

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The .NET_logo is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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

Copyright © 2009 Microsoft Corporation

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

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.
First printing, October 2008



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 APIs¹). 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 always² 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.

✓ **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();
```

X 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. See section 9.6.5 for more details.

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 `NotSupportedException` 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 non-virtual (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.”

X 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.

The next section goes into designing APIs that need to support timeouts.

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(new TimeSpan(0,15,0));
```

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(0)` 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.

X 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.



Index

A

- Abbreviations, 48, 377
- Abstract classes
 - choosing between interfaces and, 88–94
 - constructor design for, 148
 - designing, 95–97
 - extensibility using, 78
 - FxCop rules for, 384
 - implementing abstractions as, 203–205
 - Optional Feature Pattern and, 344–348
- Abstract types
 - abstract class design, 95
 - choosing between interfaces and, 93, 203–205, 384
 - defined, 417
- Abstractions
 - implementing with base classes, 206–207
 - in low vs. high-level APIs, 33–36
 - providing extensibility with, 203–205
 - in scenario-driven design, 17
 - in self-documenting APIs, 31–32
 - using classes vs. interfaces, 86–95
- AccessViolationException, 237
- Acronyms
 - avoiding in framework identifiers, 49
 - capitalization rules for, 40–42, 375
 - correct spelling of, 377
 - naming conventions for, 48
- Action<...> delegates, 198–200
- Addition through subtraction, 24
- Adjective phrases, naming interfaces, 60–61
- Aggregate Components, 289–298
 - component-oriented design, 291–294
 - design guidelines, 295–298
 - factored types, 294–295
 - overview of, 289–291
- Alias names, avoiding, 50
- API specification, sample, 405–412
 - API specification, 407–408
 - functional specification, 409–412
 - overview of, 406
 - requirements, 406
- APIs, naming new versions of, 51–54, 378
- _AppDomain, 63
- Application model
 - defined, 417
 - namespaces, 58–59
- Argument exceptions
 - ArgumentException, 235–236
 - ArgumentNullException, 180, 235–236
 - ArgumentOutOfRangeException, 235–236
 - FxCop rules for, 396
- Arguments
 - avoiding space between, 366
 - validating, 179–183
- ArrayList, 251

Arrays

- choosing between collections and, 245, 258–259
- FxCop rules for, 387
- of reference vs. value types, 84–85
- usage guidelines, 245–247, 397–398
- using params, 186–189
- working with properties that return, 136–138

ASP.NET, layered architecture, 35–36

Assemblies

- defined, 417
- FxCop rules for naming, 378
- naming conventions, 54–55
- type design guidelines, 118–119

AssemblyCopyrightAttribute, 119

AssemblyFileVersionAttribute, 119

AssemblyVersionAttribute, 119

Assignments, 85

Async Patterns, 298–312

- choosing between, 298–300
- Classic Async Pattern, 300–304
- Classic Async Pattern example, 304–305
- Event-Based Async Pattern, 305–312
- overview of, 298

Asynchronous methods, Event-Based Async Pattern, 305–307

Attached dependency property, 315–316, 417

Attribute class, 247

Attributes

- assembly, 119
 - defined, 417
 - usage guidelines, 247–250, 398
 - using properties vs. methods, 134–135
- AttributeUsageAttribute, 248, 398

B

Base classes

- designing for extensibility, 206–207, 394
- naming conventions, 62–63

Basic Dispose Pattern

- finalizable types and, 328–332
- overview of, 322–328
- when to implement, 321–322

beforefieldinit metadata, 389

Begin method, Classic Async Pattern, 301–303, 305

Binary operators, 366

Blogs, suggested, 415

Blue screens, Windows, 214

Books, suggested reading list, 413–415

Boolean properties

- choosing for parameters, 177–179
- implementing Optional Feature Pattern, 347
- selecting names for, 69–70

Boxing, 417

Brace usage, 364–367

C

C# coding style conventions, 363–370

- brace usage, 364–365
- comments, 368–369
- file organization, 369–370
- indent usage, 367
- naming conventions, 367–368
- space usage, 365–366
- var keyword usage, 367

Callbacks

- Data Contract Serialization, 277
- defined, 417

- mechanisms of, 153
- providing extensibility with, 197–201

camelCasing convention

- C# coding style, 368
- capitalizing acronyms, 41
- FxCop rules for, 375
- parameter names using, 39–40, 73–74

Cancellation, Event-Based Async Pattern, 308–309

Capitalization conventions, 38–46

- acronyms, 40–42
- case sensitivity, 45–46
- common terms, 43–46
- compound words, 43–46
- defined, 38
- FxCop rules for, 374–375, 383
- identifiers, 38–40

Case sensitivity, 45–46, 376

Case statements, omitting braces in, 365

Change notifications, 142–144, 317–318

- Class constructors. *See* Type constructors
- Classes
 - base. *See* Base classes
 - choosing interfaces vs., 88–95
 - choosing structs vs., 84–88
 - FxCop rules for, 384–385
 - naming conventions, 60–67, 379–381
 - as reference types, 78
 - sealing, 207–210
 - unsealed, 194–195
- Classic Async Pattern
 - choosing between async patterns, 298–300
 - example, 304–305
 - overview of, 300–304
- CLI (Common Language Infrastructure), 418
- Client-first programming test, 3
- Clone method, `ICloneable` interface, 204, 264
- `Close()` method, Basic Dispose Pattern, 327
- CLR
 - allowing overloading, 122–123
 - avoiding language-specific type names, 50–51
 - case sensitivity of, 45
 - releasing managed memory, 319
 - releasing unmanaged resources with finalizers, 319–320
- CLS (Common Language Specification), 418
- `CLSCompliant (true)` attribute, 421
- Coercion logic, dependency properties, 318–319
- Collection parameters, 252–253
- Collections, usage guidelines, 250–261
 - choosing between arrays and, 245, 258–259
 - collection parameters, 252
 - FxCop rules, 398–399
 - implementing custom collections, 259–260
 - naming custom collections, 260–261
 - overview of, 250–251
 - properties and return values, 253–257
 - property names, 69
 - snapshots vs. live collections, 257–258
- `Collection<T>` base class
 - designing extensibility, 206–207
 - implementing custom collections, 259–260
 - properties and return values, 253–254, 256
- `ComException`, 239
- Comments, C# conventions, 368–369
- Common Language Infrastructure (CLI), 418
- Common Language Specification (CLS), 418
- Common names
 - capitalization, 43–45
 - naming classes, structs and interfaces, 60–61
- Common types, names of, 64–66
- `CompletedSynchronously` property, `IAAsyncResult`, 302
- Component class, 205
- Component-oriented design, 291–294
- Compound words
 - capitalization rules, 43–45, 375–376
 - FCC rules for naming resources, 383
- `ComVisible(false)`, assembly attribute, 119
- Consistency
 - designing frameworks for, 6–7
 - exceptions promoting, 212
 - in self-documenting APIs, 31
- Constant fields, 161
- Constraints, 64
- Constructed type, 418
- Constructor design, 144–153
- Constructors
 - abstract class design and, 95
 - attribute usage guidelines, 249
 - design guidelines, 144–150, 389
 - designing custom exceptions, 240
 - factories vs., 333–335
 - type constructor guidelines, 151–152
- `ContentPropertyAttribute`, 360
- Conversion operators, 173–175, 336
- Core namespaces, 59
- Create-Call-Get-Pattern, 293
- Create-Set-Call-Pattern, 291–293
- `CriticalFinalizerObject`, 332
- Custom attributes, interfaces vs., 99–100

D

- Data class, 25
- Data Contract Serialization
 - choosing, 275
 - defined, 274
 - supporting, 276–280
 - XML Serialization vs., 280
- DataContractAttribute, 276
- DataMemberAttribute, 276
- DateTime, 261–263
- DateTimeKind, 263
- DateTimeOffset, 261–263
- Deadlock, 201
- Debugging, 134, 213
- Default arguments, member overloading
 - vs., 127–128
- Default constructors
 - aggregate components using, 294, 297
 - avoiding defining on structs, 101, 149
 - constructor design using, 145, 147–149
 - defined, 144, 418
 - XAML readable types using, 358–359
- Delegates, 153, 418
- Dependency, designing extension
 - methods, 164
- Dependency properties. *See* DPs (dependency properties)
- Descriptive names
 - designing extension methods, 167
 - designing generic type parameters, 64
 - designing resources, 74–75
 - designing self-documenting APIs using, 28
- Design patterns, 289–361
 - Aggregate Components. *See* Aggregate Components
 - Async Patterns. *See* Async Patterns
 - dependency properties, 312–319
 - Dispose Pattern. *See* Dispose Pattern
 - factories, 332–337
 - FxCop rules for, 402–404
 - LINQ support, 337–344
 - Optional Feature Pattern, 344–348
 - Simulated Covariance Pattern, 348–354
 - Template Method Pattern, 354–356
 - timeouts, 356–358
 - XAML readable types, 358–360
- Design Patterns* (Gamma et al), 354
 - .Design subnamespace, 83
- Directories, file organization, 369
- Dispose method, 320–328, 402–403
- Dispose Pattern, 319–332
 - Basic Dispose Pattern, 322–328
 - finalizable types, 328–332
 - FxCop rules for, 402–403
 - IDisposable interface, 266
 - overview of, 319–322
- Distributed computing, 6
- DLLs
 - naming conventions, 54–55, 378
 - type design guidelines, 118–119
- Documentation
 - naming conventions for new APIs, 52
 - purpose of providing, 27
 - self-documenting object models vs..
 - See* Self-documenting object models
- DPs (dependency properties), 312–319
 - attached, 315–316
 - change notifications, 317–318
 - defined, 418
 - designing, 313–315
 - overview of, 312–313
 - validation, 316–317
 - value coercion, 318–319
- DWORD, 110

E

- e parameter, 71
- Edit & Continue feature, 22
- EditorBrowsable attribute, 81
- EF (Entity Framework), 337
- 80/20 rule, 10
- Encapsulation, principle of, 159–160
- End method, Classic Async Pattern, 301–303, 305
- EnumIsDefined, 181–182
- Enums (enumerations), designing, 103–115
 - adding values to, 114–115
 - choosing between Boolean parameters and, 177–179
 - defined, 105
 - flag enums, 109–114
 - FxCop rules for, 385–386

- naming guidelines, 66–67, 380–381
 - simple enums, 103–109
 - validating arguments, 180–181
 - as value types, 78
 - Environment class, 98, 218
 - Equality operators, 286–287
 - Equals
 - overriding equality operators, 286
 - usage guidelines, 268–270, 400
 - Error conditions. *See* Exceptions
 - Error message design, 225–226, 232
 - Event-Based Async Pattern, 305–312
 - choosing between async patterns, 298–300
 - defining asynchronous methods, 305–307
 - supporting cancellation, 308–309
 - supporting incremental results, 311–312
 - supporting out and ref parameters, 307–308
 - supporting progress reporting, 309–311
 - Event design
 - custom event handler design, 159
 - overview of, 153–158
 - Event handlers
 - custom design for, 159
 - defined, 153, 418
 - event design guidelines, 153–158, 389–390
 - naming, 71–72
 - Event handling method, 156, 418
 - EventArgs suffix, 71–72, 156
 - EventHandler<T>, 155
 - Events
 - defined, 418
 - FxCop rules for design, 389–390
 - naming conventions, 70–72, 381
 - property change notification, 142–144
 - providing extensibility with, 197–201
 - “Ex” suffix, 45, 53
 - Exception, 234–235
 - Exception filters, 221
 - Exception handling, 227–232
 - Exceptions, 211–243
 - constructor design using, 146–147, 151
 - customizing, 239–240
 - framework design using, 22, 30
 - FxCop rules for, 395–397
 - overview of, 211–215
 - performance and, 240–243
 - standard types of, 234–239
 - throwing, 216–221
 - throwing from equality operators, 286
 - throwing from finalizers, 332
 - Exceptions, choosing type to throw, 221–234
 - error message design, 225–226, 232
 - exception handling, 227–232
 - overview of, 221–225
 - wrapping exceptions, 232–234
 - Execution failures, 218, 222
 - ExecutionEngineException, 239
 - EXEs (executables), 421
 - Expense, of framework design, 4
 - Explicit interface member implementation, 128–132
 - Expression<...> types, 198–200
 - Expression<Func<...>>, 343
 - Extensibility, designing for, 193–210
 - with abstractions, 203–205
 - base classes, 206–207
 - with events and callbacks, 197–201
 - FxCop rules for, 394
 - with protected members, 196
 - sealing, 207–210
 - with unsealed classes, 194–195
 - with virtual members, 201–203
 - Extension methods, 162–168, 418
- F**
- Façades. *See* Aggregate Components
 - Factored types, aggregate components, 294–295
 - Factories
 - Optional Feature Pattern vs., 346
 - overview of, 332–337
 - Factory methods, 145, 332–336
 - Fail fasts, 218
 - Fields
 - designing, 159–162
 - FxCop rules for design, 390–391
 - naming conventions, 72–73, 383

File organization, C#, 369–370

Finalizable types, Dispose Pattern and, 328–332

Finalize method, 146–147

Finalizers

- defined, 418
- finalizable types, 328–332
- FxCop rules for, 403–404
- limitations of, 320
- overview of, 319

Flag enums

- defined, 104
- designing, 109–114
- naming, 67, 110

FlagsAttribute, 110–111, 386

Flow control statements, 366

Framework design

- characteristics of, 3–6
- history of, 1–3
- overview of, 9–11
- principle of layered architecture, 32–36
- principle of low barrier to entry, 21–26
- principle of scenario-driven, 15–21
- principle of self-documenting object models, 26–32
- principles of, overview, 14–15
- progressive frameworks, 11–14

Func<...> delegates, 198–200

FxCop, 371–404

- defined, 371
- design patterns, 402–404
- designing for extensibility, 394
- evolution of, 372–373
- exceptions, 395–397
- how it works, 373–374
- member design. *See* Member design
- naming conventions. *See* Naming conventions, FxCop rules
- overview of, 371–372
- parameter design, 392–394
- spelling rules, 377
- type design guidelines, 384–386
- usage guidelines. *See* Usage guidelines, FxCop rules

FxCopcmd.exe, 373

FxCop.exe, 373

G

GC (Garbage Collector), 319, 320

GC.SuppressFinalize method

- constructor design, 147
- FxCop rules for design patterns, 403
- overview of, 320

Generic methods, 350, 418

Generic type parameters, names of, 64

Generics, 348–354, 419

“Get” methods, 69

GetHashCode, usage guidelines, 270–271, 400

GetObjectData, ISerializable, 282–283

Getter method, 419

Glossary, 417–421

Grid.Column, 316

H

Hashtable, 251

Hierarchy

- designing custom exceptions, 239
- namespace, 57–58
- organizing directory, 369
- organizing types into namespace, 79–80

High-level APIs, 33–36

High-level components, 419

Hungarian notation

- C# coding style conventions, 368
- positive and negative effects of, 46–47

I

“I” prefix, 62–63

IAsyncResult object, 301–303

ICloneable interface, 204, 263–264

ICollection interface, 252–253, 398–399

ICollection<T> interface

- implementing custom collections, 259–260
- supporting LINQ through IEnumerable<T>, 339
- usage guidelines, 252–254

IComparable<T> interface, 264–266

IComponent interface, 205

ID vs. id (identity or identifier), 44, 375

Identifiers, naming conventions

- abbreviations or contractions, 48–49
- acronyms, 49

- avoiding naming conflicts, 48
 - capitalization rules, 38–40
 - choosing names, 28–30
 - IDictionary<TKey, TValue>, 251, 260
 - IDisposable interface
 - as Dispose Pattern, 266
 - FxCop rules for design patterns, 402–403
 - implementing Basic Dispose Pattern, 322–324
 - releasing unmanaged resources with, 320–321
 - rules for finalizers, 403–404
 - usage guidelines, 266
 - IEnumerable interface, 252–255, 259–260
 - IEnumerable<T> interface
 - Query Pattern and, 342
 - supporting LINQ through, 339–340
 - usage guidelines for collections, 252–254
 - IEnumerator interface, 251–252
 - IEnumerator<T> interface, 251–252
 - IComparable<T> interface, 103, 264–266
 - IExtensibleDataObject interface, 279
 - ICollection<T> interface, 259–260
 - Immutable types
 - defined, 86, 419
 - enabling XAML readers with, 359
 - “Impl” suffix, 404
 - Implementation, framework, 4
 - Incremental results, Event-Based Async Pattern, 311–312
 - Indent usage, C#, 367
 - Indexed property design, 140–142, 388
 - IndexOutOfRangeException, 237
 - Infrastructure namespaces, 59
 - Inheritance hierarchy
 - base classes in, 206
 - naming classes, structs and interfaces, 61
 - Inlining, 419
 - Instance constructors, 144, 146
 - Instance method, 419
 - Instrumentation, exceptions promoting, 215
 - Int32 enum, 109
 - Integer timeouts, 357
 - Integration, framework, 6
 - Intellisense
 - naming conventions for new APIs, 52
 - naming conventions in self-documenting APIs, 29
 - operator overloads not showing in, 169
 - overview of, 27
 - strong typing for, 31
 - support for enums, 105
 - type design guidelines, 81
 - Interfaces
 - choosing between classes and, 88–95, 384–385
 - defining nested types as members of, 117
 - designing, 98–101, 385
 - designing abstractions with, 88–95, 205
 - designing extension methods for, 163–164
 - implementing members explicitly, 128–132, 387
 - naming conventions, 60–67, 379–381
 - reference and value types implementing, 78
 - .Interop subnamespace, 84
 - InvalidCastException, 175
 - InvalidOperationException, 235
 - IQueryable interface, 338
 - IQueryable<T> interface, 340–341
 - ISerializable interface, 281–283
 - Issue messages, FxCop, 373
 - It-Just-Works concept, 290
 - IXmlSerializable interface, 280–281
 - IXPathNavigable interface, 285
- J**
- Jagged arrays, 246–247
 - JIT (Just-In-Time) compiler, 160, 419
- K**
- Keyed collections, 256, 259–260
 - Keywords
 - avoiding naming identifiers that conflict with common, 48
 - FxCop rules for naming, 377
 - KnownTypeAttribute, 278–279

L

Lambda Expressions, 338, 419
 Language Integrated Query. *See* LINQ
 (Language Integrated Query)
 Language-specific names
 avoiding, 49–51
 FxCop rules for avoiding, 378
 resource names avoiding, 75
 Layered architecture principle, frame-
 works, 32–36
 Libraries, reusable, 5, 122–123
 LINQ (Language Integrated Query),
 337–344
 defined, 419
 overview of, 337–338
 supporting through `IEnumerable<T>`,
 339–340
 supporting through `IQueryable<T>`,
 340
 supporting through Query Pattern,
 341–344
 ways of implementing, 339
 LINQ to SQL, 337
`List<T>`, 251
 Live collections, snapshots vs., 257–258
 Local variables, avoiding prefixing, 368
 Low barrier to entry principle, frame-
 work design, 21–26
 Low-level APIs, 33–36
 Low-level component, 419

M

Managed code, 420
 Marker interfaces, avoiding, 99
 Markup Extensions, enabling XAML
 with, 358–359
`MarshalByRefObject`, 93
 Member design, 121–191
 constructor design. *See* Constructor
 design
 event design, 153–158
 extension methods, 162–168
 field design, 159–162
 member overloading. *See* Member
 overloading
 operator overloads, 168–175
 parameter design. *See* Parameter
 design
 property design, 138–144

Member design, FxCop rules for,
 387–394
 constructor design, 389
 event design, 389–390
 field design, 390–391
 general guidelines, 387–388
 operator overloads, 391–392
 parameter design, 392–394
 property design, 388
 Member overloading, 121–138
 avoiding inconsistent ordering, 124
 avoiding `ref` or `out` modifiers,
 125–126
 choosing between properties and
 methods, 132–138
 default arguments vs., 127–128
 implementing interface members
 explicitly, 128–132
 overview of, 121–123
 passing optional arguments, 126–127
 semantics for same parameters, 126
 using descriptive parameter names
 for, 123–124
 Members
 defined, 420
 PascalCasing for naming, 39–40
 providing extensibility with virtual,
 201–203
 renaming, 130–131
 sealing, 207–210
 with variable number of parameters,
 186–189
 Memory, reference vs. value types, 85
 Metadata
 capitalization guidelines, 44
 defined, 420
 PropertyMetadata, 318
 types and assembly, 118–119
 Methods
 choosing between properties and,
 132–138, 386–387
 designing extension, 162–168
 exception builder, 220–221
 naming conventions for, 28–30, 68
 naming for operator overloads,
 171–173
 supporting timeouts with parameters,
 356
 Microsoft Office, for FxCop spelling
 rules, 377

Microsoft Office Proofing Tools, for
 FxCop spelling rules, 377
 Microsoft Windows, blue screens, 214
 Microsoft Word development, 10
 Multidimensional arrays, jagged arrays
 vs., 246–247
 Multiline syntax (`/*...*/`), 369
 Multiple inheritance, 98–101
 Mutable types, 101–102, 162

N

Namespaces
 defining extension methods in,
 164–167
 for experimentation, 22–24
 exposing layers in same, 35–36
 exposing layers in separate, 35
 placing base classes in separate, 207
 standard subnamespace names, 83–84
 type design guidelines, 79–83
 Namespaces, naming
 FxCop rules, 378–379, 384
 overview of, 56–59
 PascalCasing for, 39–40
 type name conflicts and, 58–60
 Naming conventions, 37–75
 abbreviations, 48
 acronyms, 48
 assemblies, 54–55
 avoiding language-specific names,
 49–51
 C# coding style, 367–368
 capitalization. *See* Capitalization
 conventions
 classes, 60–67
 common types, 64–66
 custom collections, 260–261
 DLLs, 54–55
 enumerations, 66–67
 events, 70–72
 fields, 72–73
 generic type parameters, 64
 interfaces, 60–67
 methods, 68
 namespaces, 56–60
 new versions of APIs, 51–54
 overview of, 37–38
 parameters, 73–74
 properties, 68–70

resources, 74–75
 self-documenting APIs, 28–30
 structs, 60–67
 word choice, 46–48
 Naming conventions, FxCop rules,
 374–383
 assemblies and DLLs, 378
 classes, structs, and interfaces,
 379–381
 general, 376–378
 namespaces, 378–379
 overview of, 374–376
 parameters, 383
 resources, 383
 type members, 381–383
 NativeOverlapped*, 305
 Nested types
 defined, 420
 designing, 115–117
 FxCop rules for, 386
 FxCop type design guidelines, 386
 .NET Framework
 designing self-documenting APIs,
 31–32
 main goals of, 14
 as progressive framework, 13
 .NET Remoting, 281
 NotSupportedException, 340, 348
 Nouns/noun phrases
 naming classes, structs and interfaces
 with, 60–61
 property names, 68–69
 Nullable<T> interface, 266–268
 NullReferenceException, 237

O

Object models, 17. *See also* Self-docu-
 menting object models
 Object-oriented (OO) design, 2, 211–212
 Object-oriented programming (OOP), 2,
 79
 Object, usage guidelines, 268–273
 defining extension methods on, 165
 Object.Equals, 268–270
 Object.GetHashCode, 270–271
 Object.ToString method, 271–273
 Object.Equals
 overriding equality operators, 286
 usage guidelines, 268–270, 400

- Object.GetHashCode, 270–271, 400
 - Objects, defined, 420
 - Object.ToString method, 271–273
 - Ok, capitalizing, 44
 - OnDeserializedAttribute, 277–278
 - OO (object-oriented) design, 2, 211–212
 - OOP (object-oriented programming), 2, 79
 - Open sets, and enums, 105
 - Operator overloads
 - conversion operators, 173–175
 - defined, 123
 - descriptive parameter names for, 74, 123–124
 - extension methods similar to, 167
 - FxCop rules, 391–392
 - overloading operator ==, 173
 - overview of, 168–173
 - Operators, FxCop usage guidelines, 401–402
 - Optional Feature Pattern, 344–348
 - Organizational hierarchies, 57
 - out parameters
 - avoiding use of, 184–185
 - Classic Async Pattern, 302
 - Event-Based Async Pattern, 307–308
 - FxCop rules for parameter passing, 393
 - member overloading and, 125–126
 - parameter design, 176–177
 - passing arguments through, 184
 - OutOfMemoryException, 238
 - Overlapped class, 305
 - Overloading
 - avoiding for custom attribute constructors, 249
 - defined, 420
 - designing APIs for experimentation using, 24
 - equality operators, 286–287
 - member. *See* Member overloading
 - Overloading operator ==, 173, 175
- P**
- Parameter design, 175–191
 - enum vs. boolean parameters, 177–179
 - FxCop rules for, 392–394
 - indexed properties, 141
 - members with variable number of parameters, 186–189
 - overview of, 175–177
 - parameter passing, 183–185
 - pointer parameters, 190–191
 - providing good defaults, 25–26
 - space usage, 366
 - validating arguments, 179–183
 - Parameter names
 - camelCasing for, 39–40
 - conventions, 73–74
 - conventions for overloads, 123–124
 - event handlers and, 71
 - FxCop rules for, 383
 - operator overload and, 74
 - Parameter passing, 183–185
 - params keyword, 186–188
 - Parentheses, space usage and, 366
 - PascalCasing convention
 - C# coding style conventions, 368
 - field names, 72
 - FxCop rules for, 375
 - identifier names, 38–40
 - namespace names, 57
 - property names, 68–69
 - resource names, 74–75
 - Patterns, common design. *See* Design patterns
 - Performance
 - exceptions and, 240–243
 - implications of throwing exceptions, 219
 - supporting LINQ through IEnumerable<T>, 340
 - .Permission subnamespace, 83
 - Pit of Success notion, LINQ, 337
 - Plural namespace names, 57
 - Pointer parameters, 190–191
 - Post-events, 153–154, 420
 - PowerCollections project, 205
 - Pre-events
 - allowing end user to cancel events, 158
 - defined, 420
 - examples of, 153
 - Prefixes
 - Boolean properties, 69
 - class names, 61

- enum names, 67
- field names avoiding, 72
- interface names, 62–63
- namespace names, 56
- Program errors, 222–224
- Programming languages
 - case sensitivity guideline for, 45–46
 - choice of programming model and, 12
 - designing framework to work well with variety of, 11
 - exception handling syntax, 217
 - writing scenario code samples in different, 16, 18
- Progress reporting, 309–311
- ProgressChanged event, 309–312
- Progressive frameworks, 13–14, 420
- Properties
 - accessing fields with, 160
 - choosing between methods and, 132–138, 386–387
 - collection, 399
 - defined, 420
 - designing, 388
 - naming, 68–70, 381
 - providing good defaults for, 25–26
 - setting with constructor parameters, 145–146
 - usage guidelines for collections, 253–254
 - using attribute, 247–248
- Property change notification events, 142–144
- Property design
 - indexed, 140–142
 - overview of, 138–140
 - property change notification events, 142–144
- PropertyMetadata, 318–319
- Protected members, 194, 196
- Prototyping, implementation vs., 4
- Public nested type guidelines, 117
- Python, 18

Q

- Query Expressions, 338
- Query Pattern, 338, 341–344

R

- Raising events, usage guidelines, 154–155
- readonly fields, 161–162
- “ReadOnly” prefix, custom collections, 261
- ReadOnlyCollection<T>, 252–256, 259–260
- Recursive acquires, 201
- Reentrancy, 201
- ref parameters
 - Classic Async Pattern and, 302
 - Event-Based Async Pattern and, 307–308
 - member overloading and, 125–126
 - parameter design and, 176
 - passing arguments through, 183–185
- Reference types
 - defined, 420
 - equality operators and, 269–270, 287
 - overview of, 77
 - parameter passing guidelines, 185, 393
 - value types vs., 84–88
- References, reading list for this book, 413–415
- #region blocks, 370
- Reserved parameters, 176
- Resources, naming, 74–75, 383
- Return-code error handling model, 212–213, 217
- Return values
 - error reporting based on, 212–213
 - usage guidelines for collections, 253–254
- Ruby, 18
- Runtime, layered APIs at, 36
- Runtime Serialization
 - choosing, 275
 - defined, 274
 - supporting, 281–283

S

- SafeHandle resource wrapper, 329–330
- Scenario-driven principle
 - example of. *See* API specification, sample
 - overview of, 15–19
 - usability studies, 19–21

- Sealing
 - of custom attribute classes, 250
 - defined, 420
 - FxCop rules for, 394
 - preventing extensibility with, 207–210
- Security
 - avoiding explicit members, 131
 - designing custom exceptions, 240
- SEHException, 239
- Self-documenting object models, 26–32
 - consistency, 31–32
 - limiting abstractions, 32
 - overview of, 26–30
 - strong typing, 30–31
- sender parameter, event handlers, 71
- Sentinel values, and enums, 107–108
- SerializableAttribute interface, 281
- Serialization, usage guidelines, 274–283
 - choosing right technology, 275
 - overview of, 274–275
 - supporting Data Contract Serializa-
tion, 276–280
 - supporting Runtime Serialization,
281–283
 - supporting XML Serialization,
280–281
- Setter method, 420
- Simplicity, well-designed frameworks,
3–4
- Simulated Covariance Pattern, 348–354
- Single-statement blocks, brace usage,
364–365
- Singleline syntax (//...), comments, 369
- “64” suffix, 53–54
- Snapshots, live collections vs., 257–258
- SomeClass.GetReader, 334
- Source files, organizing, 369
- Space usage, C#, 365–366
- Spelling rules, FxCop, 377
- Sponsor class, 163
- StackOverflowException, 237–238
- State object, 301
- Static classes, designing, 97–98, 384–385
- Static constants, using enums, 105
- Static constructors. *See* Type constructors
- Static fields
 - defined, 421
 - initializing inline, 152, 389
 - naming conventions for, 47–48, 72
- Static methods
 - defined, 421
 - extension methods invoking, 162, 166,
418
- Stream class, 322, 347
- Strong typing, 30–31, 105
- Structs
 - defining default constructors in, 149
 - defining operator overloads in, 170
 - designing, 101–103, 385
 - naming conventions, 60–67, 379–381
 - type design guidelines, 84–88,
384–385
 - as value types, 78
- Subnamespace names, 83–84
- Suffixes
 - naming common types, 64–66
 - naming enums, 67
 - naming new APIs, 52–54
- SuppressFinalize, Basic Dispose
Pattern, 324–325
- Synchronization, event design, 157
- System failure, 222, 225
- System namespaces, 59
- System.AppDomain, 63
- System.Attribute class, 247
- System.ComponentModel.Component
class, 205
- System.ComponentModel.IComponent,
205
- System.Data, 25
- System.Enum, 110
- System.Environment class, 98
- System.Environment.FailFast, 218
- System.EventHandler<T>, 155
- SystemEvents, 200
- SystemException, 234–235
- System.FlagsAttribute, 110–111
- System.InvalidCastException, 175
- System.IO namespace, 20
- System.IO.Stream class, 322, 347
- System.Object, usage guidelines,
268–273
 - defining extension methods on, 165
 - Object.Equals, 268–270
 - Object.GetHashCode, 270–271
 - Object.ToString method, 271–273
- System.ServiceProcess namespace, 291
- System.TimeoutException, 357

`System.Uri`. *See* `Uri`, usage guidelines
`System.ValueType`, 103
`System.Xml`, usage guidelines, 284–286,
 401

T

TDD (test-driven development)
 defined, 15
 framework design and, 6–7
 scenario-driven design, 19
 unsealed classes, 195
 Technology namespace groups, 59–60
 Template method, 404
 Template Method Pattern, 203, 354–356
 Tense, for event names, 70–71
 Test-driven development. *See* TDD
 (test-driven development)
 Tester-Doer Pattern, 219, 241–242
 ThenBy method, Query Pattern, 343
 This, defined, 421
 Throwing exceptions, 216–221. *See also*
 Exceptions, choosing type to throw
`TimeoutException`, 357
 Timeouts, in API design, 356–358
`TimeSpan`, 356
`Tostring` method, 240, 271–273
 Trade-offs, design, 5–6
 Transparency, Aggregate Components,
 290
 Try-Parse Pattern, 219, 242–243
 Type arguments
 calling generic methods with, 350
 in constructed types, 418
 defined, 421
 Type constructors
 defined, 144
 designing, 151–152, 389
 Type converters, and XAML readers,
 359–360
 Type design guidelines
 abstract classes, 95–97
 adding values to enums, 114–115
 assembly metadata and, 118–119
 choosing class vs. struct, 84–88
 choosing classes vs. interfaces, 88–95
 flag enums, 109–114
 FxCop rules for, 384–386
 interfaces, 98–101
 namespaces and, 79–84

nested types, 115–117
 overview of, 77–79
 simple enums, 103–109
 static classes, 97–98
 structs, 101–103

Type members, naming, 68–73
 conflicts with namespace names,
 58–60
 designing self-documenting APIs,
 28–30
 events, 70–72, 382
 fields, 72–73, 383
 methods, 68
 PascalCasing for, 39–40
 properties, 68–70, 381
 Type parameters, 64, 421

U

UEFs (unhandled exception filters), 215
 Unary operators, 366
 Unboxing, 421
 Underscores (`_`), 73, 75
 Unhandled exception handlers, 214–215
 Unmanaged code, 421
 Unmanaged resources, 319
 Unsealed classes, 194–195
`Uri`, usage guidelines, 283–284, 400–401
`UrtCop` (Universal Runtime Cop), 373
 Usability, consistency for, 31–32
 Usability studies, API, 19–21, 25
 Usage errors, 222–223
 Usage guidelines, 245–287
 arrays, 245–247
 attributes, 247–250
 collections. *See* Collections, usage
 guidelines
`DateTime` and `DateTimeOffset`,
 261–263
 equality operators, 286–287
`ICloneable`, 263–264
`IComparable<T>` and `IEquatable<T>`,
 264–266
`IDisposable`, 266
`Nullable<T>`, 266–268
`Object`, 268–273
 serialization. *See* Serialization, usage
 guidelines
`System.Xml` usage, 284–286
`Uri`, 283–284

Usage guidelines, FxCop rules, 397–402
 arrays, 397–398
 attributes, 398
 collections, 398–399
 common operators, 401–402
 Object, 400
 Object.GetHashCode, 400
 System.Xml, 401
 System.Xml, 401
 Uri, 400–401
 User education experts, 29
 using directives, file organization, 370

V

Validation
 argument, 179–183
 dependency property, 316–317
 Value coercion, dependency property, 318–319
 Value types
 defined, 421
 equality operators on, 269, 287
 explicitly implementing members on, 129
 finalizable, 330
 implementing interface with, 94
 mutable, 101–102
 overview of, 77–78
 reference types vs., 84–88
 Values
 adding to enums, 114–115
 enum design and, 105–109
 using properties vs. methods, 135
 ValueType, 103
 var keyword usage, 367
 varargs methods, 189
 Variance, 130
 Verbs/verb phrases
 event names, 70
 method names, 68
 Versions, naming new API, 51–54

Virtual members, 149–150, 201–203
 Visual Basic developers
 case insensitivity of VB, 45
 designing frameworks for, 10
 experimental approach of, 22
 moving to .NET platform, 34
 problems with VB.NET, 15
 Visual Studio, 299
 VisualOperations class, WPF, 22
 <V>.<S>..<R> format, assemblies, 119

W

Wait handles, Classic Async Pattern, 300
 Word choice
 FxCop rules for, 376–377
 naming conventions, 46–48
 WPF (Windows Presentation Foundation) project, 22–23, 168
 Wrapping exceptions, 232–234

X

XAML, 358–360, 421
 XML Serialization
 choosing, 275
 defined, 274
 supporting, 280–281
 XmlDocument, usage guidelines, 285–286
 XmlNode, usage guidelines, 285
 XmlReader, usage guidelines, 285
 XmlNode, usage guidelines, 285
 XPathDocument, usage guidelines, 285
 XPathNavigator, usage guidelines, 286

Z

Zero values
 enum design, 108–109
 flag enums, 113



DVD-ROM Warranty

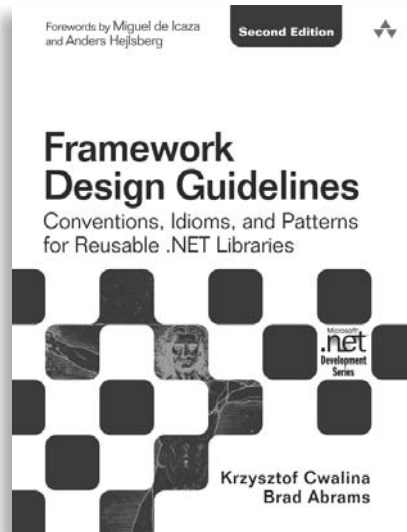
Addison-Wesley warrants the enclosed DVD-ROM to be free of defects in materials and faulty workmanship under normal use for a period of ninety days after purchase (when purchased new). If a defect is discovered in the DVD-ROM during this warranty period, a replacement DVD-ROM can be obtained at no charge by sending the defective DVD-ROM, postage prepaid, with proof of purchase to:

Disc Exchange
Addison-Wesley
Pearson Technology Group
75 Arlington Street, Suite 300
Boston, MA 02116
Email: AWPro@aw.com

Addison-Wesley makes no warranty or representation, either expressed or implied, with respect to this software, its quality, performance, merchantability, or fitness for a particular purpose. In no event will Addison-Wesley, its distributors, or dealers be liable for direct, indirect, special, incidental, or consequential damages arising out of the use or inability to use the software. The exclusion of implied warranties is not permitted in some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights. There may be other rights that you may have that vary from state to state. The contents of this DVD-ROM are intended for personal use only.

More information and updates are available at:

informit.com/aw



FREE Online Edition

Your purchase of **Framework Design Guidelines, Second Edition**, includes access to a free online edition for 45 days through the Safari Books Online subscription service. Nearly every Addison-Wesley Professional book is available online through Safari Books Online, along with more than 5,000 other technical books and videos from publishers such as Cisco Press, Exam Cram, IBM Press, O'Reilly, Prentice Hall, Que, and Sams.

SAFARI BOOKS ONLINE allows you to search for a specific answer, cut and paste code, download chapters, and stay current with emerging technologies.

Activate your FREE Online Edition at www.informit.com/safarifree

- **STEP 1:** Enter the coupon code: SKLWGCB.
- **STEP 2:** New Safari users, complete the brief registration form. Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com

