

   public void SetBounds(int x, int y, int width, int
   height, BoundsSpecified specified){
       ...
       SetBoundsCore (...);
   }
```

```
    protected virtual void SetBoundsCore(int x, int y, int width, int
    height, BoundsSpecified specified){
         // Do the real work here.
    }
}
```

The goal of the pattern is to control extensibility. In the preceding example, the extensibility is centralized to a single method (a common mistake is to make more than one overload virtual). This helps to ensure that the semantics of the overloads stay consistent, because the overloads cannot be overridden independently.

Also, public virtual members basically give up all control over what happens when the member is called. This pattern is a way for the base class designer to enforce some structure of the calls that happen in the member. The nonvirtual public methods can ensure that certain code executes before or after the calls to virtual members and that the virtual members execute in a fixed order.

As a framework convention, the protected virtual methods participating in the Template Method Pattern should use the suffix "Core."

✗ **AVOID** making public members virtual.

If a design requires virtual members, follow the template pattern and create a protected virtual member that the public member calls. This practice provides more controlled extensibility.

✓ **CONSIDER** using the Template Method Pattern to provide more controlled extensibility.

In this pattern, all extensibility points are provided through protected virtual members that are called from nonvirtual members.

✓ **CONSIDER** naming protected virtual members that provide extensibility points for nonvirtual members by suffixing the nonvirtual member name with "Core."

```
public void SetBounds(...){
   ...
   SetBoundsCore (...);
}
protected virtual void SetBoundsCore(...){ ... }
```

> **BRIAN PEPIN** I like to take the template pattern one step further and implement all argument checking in the nonvirtual public method. This way I can stop garbage entering methods that were possibly overridden by another developer, and it helps to enforce a little more of the API contract across implementations.

9.10 Timeouts

Timeouts occur when an operation returns before its completion because the maximum time allocated for the operation (timeout time) has elapsed. The user often specifies the timeout time. For example, it might take a form of a parameter to a method call.

```
server.PerformOperation(timeout);
```

An alternative approach is to use a property.

```
server.Timeout = timeout;
server.PerformOperation();
```

The following short list of guidelines describes best practices for the design of APIs that need to support timeouts.

✓ **DO** prefer method parameters as the mechanism for users to provide timeout time.

Method parameters are favored over properties because they make the association between the operation and the timeout much more apparent. The property-based approach might be better if the type is designed to be a component used with visual designers.

✓ **DO** prefer using `TimeSpan` to represent timeout time.

Historically, timeouts have been represented by integers. Integer timeouts can be hard to use for the following reasons:

- It is not obvious what the unit of the timeout is.
- It is difficult to translate units of time into the commonly used millisecond. (How many milliseconds are in 15 minutes?)

Often, a better approach is to use `TimeSpan` as the timeout type. `TimeSpan` solves the preceding problems.

```
class Server {
   void PerformOperation(TimeSpan timeout){
      ...
   }
}

var server = new Server();
server.PerformOperation(TimeSpan.FromMinutes(15));
```

Integer timeouts are acceptable if:

- The parameter or property name can describe the unit of time used by the operation, for example, if a parameter can be called `milliseconds` without making an otherwise self-describing API cryptic.
- The most commonly used value is small enough that users won't have to use calculators to determine the value, for example, if the unit is milliseconds and the commonly used timeout is less than 1 second.

✓ **DO** throw `System.TimeoutException` when a timeout elapses.

Timeout equal to `TimeSpan.Zero` means that the operation should throw if it cannot complete immediately. If the timeout equals `TimeSpan.MaxValue`, the operation should wait forever without timing out. Operations are not required to support either of these values, but they should throw an `InvalidArgumentException` if an unsupported timeout value is specified.

If a timeout expires and the `System.TimeoutException` is thrown, the server class should cancel the underlying operation.

In the case of an asynchronous operation with a timeout, the callback should be called and an exception thrown when the results of the operation are first accessed.

```
void OnReceiveCompleted(Object source, ReceiveCompletedEventArgs
asyncResult){
   MessageQueue queue = (MessageQueue)source;
   // the following line will throw if BeginReceive has timed out
   Message message = queue.EndReceive(asyncResult.AsyncResult);
   Console.WriteLine("Message: " + (string)message.Body);
   queue.BeginReceive(new TimeSpan(1,0,0));
}
```

For more information on timeouts and asynchronous operation, see section 9.2.

✗ **DO NOT** return error codes to indicate timeout expiration.

Expiration of a timeout means the operation could not complete successfully and thus should be treated and handled as any other runtime error (see Chapter 7).

9.11 XAML Readable Types

XAML is an XML format used by WPF (and other technologies) to represent object graphs. The following guidelines describe design considerations for ensuring that your types can be created using XAML readers.

✓ **CONSIDER** providing the default constructor if you want a type to work with XAML.

For example, consider the following XAML markup:

```
<Person Name="John" Age="22" />
```

It is equivalent to the following C# code:

```
new Person() { Name = "John", Age = 22 };
```

Consequently, for this code to work, the `Person` class needs to have a default constructor. Markup extensions, discussed in the next guideline in this section, are an alternative way of enabling XAML.

> **CHRIS SELLS** In my opinion, this one should really be a DO, not a CONSIDER. If you're designing a new type to support XAML, it's far preferable to do it with a default constructor than with markup extensions or type converters.

✓ **DO** provide markup extension if you want an immutable type to work with XAML readers.

Consider the following immutable type:

```
public class Person {
      public Person(string name, int age){
         this.name = name;
         this.age = age;
      }
      public string Name { get { return name; } }
      public int Age { get { return age; } }

      string name;
      int age;
}
```

Properties of such type cannot be set using XAML markup, because the reader does not know how to initialize the properties using the parameterized constructor. Markup extensions address the problem.

```
[MarkupExtensionReturnType(typeof(Person))]
public class PersonExtension : MarkupExtension {
   public string Name { get; set; }
   public int Age { get; set; }

   public override object ProvideValue(IServiceProvider serviceProvider){
      return new Person(this.Name,this.Age);
   }
}
```

Keep in mind that immutable types cannot be written using XAML writers.

✗ **AVOID** defining new type converters unless the conversion is natural and intuitive. In general, limit type converter usage to the ones already provided by the .NET Framework.

Type converters are used to convert a value from a string to the appropriate type. They're used by XAML infrastructure and in other places, such as graphical designers. For example, the string "#FFFF0000" in the following markup gets converted to an instance of a red `Brush` thanks to the type converter associated with the `Rectangle.Fill` property.

```
<Rectangle Fill="#FFFF0000"/>
```

But type converters can be defined too liberally. For example, the Brush type converter should not support specifying gradient brushes, as shown in the following hypothetical example.

```
<Rectangle Fill="HorizontalGradient White Red" />
```

Such converters define new "minilanguages," which add complexity to the system.

✓ **CONSIDER** applying the `ContentPropertyAttribute` to enable convenient XAML syntax for the most commonly used property.

```
[ContentProperty("Image")]
public class Button {
   public object Image { get; set; }
}
```

The following XAML syntax would work without the attribute:

```
<Button>
   <Button.Image>
      <Image Source="foo.jpg">
   </Button.Image>
</Button>
```

The attribute makes the following much more readable syntax possible.

```
<Button>
   <Image Source="foo.jpg">
</Button>
```

9.12 And in the End...

The process of creating a great framework is demanding. It requires dedication, knowledge, practice, and a lot of hard work. But in the end, it can be one of the most fulfilling jobs software engineers ever get to do. Large system frameworks can enable millions to build software that was not possible before. Application extensibility frameworks can turn simple applications into powerful platforms and make them shine. Finally, reusable component frameworks can inspire and enable developers to take their applications beyond the ordinary. When you create a framework like that, please let us know. We would like to congratulate you.

This page intentionally left blank

Index

D

E