Microsoft

Adaptive Code via C#

Agile coding with design patterns and SOLID principles

Gary McLean Hall

Microsoft

Adaptive Code via C#: Agile coding with design patterns and SOLID principles

Gary McLean Hall

PUBLISHED BY
Microsoft Press
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2014 by Gary McLean Hall. All rights reserved.

No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2014943458
ISBN: 978-0-7356-8320-4

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

This book is provided "as-is" and expresses the author's views and opinions. The views, opinions and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at http://www.microsoft.com on the "Trademarks" webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Devon Musgrave
Editorial Production: Online Training Solutions, Inc. (OTSI)
Technical Reviewer: Jeremy Johnson
Copyeditor: Kathy Krause (OTSI)
Indexer: Krista Wall (OTSI)
Cover: Twist Creative • Seattle and Joel Panchot

For Amelia Rose

—Gary McLean Hall

This page intentionally left blank

Contents at a glance

Introduction *xv*

PART I AN AGILE FOUNDATION

CHAPTER 1	**Introduction to Scrum**	**3**
CHAPTER 2	**Dependencies and layering**	**43**
CHAPTER 3	**Interfaces and design patterns**	**93**
CHAPTER 4	**Unit testing and refactoring**	**125**

PART II WRITING SOLID CODE

CHAPTER 5	**The single responsibility principle**	**169**
CHAPTER 6	**The open/closed principle**	**207**
CHAPTER 7	**The Liskov substitution principle**	**217**
CHAPTER 8	**Interface segregation**	**251**
CHAPTER 9	**Dependency injection**	**281**

PART III ADAPTIVE SAMPLE

CHAPTER 10	**Adaptive sample: Introduction**	**325**
CHAPTER 11	**Adaptive sample: Sprint 1**	**337**
CHAPTER 12	**Adaptive sample: Sprint 2**	**365**

Appendix A: Adaptive tools *379*
Appendix B: GitHub code samples *online*

Index *387*

About the author *403*

This page intentionally left blank

Contents

Introduction . . . . *xv*

PART I AN AGILE FOUNDATION

Chapter 1 Introduction to Scrum 3

Scrum versus waterfall 4

Roles and responsibilities. 7

Product owner 7

Scrum master 8

Development team 8

Pigs and chickens. 9

Artifacts. 9

The Scrum board 9

Charts and metrics. 22

Backlogs. 27

The sprint 28

Release planning 29

Sprint planning. 30

Daily Scrum 31

Sprint demo 33

Sprint retrospective. 34

Scrum calendar. 36

Problems with Scrum and Agile. 37

Maladaptive code 37

Conclusion 41

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Chapter 2 Dependencies and layering 43

The definition of dependency 44
A simple example. 44
Modeling dependencies in a directed graph. 51

Managing dependencies. 56
Implementations versus interfaces. 56
The *new* code smell. 57
Alternatives to object construction 60
The Entourage anti-pattern. 63
The Stairway pattern. 65
Resolving dependencies 67
Dependency management with NuGet 77

Layering. 81
Common patterns 82
Cross-cutting concerns. 87
Asymmetric layering 89

Conclusion 91

Chapter 3 Interfaces and design patterns 93

What is an interface? 94
Syntax. 94
Explicit implementation 97
Polymorphism 101

Adaptive design patterns 102
The Null Object pattern 103
The Adapter pattern 109
The Strategy pattern. 111

Further versatility 113
Duck-typing 113
Mixins. 118
Fluent interfaces. 123

Conclusion 124

Chapter 4 Unit testing and refactoring 125

Unit testing 125
Arrange, Act, Assert. 126
Test-driven development 130
More complex tests 135
Refactoring 151
Changing existing code 151
A new account type. 160
Conclusion 165

PART II WRITING SOLID CODE

Chapter 5 The single responsibility principle 169

Problem statement 169
Refactoring for clarity 172
Refactoring for abstraction 177
SRP and the Decorator pattern 184
The Composite pattern. 185
Predicate decorators 189
Branching decorators 193
Lazy decorators 194
Logging decorators 195
Profiling decorators 196
Asynchronous decorators 200
Decorating properties and events 203
Using the Strategy pattern instead of `switch` 204
Conclusion 206

Chapter 6 The open/closed principle 207

Introduction to the open/closed principle . . . 207
- The Meyer definition . . . 207
- The Martin definition . . . 208
- Bug fixes . . . 208
- Client awareness . . . 209

Extension points . . . 209
- Code without extension points . . . 209
- Virtual methods . . . 210
- Abstract methods . . . 211
- Interface inheritance . . . 212
- "Design for inheritance or prohibit it" . . . 212

Protected variation . . . 213
- Predicted variation . . . 213
- A stable interface . . . 214
- Just enough adaptability . . . 214

Conclusion . . . 215

Chapter 7 The Liskov substitution principle 217

Introduction to the Liskov substitution principle . . . 217
- Formal definition . . . 217
- LSP rules . . . 218

Contracts . . . 219
- Preconditions . . . 220
- Postconditions . . . 222
- Data invariants . . . 223
- Liskov contract rules . . . 225
- Code contracts . . . 232

Covariance and contravariance . . . 239
- Definitions . . . 239
- Liskov type system rules . . . 246

Conclusion . . . 249

Chapter 8 Interface segregation 251

A segregation example 251
A simple CRUD interface 251
Caching 257
Multiple interface decoration 261
Client construction 263
Multiple implementations, multiple instances. 263
Single implementation, single instance. 266
The Interface Soup anti-pattern 267
Splitting interfaces 268
Client need 268
Architectural need 275
Single-method interfaces. 279
Conclusion 280

Chapter 9 Dependency injection 281

Humble beginnings 281
The Task List application. 285
Constructing the object graph 287
Inversion of Control. 291
Beyond simple injection 306
The Service Locator anti-pattern 306
Illegitimate Injection 310
The composition root 311
Convention over configuration. 317
Conclusion 322

PART III ADAPTIVE SAMPLE

Chapter 10 Adaptive sample: Introduction 325

Trey Research 325
The team 325
The product 328
Initial backlog. 328
Finding stories in prose. 329
Story point estimation 330
Summary 335

Chapter 11 Adaptive sample: Sprint 1 337

Planning 337
"I want to create rooms for categorizing conversations.". 340
The controller 340
The room repository 344
"I want to view a list of rooms that represent conversations.". 349
"I want to view the messages that have been sent to a room." 353
"I want to send plain text messages to other room members." 356
Sprint demo 357
First demonstration of Proseware 358
Sprint retrospective. 358
What went well?. 359
What went badly? 360
Things to change? 361
Things to keep? 362
Surprises?. 362
Summary 363

Chapter 12 Adaptive sample: Sprint 2 **365**

Planning . . . 365

"I want to send markdown that will be correctly formatted." . . . 367

"I want to filter message content so that it is appropriate." . . . 370

"I want to serve hundreds of users concurrently." . . . 373

Sprint demo . . . 375

Sprint retrospective. . . . 376

- What went well? . . . 376
- What went badly? . . . 377
- Things to change? . . . 377
- Things to keep? . . . 377
- Surprises? . . . 378
- Summary . . . 378

Appendix A: Adaptive Tools *379*

Appendix B: GitHub code samples *online*

Index *387*

About the author *403*

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

This page intentionally left blank

Introduction

The first words of the title of this book, *Adaptive Code*, provide a good description of the outcome of applying the principles in the book: the ability of code to adapt to any new requirement or unforeseen scenario while avoiding significant rework. The aim of this book is to aggregate into one volume many of the current best practices in the world of C# programming with the Microsoft .NET Framework. Although some of the content is covered in other books, those books either focus heavily on theory or are not specific to .NET development.

Programming can be a slow process. If your code is adaptive, you will be able to make changes to it more quickly, more easily, and with fewer errors than you would if you were working with a codebase that impedes changes. Requirements, as every developer knows, are subject to change. How change is managed is a key differentiating factor between successful software projects and those that atrophy due to scope creep. Developers can react in many ways to requirement changes, with two opposing viewpoints highlighting the continuum that lies between.

First, developers can choose a rigid viewpoint. In this approach, from the development process down to class design, the project is as inflexible as if it were implemented 50 years ago by using punch cards. Waterfall methodologies are conspicuous culprits in ensuring that software cannot change freely. Their determination that the phases of analysis, design, implementation, and testing be distinct and one-way make it difficult—or at least expensive—for customers to change requirements after implementation has begun. The code, then, does not *need* to be built for change: the process all but forbids alterations.

The second approach, Agile methodology, is not just an alternative to such rigid methodologies, but a *reaction* to them. The aim of Agile processes is to embrace change as a necessary part of the contract between client and developer. If customers want to change something in the product that they are paying for, the temporal and financial cost should be correlated to the size of the change, not the phase of the process that is currently in progress. Unlike physical engineering, software engineering works with a malleable tool: source code. The bricks and mortar that form a house are literally fused together as construction progresses. The expense involved in changing the design of a house is necessarily linked to the completion of the building phase. If the project has not been started—if it is still just in blueprints—change is relatively cheap. If the windows are in, the electricity wired up, and the plumbing fitted, moving the upstairs bathroom down next to the kitchen could be prohibitively expensive. With code, moving features around and reworking the navigation of a user interface should not be as

significant. Unfortunately, this is not always the case. The temporal cost alone often prohibits such changes. This, I find, is largely a result of a lack of adaptability in code.

This book demonstrates the second approach and explains, with real-world examples, the practicalities of implementing adaptive code.

Who should read this book

This book is intended to bridge a gap between theory and practice. The reader for whom this book is written is an experienced programmer who seeks more practical examples of design patterns, SOLID principles, unit testing and refactoring, and more.

Capable intermediate programmers who want to plug the gaps in their knowledge or have doubts and questions about how some of the industry's best practices fit together will benefit most from this book, especially because the day-to-day reality of programming rarely matches simple examples or theory. Much of SOLID is now understood, but the intricacies of the open/closed principle (covered in Chapter 6) and Liskov substitution (covered in Chapter 7) are not fully comprehended. Even experienced programmers sometimes do not fully realize the benefits provided by dependency injection (covered in Chapter 9). Similarly, the flexibility—adaptability—that interfaces (covered in Chapter 3) lend to code is often overlooked.

This book can also help the more junior developer learn, from the ground up, which aspects of common patterns and practices are benevolent and which are, in the long term, malevolent. The code samples that I see from prospective employees have a lot in common. The general theme is that the candidate is *almost there* with respect to many skills but just needs a slight push in the right direction to become a significantly better programmer. Specifically, the Entourage anti-pattern (covered in Chapter 2) and the Service Locator anti-pattern (covered in Chapter 9) are very prevalent in sample code. Practical alternatives, and their rationales, are provided in this book.

Assumptions

Ideally, you should have some practical experience of programming in a language that is syntactically similar to C#, such as Java or C++. You should also have a strong foundation in core procedural programming concepts such as conditional branching, loops, and expressions. You should also have some experience of object-oriented programming using classes, and at least a passing familiarity with interfaces.

This book might not be for you if…

This book might not be for you if you are just starting out on a journey to learn how to program. This book covers advanced topics that assume a thorough understanding of fundamental programming concepts.

Organization of this book

This book is made up of three parts, each of which builds on the last. That said, the book can also be read out of order. Each chapter covers a self-contained subject in detail, with cross references included where appropriate.

Part I: An Agile foundation

This part lays the foundation for building software in an adaptive way. It covers the high-level Agile process known as *Scrum*, which requires code to be adaptive to change. The chapters in this part focus on details around interfaces, design patterns, refactoring, and unit testing.

- **Chapter 1: Introduction to Scrum** This chapter sets the scene for the book by introducing Scrum, which is an Agile project management methodology. The chapter gives an in-depth overview of the artifacts, roles, metrics, and phases of a Scrum project. Finally, it shows how developers should organize themselves and their code when operating in an Agile environment.
- **Chapter 2: Dependencies and layering** This chapter explores dependencies and architectural layering. Code can only be adaptive if the solution's structure allows it to be. The different types of dependencies are described: first-party, third-party, and framework. The chapter describes how to manage and organize dependencies, from anti-patterns (which should be avoided) to patterns (which should be embraced). It also introduces advanced topics such as aspect-oriented programming and asymmetric layering, providing further depth.
- **Chapter 3: Interfaces and design patterns** Interfaces are, by now, ubiquitous in modern .NET development. However, they are often misapplied, misunderstood, and misappropriated. This chapter shows some of the more common and practically useful design patterns, exploring how versatile an interface can be. Leading the reader beyond the simple extraction of an interface, the chapter

shows how interfaces can be elaborated in many different ways to solve a problem. Mixins, duck-typing, and interface fluency further underscore the versatility of this key weapon in the programmer's arsenal.

- **Chapter 4: Unit testing and refactoring** Two practices that are becoming prerequisite skills are unit testing and refactoring. The two are closely related and work in unison to produce adaptive code. Without the safety net of unit tests, refactoring is prone to error; without refactoring, code becomes unwieldy, rigid, and hard to comprehend. This chapter takes an example of unit testing from humble beginnings and expands it to use more advanced—but practical—patterns and practices such as fluent assertions, test-driven development, and mocking. For refactoring, the chapter provides examples of real-world refactors that improve the readability and maintainability of the source code.

Part II: Writing SOLID code

This part builds on the foundation laid in Part I. Each chapter is devoted to examining one principle of SOLID. The emphasis in these chapters is on practical examples for implementing the principles, rather than solely on the theory of why. By placing each example in a real-world context, the chapters in this part of the book clearly demonstrate the utility of SOLID.

- **Chapter 5: The single responsibility principle** This chapter shows how to implement the single responsibility principle in practice by using the Decorator and Adapter patterns. The outcome of applying the principle is an increase in the number of classes and a decrease in the size of those classes. The chapter shows that, in contrast with monolithic classes that provide extensive features, these smaller classes are more directed and focused on solving only a small part of a larger problem. It is in their aggregation that these classes then become more than the sum of their parts.
- **Chapter 6: The open/closed principle** The open/closed principle (OCP) is simply stated, but it can have a significant impact on code. It is responsible for ensuring that code that follows SOLID principles is only appended to and never edited. This chapter also discusses the concept of predicted variation in relation to OCP and explores how it can help developers identify extension points for further adaptability.
- **Chapter 7: The Liskov substitution principle** This chapter shows the positive effects that result from applying the Liskov substitution principle on code, particularly the fact that the guidelines help enforce the open/closed principle and prevent the unintended consequences of change. Contracts—through

preconditions, postconditions, and data invariants—are covered by using the Code Contracts tooling. The chapter also describes subtyping guidelines such as covariance, contravariance, and invariance, along with the negative impact of breaking these rules.

- **Chapter 8: Interface segregation** Not only should classes be smaller than they commonly are, this chapter shows that interfaces are, similarly, often too big. Interface segregation is a simple practice that is often overlooked; this chapter shows the benefits of limiting interfaces to the smallest size possible, along with the benefits of working with smaller interfaces. It also explores the different reasons that might motivate the segregation of interfaces, such as client need and architectural need.
- **Chapter 9: Dependency injection** This chapter contains the cohesive glue that holds together the rest of the features in the book. Without dependency injection (DI), there is a lot that would not be possible—it is really that important. This chapter contains an introduction to DI and a comparison of the different methods of implementing it. The chapter includes discussions on managing object lifetimes, working with Inversion of Control containers, avoiding common anti-patterns relating to service location, and identifying composition roots and resolution roots.

Part III: Adaptive sample

This part uses a sample application as a way of tying together the rest of the book. Although there is a lot of code in these chapters, there is ample accompanying explanation. Because this book is about working in an Agile environment, the chapters map to Scrum sprints, and all work is the result of backlog items and client change requests.

- **Chapter 10: Adaptive sample: Introduction** This first chapter describes the application that is to be developed: an online chat application developed in ASP.NET MVC 5. A brief design is provided as a guideline for the planned architecture, in addition to an explanation of the features on the backlog.
- **Chapter 11: Adaptive sample: Sprint 1** Using a test-driven development (TDD) approach, the first features of the application are developed, including viewing and creating chat rooms and messages.
- **Chapter 12: Adaptive sample: Sprint 2** The client, inevitably, makes some changes to the requirements of the application, and the team accommodates those changes through adaptive code.

Appendices

Some reference material is available in the appendices, specifically for working with Git source control and to explain how the code for this book is organized on GitHub.

- **Appendix A: Adaptive tools** This is a very brief introduction to Git source control that should, at the very least, allow you to download the code from GitHub and compile it in Microsoft Visual Studio 2013. It is not intended as a thorough guide to Git—there are some excellent sources already out there, such as the official Git tutorial:

  http://git-scm.com/docs/gittutorial

  A quick web search will find other sources.

  This appendix also looks at other developer tools, such as continuous integration and the development environment.

- **Appendix B (available online only): GitHub code samples** By putting the code for this book on GitHub, I am able to make changes in a centralized location. The repository is read-only, but Appendices A and B together show you how to find the code for a listing, download it, compile it, run it, and make local changes. If you think you have found a defect or want to suggest a change, you can issue a pull request to the main AdaptiveCode repository and I will gladly take a look. You can find Appendix B via this book's page at *microsoftpressstore.com*.

Conventions and features in this book

Throughout this book, there are a number of repeated conventions. These are mainly standard to Microsoft Press publications, but it won't hurt to explain them up front.

Code listings

Code listings are included where appropriate, and a call-out is made to them where relevant, as shown in Listing I-1.

LISTING I-1 This is a code listing. There are plenty of these in the book.

```
public void MyService : IService
{

}
```

Whenever your attention should be drawn to a certain part of the code—for instance, when changes have been made to a previous example—the code will be highlighted in bold.

Readeraids and sidebars

Readeraids are used for small asides, such as notes or warnings, whereas sidebars are reserved for larger digressions. Here are some examples:

Note This is a readeraid. It contains small information nuggets that relate to the main content but have some kind of added importance.

This is a sidebar

Although this one is necessarily short, sidebars are usually reserved for longer discussions on topics that are somewhat tangential to the main topic.

Images

Sometimes, an explanation—no matter how florid—is not enough. In these cases, an image is provided. All diagrams were created in Microsoft Visio 2013 with no theming, to create a high contrast and to focus solely on exposition. Screenshots were taken with a high-contrast theme applied.

System requirements

You will need the following hardware and software to use the code samples in this book:

- Either Windows XP Service Pack 3 (except Starter Edition), Windows Vista Service Pack 2 (except Starter Edition), Windows 7, Windows Server 2003 Service Pack 2, Windows Server 2003 R2, Windows Server 2008 Service Pack 2, or Windows Server 2008 R2
- Visual Studio 2013, any edition (multiple downloads might be required if you are using Express Edition products)
- Microsoft SQL Server 2008 Express Edition or higher (2008 or R2 release), with SQL Server Management Studio 2008 Express or higher (included with Visual Studio; Express Editions require a separate download)
- A computer that has a 1.6-gigahertz (GHz) or faster processor (2 GHz recommended)
- 1 gigabyte (GB) (32 bits) or 2 GB (64 bits) of RAM (add 512 megabytes [MB] if running in a virtual machine or using SQL Server Express Editions, more for advanced SQL Server editions)
- 3.5 GB of available hard disk space
- A 5,400-RPM hard disk drive
- A DirectX 9–capable video card running at 1024 x 768 or a higher-resolution display
- A DVD-ROM drive (if installing Visual Studio from DVD)
- An Internet connection for downloading software or code samples

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2013 and SQL Server 2008 products.

Downloads: Code samples

As far as possible, I ensured that the code listings were part of a larger example that could be run either as a stand-alone application or a unit test. I wrote many of the simpler unit tests by using MSTest, so that no external test runner was needed, but I wrote the more complex unit tests by using NUnit. I used Visual Studio 2013 Ultimate to write all of the code. Although I wrote some of it by using the preview version, it has all been compiled and tested on the full version. As far as possible, I didn't use features that were unavailable to the Express Editions of Visual Studio 2013, but for some topics, this was not possible. Readers wanting to run this code will need to install a paid-for version.

The code itself is available from GitHub, at the following address:

http://aka.ms/AdaptiveCode_CodeSamples

Appendix A contains explanations for using Git, and Appendix B (online only) details how the code in the AdaptiveCode repository is organized.

If you want to make a comment where I am likely to see it, my WordPress blog is here:

http://garymcleanhall.wordpress.com

Acknowledgments

The byline for this book is not really accurate. I couldn't have written any of this without the following people, all of whom have helped me in different ways.

Victoria, my wife, for making this book possible. That's not lip service—it's simply a fact.

Amelia, my daughter, for being perfect in every way.

Pam, my mother, for proofreading and for her wonderfully hyperbolic words of encouragement.

Les, my father, for all his hard work.

Darryn, my brother, for general, continual guidance.

Kathy Krause at Online Training Solutions, Inc., for her excellent work making this book readable.

Devon Musgrave, for his apparently limitless patience.

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

http://aka.ms/Adaptive/errata

If you discover an error that is not already listed, please submit it to us at the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

This page intentionally left blank

This page intentionally left blank

CHAPTER 7

The Liskov substitution principle

After completing this chapter, you will be able to

- Understand the importance of the Liskov substitution principle.
- Avoid breaking the rules of the Liskov substitution principle.
- Further solidify your single responsibility principle and open/closed principle habits.
- Create derived classes that honor the contracts of their base classes.
- Use code contracts to implement preconditions, postconditions, and data invariants.
- Write correct exception-throwing code.
- Understand covariance, contravariance, and invariance and where each applies.

Introduction to the Liskov substitution principle

The Liskov substitution principle (LSP) is a collection of guidelines for creating inheritance hierarchies in which a client can reliably use any class or subclass without compromising the expected behavior.

If the rules of the LSP are not followed, an extension to a class hierarchy—that is, a new subclass—might necessitate changes to any client of the base class or interface. If the LSP is followed, clients can remain unaware of changes to the class hierarchy. As long as there are no changes to the interface, there should be no reason to change any existing code. The LSP, therefore, helps to enforce both the open/closed principle and the single responsibility principle.

Formal definition

The definition of the LSP by prominent computer scientist Barbara Liskov is a bit dry, so it requires further explanation. Here is the official definition:

> *If S is a subtype of T, then objects of type T may be replaced with objects of type S, without breaking the program.*
>
> *—Barbara Liskov*

There are three code ingredients relating to the LSP:

- **Base type** The type (T) that clients have reference to. Clients call various methods, any of which can be overridden—or partially specialized—by the subtype.
- **Subtype** Any one of a possible family of classes (S) that inherit from the base type (T). Clients should not know which specific subtype they are calling, nor should they need to. The client should behave the same regardless of the subtype instance that it is given.
- **Context** The way in which the client interacts with the subtype. If the client doesn't interact with a subtype, the LSP can neither be honored nor contravened.

LSP rules

There are several "rules" that must be followed for LSP compliance. These rules can be split into two categories: contract rules (relating to the expectations of classes) and variance rules (relating to the types that can be substituted in code).

Contract rules

These rules relate to the contract of the supertype and the restrictions placed on the contracts that can be added to the subtype.

- Preconditions cannot be strengthened in a subtype.
- Postconditions cannot be weakened in a subtype.
- Invariants—conditions that must remain true—of the supertype must be preserved in a subtype.

To understand the contract rules, you should first understand the concept of contracts and then explore what you can do to ensure that you follow these rules when creating subtypes. The "Contracts" section later in this chapter covers both in depth.

Variance rules

These rules relate to the variance of arguments and return types.

- There must be contravariance of the method arguments in the subtype.
- There must be covariance of the return types in the subtype.
- No new exceptions can be thrown by the subtype unless they are part of the existing exception hierarchy.

The concept of type variance in the languages of the Common Language Runtime (CLR) of the Microsoft .NET Framework is limited to generic types and delegates. However, variance in these scenarios is well worth exploring and will equip you with the requisite knowledge to write code that is LSP compliant for variance. This will be explored in depth in the "Covariance and contravariance" section later in this chapter.

Contracts

It is often said that developers should *program to interfaces*, and a related idiom is to *program to a contract*. However, beyond the apparent method signatures, interfaces convey a very loose notion of a contract. A method signature reveals little about the actual requirements and guarantees of the method's implementation, as Figure 7-1 shows. In a strongly typed language like C#, there is at least a notion of passing the correct type for an argument, but this is largely where the interface ends and the concept of the contract must begin.

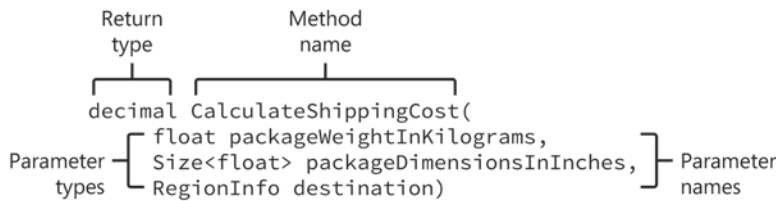

FIGURE 7-1 Method signatures reveal little about the expectations of the implementation.

All methods have at least an optional return type, a name, and an optional list of formal parameters. Each parameter consists of a type specifier and a name. When calling the method shown in Figure 7-1, you know—from only looking at the signature—that you need to pass in three parameters, one of type `float`, one of type `Size<float>`, and another of type `RegionInfo`. You also know that you can save the return value, of type `decimal`, in a variable or otherwise operate on this value after the call has been made.

> **Note** It is not advisable to use the `decimal` type to represent currency values, as is done in Figure 7-1. Instead, a `Money`[1] value type should be used. Although effort has been taken to ensure that the examples in this book are, as much as possible, relevant to a real-world context and are not just contrivances, some concessions have been made in the interest of brevity.

As a method writer, you can control the names given to parameters and methods. Take extra care to ensure that the method name truly represents the method's purpose and that the parameter names are as descriptive as possible. The `CalculateShippingCost` function's name uses a verb-noun form. Here the verb—the action performed by the method—is `Calculate`, and the noun—the object of the verb—is `ShippingCost`. This noun is, in a sense, the name of the return value. Descriptive names have also been chosen for the parameters: `packageDimensionsInInches` and `packageWeightIn-Kilograms` are self-explanatory parameter names, especially in the context of the method. They form a starting point for documenting the method.

[1] http://moneytype.codeplex.com/

Tip For further information on good variable and method naming and other best practices, Steve McConnell's *Code Complete*[2] is essential reading.

What is missing, though, is the *contract* of the method. For example, the `packageWeightInKilograms` parameter is of type `float`. What clients of this method might infer is that *any* `float` value is valid, including a negative value. Because the parameter represents a weight, a negative value should not be valid. The contract of this method should enforce a weight of greater than zero. For this, the method must implement a *precondition*.

Tip Although contracts as outlined in this chapter add run-time protection against many invalid calls to methods, the importance of good method and parameter naming is hard to exaggerate. If the formal parameters of the `CalculateShippingCost` method did not specify that they are in inches or kilograms, clients could, for example, call the method with values representing centimeters and pounds, respectively.

Preconditions

Preconditions are defined as all of the conditions necessary for a method to run reliably and without fault. Every method requires some preconditions to be true before it should be called. By default, interfaces force no guarantees on any of the implementers of their methods. Listing 7-1 shows how you can implement a precondition by using a guard clause at the start of a method.

LISTING 7-1 Throwing an exception is an effective way of enforcing precondition contracts.

```
public decimal CalculateShippingCost(
    float packageWeightInKilograms,
    Size<float> packageDimensionsInInches,
    RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f) throw new Exception();

    return decimal.MinusOne;
}
```

The `if` statement at the very start of the method is one way to enforce a precondition, such as the requirement for a positive weight. If the condition `packageWeightInKilograms <= 0f` is met, an exception is thrown and the method stops executing immediately. This certainly prevents a method

2 http://www.stevemcconnell.com/cc.htm

from being executed unless all parameters have valid values. By using a more descriptive exception, you can provide more context to the caller, as shown in Listing 7-2.

LISTING 7-2 It is important to provide as much context as possible about why the precondition caused a failure.

```
public decimal CalculateShippingCost(
    float packageWeightInKilograms,
    Size<float> packageDimensionsInInches,
    RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f)
        throw new ArgumentOutOfRangeException("packageWeightInKilograms", "Package weight
  must be positive and non-zero");

    return decimal.MinusOne;
}
```

This is an improvement on the first exception that was thrown. In addition to using an exception specifically for the purpose of out-of-range arguments, the client is also informed which parameter is errant and a description of the problem is provided.

By chaining more guard clauses like this together, you can add more conditions that must be fulfilled in order to call the method without generating an exception. The changes shown in Listing 7-3 include exceptions that are thrown when the package dimensions are out of range, too.

LISTING 7-3 As many preconditions as necessary can be added to prevent the method from being called with invalid parameters.

```
public decimal CalculateShippingCost(
    float packageWeightInKilograms,
    Size<float> packageDimensionsInInches,
    RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f)
        throw new ArgumentOutOfRangeException("packageWeightInKilograms", "Package weight
  must be positive and non-zero");

    if (packageDimensionsInInches.X <= 0f || packageDimensionsInInches.Y <= 0f)
        throw new ArgumentOutOfRangeException("packageDimensionsInInches", "Package
  dimensions must be positive and non-zero");

    return decimal.MinusOne;
}
```

With these preconditions in place, clients must ensure that the parameters that they provide are within valid ranges before calling. One corollary from this is that all of the state that is checked in a precondition *must* be publically accessible by clients. If the client is unable to verify that the method they are about to call will throw an error due to an invalid precondition, the client won't be able to ensure that the call will succeed. Therefore, private state should not be the target of a precondition; only method parameters and the class's public properties should have preconditions.

Postconditions

Postconditions check whether an object is being left in a valid state as a method is exited. Whenever state is mutated in a method, it is possible for the state to be invalid due to logic errors.

Postconditions are implemented in the same manner as preconditions, through guard clauses. However, rather than placing the clauses at the start of the method, postcondition guard clauses must be placed at the end of the method after all edits to state have been made, as Listing 7-4 shows.

LISTING 7-4 The guard clause at the end of the method is a postcondition that ensures that the return value is in range.

```
public virtual decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f)
        throw new ArgumentOutOfRangeException("packageWeightInKilograms", "Package weight
  must be positive and non-zero");

    if (packageDimensionsInInches.X <= 0f || packageDimensionsInInches.Y <= 0f)
        throw new ArgumentOutOfRangeException("packageDimensionsInInches", "Package
  dimensions must be positive and non-zero");

    // shipping cost calculation

    var shippingCost = decimal.One;

    if(shippingCost <= decimal.Zero)
        throw new ArgumentOutOfRangeException("return", "The return value is out of
  range");

    return shippingCost;
}
```

By testing state against a predetermined valid range—and throwing an exception if the value falls outside of that range—you can enforce a postcondition on the method. The postcondition here relates not to the state of the object but to the return value. Much like method argument values are tested against preconditions for validity, so are method return values tested against postconditions

for validity. If, at any point during the method, the return value is set to zero or a negative value, the postcondition will detect this and halt execution at the end of the method. This way, clients of this method will never inadvertently receive an invalid value and they can continue to assume that it will always be valid. Note that the interface of the method does not communicate that the return value will always be non-zero and positive—that is a feature of the interface's contract with clients.

Data invariants

A third type of contract is the data invariant. A *data invariant* is a predicate that remains true for the lifetime of an object; it is true after construction and must remain true until the object is out of scope. Data invariants relate to the expected internal state of the object. An example of a data invariant for the `ShippingStrategy` call is that the flat rate provided is positive and non-zero. If, as shown in Listing 7-5, the flat rate is set on construction, a simple guard clause in the constructor will prevent an invalid value from being set.

LISTING 7-5 Adding a precondition to a constructor can help protect a data invariant.

```
public class ShippingStrategy
{
    public ShippingStrategy(decimal flatRate)
    {
        if (flatRate <= decimal.Zero)
            throw new ArgumentOutOfRangeException("flatRate", "Flat rate must be positive 
  and non-zero");

        this.flatRate = flatRate;
    }

    protected decimal flatRate;
}
```

Because the `flatRate` value is a protected member variable, the only opportunity that clients have for setting the value is through the constructor. If `flatRate` is set to a valid value at this point, it is guaranteed to be valid for the rest of the lifetime of the object because clients have no way of changing this value.

However, if the `flatRate` variable is instead a publically settable property, the guard clause would have to be moved to the setter block in order to protect the data invariant. Listing 7-6 shows the flat rate refactored as a public property, with an accompanying guard clause.

LISTING 7-6 When a data invariant is a public property, the guard clause moves to the setter.

```
public class ShippingStrategy
{
    public ShippingStrategy(decimal flatRate)
    {
        FlatRate = flatRate;
    }

    public decimal FlatRate
    {
        get
        {
            return flatRate;
        }
        set
        {
            if (value <= decimal.Zero)
                throw new ArgumentOutOfRangeException("value", "Flat rate must be positive
  and non-zero");

            flatRate = value;
        }
    }
}
```

Now clients might be able to change the value of the `FlatRate` property but, because of the `if` statement and exception, the invariant cannot be broken.

Encapsulation vs. contracts

The contracts implemented in this example make sense, but they are caused by a poor choice of types for each value. The precondition contract for ensuring that the package weight argument is non-zero and positive is intrinsically linked with the type of the variable: weight should never be zero or negative. This makes weight a candidate for encapsulation into its own type. If, as is likely, another class or method requires a weight, you would need to carry this precondition across to the new code. This is inefficient, hard to maintain, and error-prone. It makes more sense to create a new type and define the precondition with it so that all uses of the `Weight` type must have a non-zero and positive value. It is, in fact, an invariant of the type rather than a precondition of the `CalculateShippingCost` method.

Similarly, the flat rate is modeled poorly by the `decimal` type. Instead, this should be promoted to its own value type, and the invariant requiring it to also be non-zero and positive should be applied to this type.

Liskov contract rules

All of this method contract discussion is merely preamble to some of the tenets of the Liskov substitution principle. The LSP sets rules by which types must inherit contracts. A reminder of the definition of the LSP is shown here:

> *If S is a subtype of T, then objects of type T may be replaced with objects of type S, without breaking the program.*

Where contracts are concerned, this leads to the guidelines that were stated earlier:

- Preconditions cannot be strengthened in a subtype.
- Postconditions cannot be weakened in a subtype.
- Invariants of the supertype must be preserved in a subtype.

If you follow all of these rules when creating subclasses of existing classes, substitutability will be retained when you are dealing with contracts.

Whenever a subclass is created, it brings with it all of the methods, properties, and fields that make up the parent class. This also includes the contracts inside the methods and property setters. Preconditions, postconditions, and data invariants are all expected to be maintained in the same way that they were in the parent class. Subclasses are, where applicable, allowed to override method implementations, which includes the possibility for changing the contracts. Liskov substitution stipulates that some changes are not allowed, because they could break existing clients that must be able to use the new subclass as if it were an instance of the superclass.

Preconditions cannot be strengthened

Whenever a subclass overrides an existing method that contains preconditions, it must never *strengthen* the existing preconditions. Doing so would potentially break any client code that already assumes that the subclass defines the strongest possible precondition contracts for any method.

Listing 7-7 shows the addition of a new `WorldWideShippingStrategy`. Due to the large number of similarities in how the classes behave, this new class is implemented as a subclass of the `ShippingStrategy` class. The `CalculateShippingCost` method is overridden to provide a new value that takes into account the destination of the package being sent via the `RegionInfo` parameter. Although the `ShippingStrategy` class did not make any guarantees that the destination of the package would be provided, `WorldWideShippingStrategy` now requires this parameter to be provided, otherwise it cannot correctly calculate how much it would cost to send the package to that location.

LISTING 7-7 This subclass adds a new guard clause, thus strengthening the preconditions.

```
public class WorldWideShippingStrategy : ShippingStrategy
{
    public override decimal CalculateShippingCost(
        float packageWeightInKilograms,
        Size<float> packageDimensionsInInches,
        RegionInfo destination)
    {
        if (packageWeightInKilograms <= 0f)
            throw new ArgumentOutOfRangeException("packageWeightInKilograms", "Package
  weight must be positive and non-zero");

        if (packageDimensionsInInches.X <= 0f || packageDimensionsInInches.Y <= 0f)
            throw new ArgumentOutOfRangeException("packageDimensionsInInches", "Package
  dimensions must be positive and non-zero");

        if (destination == null)
            throw new ArgumentNullException("destination", "Destination must be
  provided");

        return decimal.One;
    }
}
```

The temptation is to strengthen the preconditions so that you can guarantee that the destination parameter is provided. This creates a conflict that calling code is unable to solve. If a class calls the `CalculateShippingCost` method of the `ShippingStrategy` class, it is free to pass in a null value for the destination parameter without experiencing a side effect. But if it is calling the `Calculate-ShippingCost` method of the `WorldWideShippingStrategy` class, it must not pass in a null value for the destination parameter. Doing so would violate a precondition and cause an exception to be thrown. As earlier chapters have demonstrated, client code must never make assumptions about what type it is acting on. Doing so only leads to strong coupling between classes and an inability to adapt to changes in requirements.

To demonstrate the problem, examine the unit test shown in Listing 7-8.

LISTING 7-8 When the precondition is strengthened, clients cannot reliably use a `WorldWideShippingStrategy` where a `ShippingStrategy` is required.

```
[Test]
public void ShippingRegionMustBeProvided()
{
    strategy.Invoking(s => s.CalculateShippingCost(1f, ValidDimensions, null))
        .ShouldThrow<ArgumentNullException>("Destination must be provided")
        .And.ParamName.Should().Be("destination");
}
```

If the strategy used by this test is of type `WorldWideShippingStrategy`, the test will pass; no destination is provided but one is required, thus an exception meeting the specification is thrown. If a `ShippingStrategy` is used instead, this test will fail because no precondition exists to prevent the null value for the destination and no exception will be thrown.

Listing 7-9 shows a refactored set of unit tests that do not attempt to test the same preconditions on both strategy types. A test asserting that the shipping region must be provided is only valid for the `WorldWideShippingStrategy`. However, regardless of shipping strategy, the precondition that the shipping weight must be positive is always valid, so this is included in a base class of tests that will be run for each shipping strategy class.

LISTING 7-9 These refactored unit tests separately target the two shipping strategy classes.

```
[TestFixture]
public class WorldWideShippingStrategyTests : ShippingStrategyTestsBase
{
    [Test]
    public void ShippingRegionMustBeProvided()
    {
        strategy.Invoking(s => s.CalculateShippingCost(1f, ValidSize, null))
            .ShouldThrow<ArgumentNullException>("Destination must be provided")
            .And.ParamName.Should().Be("destination");
    }

    protected override ShippingStrategy CreateShippingStrategy()
    {
        return new WorldWideShippingStrategy(decimal.One);
    }
}
// . . .
public abstract class ShippingStrategyTestsBase
{
    [Test]
    public void ShippingWeightMustBePositive()
    {
        strategy.Invoking(s => s.CalculateShippingCost(-1f, ValidSize, null))
            .ShouldThrow<ArgumentOutOfRangeException>("Package weight must be positive and
  non-zero")
            .And.ParamName.Should().Be("packageWeightInKilograms");
    }
}
```

Postconditions cannot be weakened

When applying postconditions to subclasses, the opposite rule applies. Instead of not being able to strengthen postconditions, you cannot weaken them. As for all of the Liskov substitution rules relating to contracts, the reason that you cannot weaken postconditions is because existing clients might break when presented with the new subclass. Theoretically, if you comply with the LSP, any subclass you create should be usable by all existing clients without causing them to fail in unexpected ways.

One such example of causing an unexpected failure in an existing client is explored in Listing 7-10. The unit test and implementation relate to the `WorldWideShippingStrategy`, the `ShippingStrategy` subclass for international packages.

LISTING 7-10 The new implementation requires a weakening of the postcondition.

```
[Test]
public void ShippingDomesticallyIsFree()
{
   strategy.CalculateShippingCost(1f, ValidDimensions, RegionInfo.CurrentRegion)
        .Should().Be(decimal.Zero);
}
// . . .
public override decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
{
    if (destination == null)
        throw new ArgumentNullException("destination", "Destination must be provided");

    if (packageWeightInKilograms <= 0f)
        throw new ArgumentOutOfRangeException("packageWeightInKilograms", "Package weight
  must be positive and non-zero");

    if (packageDimensionsInInches.X <= 0f || packageDimensionsInInches.Y <= 0f)
        throw new ArgumentOutOfRangeException("packageDimensionsInInches", "Package
  dimensions must be positive and non-zero");

    var shippingCost = decimal.One;

    if(destination == RegionInfo.CurrentRegion)
    {
        shippingCost = decimal.Zero;
    }

    return shippingCost;
}
```

The unit test asserts that, when the current region is used for the destination—that is, the shipping is domestic—the `WorldWideShippingStrategy` does not charge for shipping at all. This is reflected in the accompanying implementation. This assertion is, again, in conflict with an existing unit test for the base class that asserts the original postcondition: that the result is always positive and non-zero, as shown in Listing 7-11.

LISTING 7-11 This unit test shows the original unit test, which fails when the strategy is a `WorldWideShipping-Strategy`.

```
[Test]
public void ShippingCostMustBePositiveAndNonZero()
{
    strategy.CalculateShippingCost(1f, ValidDimensions, RegionInfo.CurrentRegion)
        .Should().BeGreaterThan(0m);
}
```

A client could easily be broken by this change in behavior due to its assumption of the value of the shipping cost. For example, the client assumes that the shipping cost is always positive and non-zero, as indicated by the postcondition contract of the `ShippingStrategy`. This client then uses the shipping cost as the denominator in a subsequent calculation. When a switch is made to use the new `WorldWideShippingStrategy`, the client unexpectedly starts throwing `DivideByZeroException` errors for all domestic orders.

Had the LSP been honored and the postcondition never weakened, this defect would never have been introduced.

Invariants must be maintained

Whenever a new subclass is created, it must continue to honor all of the data invariants that were part of the base class. This is an easy problem to introduce because subclasses have a lot of freedom to introduce new ways of changing previously private data.

Listing 7-12 returns to the previous data invariant example from earlier in the chapter. However, in this instance, the `ShippingStrategy` accepts the flat rate value as a constructor parameter and maintains this value as a read-only data invariant. The new `WorldWideShippingStrategy` is introduced, and the means to change the flat rate value is made public through a property.

LISTING 7-12 The subclass breaks the data invariant of the superclass, violating the LSP.

```
[Test]
public void ShippingFlatRateCanBeChanged()
{
    strategy.FlatRate = decimal.MinusOne;

    strategy.FlatRate.Should().Be(decimal.MinusOne);
}
// . . .
public class WorldWideShippingStrategy : ShippingStrategy
{
    public WorldWideShippingStrategy(decimal flatRate)
        : base(flatRate)
    {

    }
```

```
    public decimal FlatRate
    {
        get
        {
            return flatRate;
        }
        set
        {
            flatRate = value;
        }
    }
}
```

Although the subclass reuses the base class's constructor and guard clause, it does not maintain the data invariant and therefore breaks the Liskov substitution principle. The unit test proves that clients are able to set the value to a negative number, which should be disallowed by the class if it is to correctly protect its data invariants.

Listing 7-13 shows that when the base class is reworked to disallow direct write access to the flat rate field, the invariant is properly honored by the subclass. This is a very common pattern whereby fields are private but have protected or public properties that contain guard clauses to protect the invariants.

LISTING 7-13 The base class allows the subclass write access to the field only through the guarded property setter.

```
public class WorldWideShippingStrategy : ShippingStrategy
{
    public WorldWideShippingStrategy(decimal flatRate)
        : base(flatRate)
    {

    }

    public new decimal FlatRate
    {
        get
        {
            return base.FlatRate;
        }
        set
        {
            base.FlatRate = value;
        }
    }
}
// . . .
public class ShippingStrategy
{
```

```
    public ShippingStrategy(decimal flatRate)
    {
        if (flatRate <= decimal.Zero)
            throw new ArgumentOutOfRangeException("flatRate", "Flat rate must be positive
  and non-zero");

        this.flatRate = flatRate;
    }

    protected decimal FlatRate
    {
        get
        {
            return flatRate;
        }
        set
        {
            if (value <= decimal.Zero)
                throw new ArgumentOutOfRangeException("value", "Flat rate must be positive
  and non-zero");

            flatRate = value;
        }
    }
}
```

Tightening the visibility of the field and instead providing access only through the property setter protects the invariant with a guard clause. Doing this at subclass level is also preferable because it means that all future subclasses are absolved of this responsibility and simply cannot directly write to the field at all.

A new unit test can be created that asserts this new behavior, as shown in Listing 7-14.

LISTING 7-14 With the invariant maintained, this unit test passes.

```
[Test]
public void ShippingFlatRateCannotBeSetToNegativeNumber()
{
    strategy.Invoking(s => s.FlatRate = decimal.MinusOne)
        .ShouldThrow<ArgumentOutOfRangeException>("Flat rate must be positive and non-
  zero")
        .And.ParamName.Should().Be("value");
}
```

If a client tries to set the `FlatRate` property to a negative value, or even to zero, the guard clause prevents the assignment and an `ArgumentOutOfRangeException` is thrown.

Code contracts

Throughout the previous section, the guard clauses that formed the basis of the contracts were all written in long form, using `if` statements and exceptions. It is worth exploring an alternative to these manual guard clauses: code contracts.

Previously a separate library, code contracts were integrated into the .NET Framework 4.0 main libraries. In addition to being easier to read, write, and comprehend than manual guard clauses, code contracts bring with them the possibility of using static verification and automatic generation of reference documentation.

With static contract verification, code contracts are able to check for contract violations without executing the application. This helps expose implicit contracts such as null dereferences and problems with array bounds, in addition to the explicitly coded contracts shown throughout this section.

Generating reference documentation relating to the contract of a method or class is important because client code has no other way of knowing the exp ectations. When more detail is included in the XML comments that form the documentation to methods and classes, clients can view the expectations via IntelliSense. This makes working with classes that use contracts a bit easier.

Preconditions

Preconditions can be written succinctly by using code contracts. You will need to include the `System.Diagnostics.Contracts` namespace, which is part of the `mscorlib.dll` and so should not need an additional assembly reference. The static `Contract` class provides the majority of the functionality that is required to implement contracts.

Note If you make the decision to use code contracts, the static `Contract` class will permeate throughout almost all of your code base. This is less of a problem than it is with most static references because code contracts are ubiquitous infrastructure that, it is assumed, will not be removed or replaced. Thus, it is a significant undertaking to undo the decision to use code contracts, and it is best to use them from the outset of a project, or not at all.

Listing 7-15 shows the declarative nature of a code contract precondition.

LISTING 7-15 The `System.Diagnostics.Contracts` namespace can provide guard clauses to methods.

```
using System.Diagnostics.Contracts;

public class ShippingStrategy
{
    public decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
    {
        Contract.Requires(packageWeightInKilograms > 0f);
        Contract.Requires(packageDimensionsInInches.X > 0f && packageDimensionsInInches.Y
  > 0f);

        return decimal.MinusOne;
    }
}
```

The `Contract.Requires` method accepts a Boolean predicate value. This represents the state that the method requires in order to proceed. Note that this is the exact opposite of the predicate used in an `if` statement in manual guard clauses. In that case, the clauses were checking for state that was invalid before throwing an exception. With code contracts, the predicate is closer to an assertion: that the Boolean value must return `true`, otherwise the contract fails. This example requires that the `packageWeightInKilograms` parameter is non-zero and positive and that the `packageDimensionsInInches` parameter is non-zero and positive for both its X and Y properties.

This version of the `Contract.Requires` method throws an exception when the contract predicate is not met, but the type of exception is a `ContractException`, which does not match the expected exception in the existing unit tests. Therefore, they fail.

```
Expected System.ArgumentOutOfRangeException because Package dimension must be positive and non-
  zero, but found System.Diagnostics.Contracts.__ContractsRuntime+ContractException with message
  "Precondition failed: packageDimensionsInInches.X > 0f && packageDimensionsInInches.Y > 0f"
```

Furthermore, if you run this example while passing in an invalid value for one of the parameters, you will get the message shown in Figure 7-2. This informs you that you have not properly configured code contracts for use.

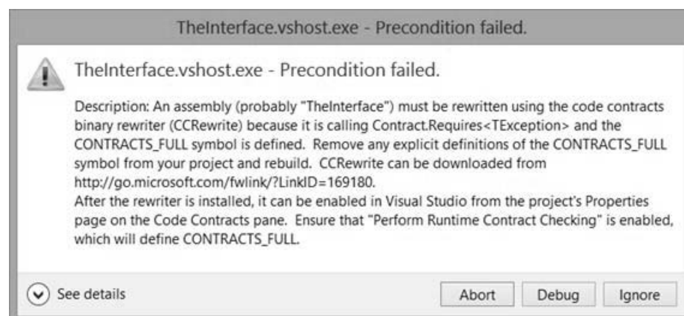

FIGURE 7-2 Code contracts must be configured before use.

The property pages of each project include a Code Contracts tab on which you can configure code contracts. A minimal working setup is shown in Figure 7-3.

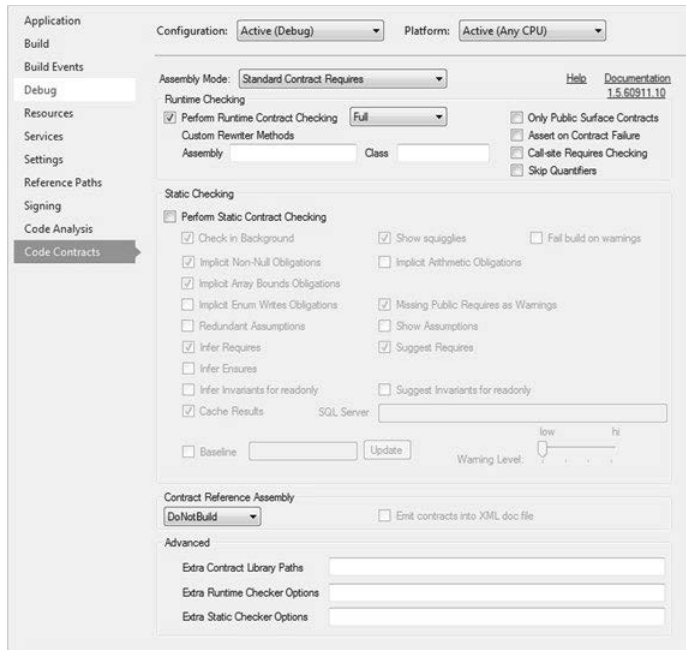

FIGURE 7-3 The property pages for code contracts contain a lot of settings.

When they are configured correctly, the contract preconditions can be rewritten to use an alternative version of the `Contract.Requires` method. Listing 7-16 shows this version.

LISTING 7-16 This version of the `Requires` method accepts the type of the exception to be thrown.

```
public class ShippingStrategy
{
    public decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
    {
        Contract.Requires<ArgumentOutOfRangeException>(packageWeightInKilograms > 0f,
  "Package weight must be positive and non-zero");
        Contract.Requires<ArgumentOutOfRangeException>(packageDimensionsInInches.X > 0f &&
  packageDimensionsInInches.Y > 0f, "Package dimensions must be positive and non-zero");

        return decimal.MinusOne;
    }
}
```

This generic version of the `Requires` method accepts the type of exception that you would like the contract to throw when the predicate fails. This, along with the exception message included in a subsequent method parameter, will cause the existing unit tests to pass.

Postconditions

Code contracts can similarly provide a shortcut to defining postconditions. The Contract static class contains an Ensures method that is the postcondition complement to the precondition's Requires method. This method also accepts a Boolean predicate that must be true in order to progress through to the return statement. It is worth noting that the return statement must be the only line that follows a call to Contract.Ensures. This makes intuitive sense because, otherwise, it would be possible to further modify state in a way that might break the postcondition.

Listing 7-17 reiterates the ShippingCostMustBePositive unit test and includes a rewritten CalculateShippingCost implementation that uses the Contract.Ensures method as a postcondition.

LISTING 7-17 The Ensures method creates a postcondition that should be true on exiting the method.

```
[Test]
public void ShippingCostMustBePositive()
{
    strategy.CalculateShippingCost(1, ValidSize, null)
        .Should().BeGreaterThan(decimal.MinusOne);
}
// . . .
public class ShippingStrategy
{
    public decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
    {
        Contract.Requires<ArgumentOutOfRangeException>(packageWeightInKilograms > 0f,
  "Package weight must be positive and non-zero");
        Contract.Requires<ArgumentOutOfRangeException>(packageDimensionsInInches.X > 0f &&
  packageDimensionsInInches.Y > 0f, "Package dimensions must be positive and non-zero");

        Contract.Ensures(Contract.Result<decimal>() > 0m);

        return decimal.MinusOne;
    }
}
```

The predicate in this example is a bit different from the ones in prior examples and demonstrates a common use of the postcondition: testing that a return value is valid. Checking that the shipping cost is positive (and, in fact, non-negative) requires knowledge of the return value. The return value is often, but not always, a local variable that is declared and defined within the method. You could trivially assert that the value you are returning is greater than zero, but this is not really foolproof. To access the value that is actually returned from the method, you can use the Contract.Result method to retrieve it. This generic method accepts the return type of the method and returns whichever result is eventually returned by the method. This is how you can ensure that no subsequent lines can replace a valid value with an invalid value without the postcondition failing and an exception being thrown.

Data invariants

It is common for each method in a class to contain its own preconditions and postconditions, but data invariants relate to the class as a whole. Code contracts allow you to create a private method on the class that contains declarative definitions of the class's invariants.

Each invariant is defined by another method of the `Contract` static class, as Listing 7-18 shows.

LISTING 7-18 Data invariants can be protected by a method dedicated to the purpose.

```
public class ShippingStrategy
{
    public ShippingStrategy(decimal flatRate)
    {
        this.flatRate = flatRate;
    }

    [ContractInvariantMethod]
    private void ClassInvariant()
    {
        Contract.Invariant(this.flatRate > 0m, "Flat rate must be positive and non-zero");
    }

    protected decimal flatRate;
}
```

The `Contract.Invariant` method follows the same pattern as the `Requires` and `Ensures` methods in that it accepts a Boolean predicate that must be true in order to satisfy the contract. In this example, there is also a second string parameter provided that describes the fault if this contract fails to be met and the invariant is unprotected. The client is allowed to make as many calls to the `Invariant` method as necessary, so it is best to break the invariants down to their most granular, rather than logically AND them all together with the && operator. This gives you the maximum benefit of knowing exactly which data invariant has been broken.

If this were a normal private method, you would be obliged to call the method at the start and end of every method, to ensure that the invariants were correctly protected. Luckily, you can have code contracts do this on your behalf by marking the method with the `ContractInvariantMethodAttribute`. Remember that attributes do not require the `Attribute` suffix, so this has been shortened in the example to `ContractInvariantMethod`. This flags the method as one that code contracts must call when entering and leaving a method, to confirm that the class's data invariants are not being violated. The prerequisites for marking a method as a `ContractInvariantMethod` are that it must return `void` and accept no arguments. However, it can be public or private, and you can choose any name to describe the method. Classes can have more than one `ContractInvariantMethod`, so logically grouping them is also possible. The body of the method must only make calls to the `Contract.Invariant` method.

Interface contracts

The final feature of code contracts to be covered here is that of interface contracts. So far, you have embedded all of your calls to `Contract.Requires`, `Contract.Ensures`, and `Contract.Invariant` in the class implementation itself. As has been mentioned, the static nature of the `Contract` class makes this code ubiquitous and difficult to remove or change in favor of an alternative library in the future. This is somewhat contrary to the adaptive codebase that is the ideal, but some infrastructural concessions are justifiable for pragmatic reasons.

A more immediate concern is the drop in readability that occurs when code contracts are liberally applied to classes. In fact, this is not really a fault of code contracts but a result of diligently applying contracts in general. Preconditions, postconditions, and data invariants are naturally implemented in code, but this code tends to increase the noise-to-signal ratio.

An interface contract, such as that shown in Listing 7-19 for the ongoing `ShippingStrategy` example, can alleviate this problem in addition to providing another helpful feature.

LISTING 7-19 A dedicated class can define preconditions, postconditions, and invariants for every implementation of an interface.

```
[ContractClass(typeof(ShippingStrategyContract))]
interface IShippingStrategy
{
    decimal CalculateShippingCost(
        float packageWeightInKilograms,
        Size<float> packageDimensionsInInches,
        RegionInfo destination);
}
//. . .
[ContractClassFor(typeof(IShippingStrategy))]
public abstract class ShippingStrategyContract : IShippingStrategy
{
    public decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
    {
        Contract.Requires<ArgumentOutOfRangeException>(packageWeightInKilograms > 0f,
  "Package weight must be positive and non-zero");
        Contract.Requires<ArgumentOutOfRangeException>(packageDimensionsInInches.X > 0f &&
  packageDimensionsInInches.Y > 0f, "Package dimensions must be positive and non-zero");

        Contract.Ensures(Contract.Result<decimal>() > 0m);

        return decimal.One;
    }

    [ContractInvariantMethod]
    private void ClassInvariant()
    {
        Contract.Invariant(flatRate > 0m, "Flat rate must be positive and non-zero");
    }
}
```

For interface contracts, you of course need an interface to work with. In this example, the `CalculateShippingCost` method has been extracted into its own `IShippingStrategy` interface. It is this interface, rather than a single implementation, that is going to have the contracts applied. This is an important departure from the previous examples because it means that all implementations of this interface will acquire the applied contracts. This is how you can enhance a simple interface that provides few instructions for implementation and use, to give it more powerful requirements and assurances.

When writing an interface contract, you also need a class that is going to implement the methods of the interface but only fill them with uses of the `Contract.Requires` and `Contract.Ensures` methods. The abstract `ShippingStrategyContract` provides this functionality and looks like the prior examples, but what the prior examples lacked was the real functionality of the method. Even in production code, this is the limit of the code contained in a contract class. There is also a `ContractInvariantMethod` to house any calls to `Contract.Invariant`, just as if this class were the real implementation.

To link the interface to the contract class implementation, you unfortunately need a two-way reference via an attribute. This is somewhat unfortunate because it adds noise to the interface, which it would be nice to avoid. Nevertheless, by marking the interface with the `ContractClass` attribute and the contract class with the `ContractClassFor` attribute, you can write your preconditions, postconditions, and data invariant protection code once and have it apply to all subsequent implementations of the interface. Both the `ContractClass` and `ContractClassFor` attributes accept a `Type` argument. The `ContractClass` is applied to the interface and has the contract class type passed in, whereas the `ContractClassFor` is applied to the contract class and has the interface type passed in.

This concludes the introduction to code contracts and the foray into the Liskov substitution principle's rules relating to contracts. One final important point needs to be emphasized. Whether they are implemented manually or by using code contracts, if a precondition, postcondition, or invariant fails, *clients should not catch the exception*. Catching an exception is an action that indicates that the client can recover from this situation, which is seldom likely or perhaps even possible when a contract is broken. The ideal is that all contract violations will happen during functional testing and that the offending code will be fixed before shipping. This is why it is so important to unit test contracts. If a contract violation is not fixed before shipping and an end user is unfortunate enough to trigger an exception, it is most likely the best course of action to force the application to close. It is advisable to allow the application to fail because it is now in a potentially invalid state. For a web application, this will mean that the global error page is displayed. For a desktop application, the user can be shown a friendly message and be given a chance to report the problem. In any and all cases, a log should be made of the exception, with full stack trace and as much context as possible.

The next section covers the rest of the LSP's rules—those that apply to covariance and contravariance.

Covariance and contravariance

The remaining rules of the Liskov substitution principle all relate to covariance and contravariance. Generally, *variance* is a term applied to the expected behavior of subtypes in a class hierarchy containing complex types.

Definitions

As previously demonstrated, it is important to cover the basics of this topic before diving in to the specifics of the LSP's requirements for variance.

Covariance

Figure 7-4 shows a very small class hierarchy of just two types: the generically named `Supertype` and `Subtype`, which are conveniently named after their respective roles in the inheritance structure. `Supertype` defines some fields and methods that are inherited by `Subtype`. `Subtype` enhances the `Supertype` by defining its own fields and methods.

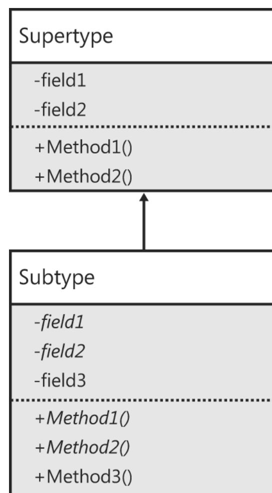

FIGURE 7-4 `Supertype` and `Subtype` have a parent/child relationship in this class hierarchy.

Polymorphism is the ability of a subtype to be treated as if it were an instance of the supertype. Thanks to this feature of object-oriented programming, which C# supports, any method that accepts an instance of `Supertype` will also be able to accept an instance of `Subtype` without any casting required by either the client or service code, and also without any type sniffing by the service. To the service, it has been handed an instance of `Supertype`, and this is the only fact it is concerned with. It doesn't care what specific subtype has been handed to it.

Variance enters the discussion when you introduce another type that might use `Supertype` and/or `Subtype` through a generic parameter.

Figure 7-5 is a visual explanation of the concept of *covariance*. First, you define a new interface called `ICovariant`. This interface is a generic of type T and contains a single method that returns this type, T. Because the `out` keyword is used before the generic type argument T, this interface is well named because it exhibits covariant behavior.

The second half of the class diagram details a new inheritance hierarchy that has been created thanks to the covariance of the `ICovariant` interface. By plugging in the values for the `Supertype` and `Subtype` classes that were defined previously, `ICovariant<Supertype>` becomes a supertype for the `ICovariant<Subtype>` interface.

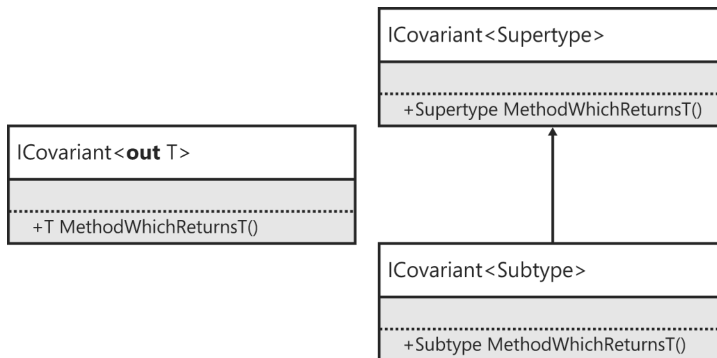

FIGURE 7-5 Due to covariance of the generic parameter, the base-class/subclass relationship is preserved.

Polymorphism applies here, just as it did previously, and this is where it gets interesting. Thanks to covariance, whenever a method requires an instance of `ICovariant<Supertype>`, you are perfectly at liberty to provide it with an instance of `ICovariant<Subtype>`, instead. This will work seamlessly thanks to the simultaneous interoperating of both covariance and polymorphism.

So far, this is of limited general use. To firm up this explanation, I'll move away from class diagrams and instructive type names to a more real-world scenario. Listing 7-20 shows a class hierarchy between a general `Entity` base class and a specific `User` subclass. All `Entity` types inherit a GUID unique identifier and a string name, and each `User` has an `EmailAddress` and a `DateOfBirth`.

LISTING 7-20 In this small domain, a `User` is a specialization of the `Entity` type.

```
public class Entity
{
    public Guid ID { get; private set; }

    public string Name { get; private set; }
}
// . . .
public class User : Entity
{
    public string EmailAddress { get; private set; }

    public DateTime DateOfBirth { get; private set; }
}
```

This is directly analogous to the Supertype/Subtype example, but with a more directed purpose. This small domain is going to have the Repository pattern applied to it. The Repository pattern provides you with an interface for retrieving objects as if they were in memory but that could realistically be loaded from a very different storage medium. Listing 7-21 shows an EntityRepository class and its UserRepository subclass.

LISTING 7-21 Without involving generics, all inheritance in C# is invariant.

```
public class EntityRepository
{
    public virtual Entity GetByID(Guid id)
    {
        return new Entity();
    }
}
// . . .
public class UserRepository : EntityRepository
{
    public override User GetByID(Guid id)
    {
        return new User();
    }
}
```

This example is not the same as that previously described because of one key difference: in the absence of generic types, C# is not covariant for method return types. In fact, a compilation error is generated due to an attempt to change the return type of the GetByID method in the subclass to match the User class.

```
error CS0508: 'SubtypeCovariance.UserRepository.GetByID(System.Guid)': return type must be
  'SubtypeCovariance.Entity' to match overridden member
  'SubtypeCovariance.EntityRepository.GetByID(System.Guid)'
```

Perhaps experience tells you that this will not work, but the reason is a lack of covariance in this scenario. If C# supported covariance for general classes, you would be able to enforce the change of return type in the UserRepository. Because it does not, you have only two options. You can amend the UserRepository.GetByID method's return type to be Entity and use polymorphism to allow you to return a User in its place. This is dissatisfying because clients of the UserRepository would have to downcast the return type from an Entity type to a User type, or they would have to sniff for the User type and execute specific code if the expected type was returned.

Instead, you should redefine EntityRepository as a generic class that requires the Entity type it intends to operate on via a generic type argument. This generic parameter can be marked out, thus covariant, and the UserRepository subclass can specialize its parent base class for the User type. Listing 7-22 exemplifies this.

LISTING 7-22 Make base classes generic to take advantage of covariance and allow subclasses to override the return type.

```
public interface IEntityRepository<TEntity>
    where TEntity : Entity
{
    TEntity GetByID(Guid id);
}
// . . .
public class UserRepository : IEntityRepository<User>
{
    public User GetByID(Guid id)
    {
        return new User();
    }
}
```

Rather than maintaining `EntityRepository` as a concrete class that can be instantiated, this code has converted it into an interface that removes the default implementation of `GetByID`. This is not entirely necessary, but the benefits of clients depending on interfaces rather than implementations have been demonstrated consistently, so it is a sensible reinforcement of that policy.

Note also that there is a `where` clause applied to the generic type parameter of the `EntityRepository` class. This clause prevents subclasses from supplying a type that is not part of the `Entity` class hierarchy, which would have made this new version more permissive than the original implementation.

This version prevents the need for `UserRepository` clients to mess around with downcasting because they are guaranteed to receive a `User` object, rather than an `Entity` object, and yet the inheritance of `EntityRepository` and `UserRepository` is preserved.

Contravariance

Contravariance is a similar concept to covariance. Whereas covariance relates to the treatment of types that are used as return values, *contravariance* relates to the treatment of types that are used as method parameters.

Using the same `Supertype` and `Subtype` class hierarchy as previously discussed, Figure 7-6 explores the relationship between types that are marked as contravariant via generic type parameters.

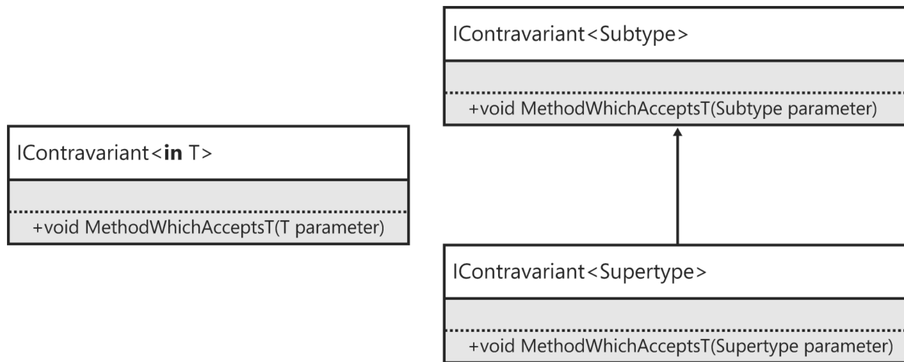

FIGURE 7-6 Due to contravariance of the generic parameter, the base-class/subclass relationship is inverted.

The `IContravariant` interface defines a method that accepts a single parameter of the type dictated by the generic parameter. Here, the generic parameter is marked with the `in` keyword, meaning that it is contravariant.

The subsequent class hierarchy can be inferred, indicating that the inheritance hierarchy has been inverted: `IContravariant<Subtype>` becomes the superclass, and `IContravariant<Supertype>` becomes the subclass. This seems strange and counterintuitive, but it will soon become apparent why contravariance exhibits this behavior—and why it is useful.

In Listing 7-23, the .NET Framework `IEqualityComparer` interface is provided for reference and an application-specific implementation is created. The `EntityEqualityComparer` accepts the previous `Entity` class as a parameter to the `Equals` method. The details of the comparison are not relevant, but a simple identity comparison is used.

LISTING 7-23 The `IEqualityComparer` interface allows the definition of function objects like `EntityEqualityComparer`.

```
public interface IEqualityComparer<in TEntity>
    where TEntity : Entity
{
    bool Equals(TEntity left, TEntity right);
}
// . . .
public class EntityEqualityComparer : IEqualityComparer<Entity>
{
    public bool Equals(Entity left, Entity right)
    {
        return left.ID == right.ID;
    }
}
```

The unit test in Listing 7-24 explores the affect that contravariance has on the `EntityEquality-Comparer`.

LISTING 7-24 Contravariance inverts class hierarchies, allowing a more general comparer to be used wherever a more specific comparer is requested.

```
[Test]
public void UserCanBeComparedWithEntityComparer()
{
    SubtypeCovariance.IEqualityComparer<User> entityComparer = new
  EntityEqualityComparer();
    var user1 = new User();
    var user2 = new User();
    entityComparer.Equals(user1, user2)
        .Should().BeFalse();
}
```

Without contravariance—the innocent-looking `in` keyword applied to generic type parameters—the following error would be shown at compile time.

```
error CS0266: Cannot implicitly convert type 'SubtypeCovariance.EntityEqualityComparer' to
  'SubtypeCovariance.IEqualityComparer<SubtypeCovariance.User>'. An explicit conversion exists
  (are you missing a cast?)
```

There would be no type conversion from `EntityEqualityComparer` to `IEqualityComparer-<User>`, which is intuitive because `Entity` is the supertype and `User` is the subtype. However, because the `IEqualityComparer` supports contravariance, the existing inheritance hierarchy is inverted and you are able to assign what was originally a less specific type to a more specific type via the `IEqualityComparer` interface.

Invariance

Beyond covariant or contravariant behavior, types are said to be invariant. This is not to be confused with the term *data invariant* used earlier in this chapter as it relates to code contracts. Instead, invariant in this context is used to mean "not variant." If a type is not variant at all, no arrangement of types will yield a class hierarchy. Listing 7-25 uses the `IDictionary` generic type to demonstrate this fact.

LISTING 7-25 Some generic types are neither covariant or contravariant. This makes them *invariant*.

```
[TestFixture]
public class DictionaryTests
{
    [Test]
    public void DictionaryIsInvariant()
    {
        // Attempt covariance...
        IDictionary<Supertype, Supertype> supertypeDictionary = new Dictionary<Subtype,
  Subtype>();

        // Attempt contravariance...
        IDictionary<Subtype, Subtype> subtypeDictionary = new Dictionary<Supertype,
  Supertype>();
    }
}
```

The first line of the `DictionaryIsInvariant` test method attempts to assign a dictionary whose key and value parameters are of type `Subtype` to a dictionary whose key and value parameters are of type `Supertype`. This will not work because the `IDictionary` type is not covariant, which would preserve the class hierarchy of `Subtype` and `Supertype`.

The second line is also invalid, because it attempts the inverse: to assign a dictionary of `Supertype` to a dictionary of `Subtype`. This fails because the `IDictionary` type is not contravariant, which would invert the class hierarchy of `Subtype` and `Supertype`.

The fact that the `IDictionary` type is neither covariant nor contravariant leads to the conclusion that it must be invariant. Indeed, Listing 7-26 shows how the `IDictionary` type is declared, and you can tell that there is no reference to the `out` or `in` keywords that would specify covariance and contravariance, respectively.

LISTING 7-26 None of the generic parameters of the `IDictionary` interface are marked with `in` or `out`.

```
public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>,
  IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
```

As previously proven for the general case—that is, without generic types—C# is invariant for both method parameter types and return types. Only when generics are involved is variance customizable on a per-type basis.

Liskov type system rules

Now that you have a grounding in variance, this section can circle back and relate all of this to the Liskov substitution principle. The LSP defines the following rules, two of which relate directly to variance:

- There must be contravariance of the method arguments in the subtype.
- There must be covariance of the return types in the subtype.
- No new exceptions are allowed.

Without contravariance of method arguments and covariance of return types, you cannot write code that is LSP-compliant.

The third rule stands alone as not relating to variance and bears its own discussion.

No new exceptions are allowed

This rule is more intuitive than the other LSP rules that relate to the type system of a language. First, you should consider: what is the purpose of exceptions?

Exceptions aim to separate the reporting of an error from the handling of an error. It is common for the reporter and the handler to be very different classes with different purposes and context. The exception object represents the error that occurred through its type and the data that it carries with it. Any code can construct and throw an exception, just as any code can catch and respond to an exception. However, it is recommended that an exception only be caught if something meaningful can be done at that point in the code. This could be as simple as rolling back a database transaction or as complex as showing users a fancy user interface for them to view the error details and to report the error to the developers.

It is also often inadvisable to catch an exception and silently do nothing, or catch the general `Exception` base type. Both of these two scenarios together are even more discouraged. With the latter scenario, you end up attempting to catch and respond to *everything*, including exceptions that you realistically have no meaningful way of recovering from, like `OutOfMemoryException`, `StackOverflowException`, or `ThreadAbortException`. You could improve this situation by ensuring that you always inherit your exceptions from `ApplicationException`, because many unrecoverable exceptions inherit from `SystemException`. However, this is not a common practice and relies on third-party libraries to also follow this practice.

Listing 7-27 shows two exceptions that have a sibling relationship in the class hierarchy. It is important to note that this precludes the ability to create a catch block specifically targeting one of the exception types and to intercept both types of exception.

LISTING 7-27 Both of these exceptions are of type `Exception`, but neither inherits from the other.

```
public class EntityNotFoundException : Exception
{
    public EntityNotFoundException()
        : base()
    {

    }

    public EntityNotFoundException(string message)
        : base(message)
    {

    }
}
//. . .
public class UserNotFoundException : Exception
{
    public UserNotFoundException()
        : base()
    {

    }

    public UserNotFoundException(string message)
        : base(message)
    {

    }
}
```

Instead, in order to catch both an `EntityNotFoundException` and a `UserNotFoundException` with a single catch block, you would have to resort to catching the general `Exception`, which is not recommended.

This problem is exacerbated in the potential code taken from the `EntityRepository` and `UserRepository` classes, as shown in Listing 7-28.

LISTING 7-28 Two different implementations of an interface might throw different types of exception.

```
public Entity GetByID(Guid id)
{
    Contract.Requires<EntityNotFoundException>(id != Guid.Empty);

    return new Entity();
}
//. . .
public User GetByID(Guid id)
{
    Contract.Requires<UserNotFoundException>(id != Guid.Empty);

    return new User();
}
```

Both of these classes use code contracts to assert a precondition: that the provided `id` parameter must not be equal to `Guid.Empty`. Each uses its own exception type if the contract is violated. Think for a second about the impact that this would have on a client using the repository. The client would need to catch both kinds of exception and could not use a single catch block to target both exceptions without resorting to catching the `Exception` type. Listing 7-29 shows a unit test that is a client to these two repositories.

LISTING 7-29 This unit test will fail because a `UserNotFoundException` is not assignable to an `EntityNotFoundException`.

```
[TestFixture(typeof(EntityRepository), typeof(Entity))]
[TestFixture(typeof(UserRepository), typeof(User))]
public class ExceptionRuleTests<TRepository, TEntity>
    where TRepository : IEntityRepository<TEntity>, new()
{
    [Test]
    public void GetByIDThrowsEntityNotFoundException()
    {
        var repo = new TRepository();
        Action getByID = () => repo.GetByID(Guid.Empty);

        getByID.ShouldThrow<EntityNotFoundException>();
    }
}
```

This unit test fails because the `UserRepository` does not, as required, throw an `EntityNotFoundException`. If the `UserNotFoundException` was a subclass of the type `EntityNotFoundException`, this test would pass and a single catch block could guarantee catching both kinds of exception.

This becomes a problem of client maintenance. If the client is using an interface as a dependency and calling methods on that interface, it should not know anything about the classes behind that interface. This is a return to the argument concerning the Entourage anti-pattern versus the Stairway pattern. If new exceptions that are not part of an expected exception class hierarchy are introduced, clients must start referencing these exceptions directly. And—even worse—clients will have to be updated whenever a new exception type is introduced.

Instead, it is important that every interface have a unifying base class exception that conveys the necessary information about an error from the exception reporter to the exception handler.

Conclusion

On the surface, the Liskov substitution principle is one of the more complex facets of the SOLID principles. It requires a foundational knowledge of both contracts and variance to build rules that guide you toward more adaptive code.

By default, interfaces do not convey rules for preconditions or postconditions to clients. Creating guard clauses that halt the application at run time serves to further narrow the allowed range of valid values for parameters. The LSP provides guidelines such that each subclass in a class hierarchy cannot strengthen preconditions or weaken postconditions.

Similarly, the LSP suggests rules for variance in subtypes. There should be contravariance of method arguments in subtypes and covariance of return values in subtypes. Additionally, any new exception that is introduced, perhaps with the creation of a new interface implementation, should inherit from an existing base exception. To do otherwise would be to potentially cause an existing client to miss the catch—effectively to fumble the exception and allow it to cause an application crash.

If the LSP is violated with respect to these rules, it becomes harder for clients to treat all types in a class hierarchy the same. Ideally, clients would be able to hold a reference to a base type or interface and not alter its own behavior depending on the concrete subclass that it is actually using at run time. Such mixed concerns create dependencies between sections of the code that are better kept separate. Any violation of the LSP should be considered technical debt and, as demonstrated in prior chapters, this debt should be paid off sooner rather than later.

This page intentionally left blank

Index

A

AAA (Arrange, Act, Assert) pattern
  bug fixes and, 208
  described, 126–130
  unit tests and, 132
abstract classes, 155, 156
abstract methods, 211
abstraction, refactoring for, 177–183
abstractions
  leaky, 85
  maladaptive code and, 37, 38
acceptance criteria
  product owner responsibility for, 7
  user stories and, 14
Act phase of unit tests, 127
actions, MVC applications and, 312
ActLike<T>() method, 116
acyclic digraphs, 53
adapter classes, 181
Adapter pattern, 109–113, 192, 200, 261
adapters, 289
adaptive code
  described, xv, xvi, 37
  single responsibility principle and, 169, 206
affinity estimation, 31
Agile Manifesto, 4
Agile processes and practices
  described, 1, 2
  Scrum and, 4
  vs. waterfall, xv
aliasing shorter names for longer types, 296
ambient containers, 307
anti-patterns. *See also* patterns
  described, 56
  Entourage, 63–65, 284, 311
  Illegitimate Injection, 310–312
  Interface Soup, 267, 268
  IsNull property, 105–107
  Service Locator, 306–310
  solving with Stairway pattern, 85
AOP (aspect-oriented programming), 88, 89, 255
API layer, 84
ApiController class, 74
apocalyptic defects
  described, 16
  swarming on, 18
AppFabric AutoStart, 74
application configuration files
  assembly resolution process and, 68
  mapping interfaces to implementations, 296
applications, installing with Chocolatey, 80
ArgumentNullException, 144
arguments, variance and, 218
Arrange, Act, Assert (AAA) pattern
  bug fixes and, 208
  described, 126–130
  unit tests and, 132
artifacts
  backlogs, 27–29
  charts and metrics, 22–27
  daily Scrum, 31–33
  release planning, 29–31
  Scrum board, 9
  Scrum calendar, 36
  sprint demo, 33
  sprint retrospective, 34, 35
aspect-oriented programming (AOP), 88, 89, 255
ASP.NET MVC. *See* MVC (Model-View-Controller)
ASP.NET Web API, 74
assemblies
  cyclic dependencies and, 54
  dependencies between, defined, 44
  Entourage anti-pattern and, 63

assemblies *(continued)*
- exceptions when loading, 48
- layering and, 81–84
- process for loading, 47
- refactoring for abstraction and, 178
- resolution process, 67, 68
- sample application showing dependencies, 44–51

asserting expected behavior in unit tests, 128
assigning tasks, 14
asymmetric layering, 89–91
asynchronous decorators, 200–202
asynchronous methods, 200
audit tracking, decorators for, 260
authorization, interface segregation and, 274
avatars for Scrum boards, 17

B

backlogs
- described, 27
- product, 3, 27, 28
- sprint, 3, 28
- swimlane for, on Scrum board, 18
- user stories and, 328–335

base type, Liskov substitution principle and, 218
BDUF (Big Design Up Front), 9
behavior, expected
- asserting in unit tests, 128
- Liskov substitution principle, 217
- naming test methods and, 132
- refactoring code and, 152
- test-driven development and, 130
- variance and, 239

behavioral errors, 16
Big Design Up Front (BDUF), 9
binding to assemblies, 68, 69
blackbox reuse, 110
Bloch, Joshua, 212
branching
- correlation with defect count, 41
- decorators, 193, 194
- source control and, 379–383

brownfield projects, 27
bug fixes. *See also* refactoring code
- open/closed principle and, 208, 209
- writing tests for, 148, 149

build process, 384, 385
burndown charts, 24, 25
business logic layer, 86, 87

C

C#, duck-typing and, 113–118
caching decorators, 257–261
calendars
- niko-niko, 32
- of meetings, for full sprint, 36

cards
- color schemes, 17
- features, 12, 13
- hierarchy of, on Scrum board, 11
- purpose of, on Scrum board, 10
- types, 10–17
- who creates, 17

Cascading Nulls, 108
catching exceptions, 238, 247–250
ceremonies, 4
chaining
- composite instances, 188
- dependencies, 52, 58, 291
- methods, 123, 124

ChannelFactory class, 70, 71
charts
- feature burnup, 25–27
- Scrum and, 22
- sprint burndown, 24, 25
- story points, 23
- velocity, 23, 24

chickens and pigs, 9
Chocolatey package management tool, 80, 81
CI (continuous integration)
- build process, 384
- server, unit test coverage and, 40

clarity, refactoring code for, 172–176
Class Adapter pattern, 109, 110
classes
- abstract, 155, 156
- Adapter, 109
- as adapters, 289
- changing behavior of without recompilation, 111–113
- ChannelFactory, 70, 71

ConfigurationManager, 287, 297
constructors and, 57
Contract, 232, 235, 236, 237
defined, 57
DiscoveryClient, 73
disposing of dependencies, 301
explicit interface implementation, 97–101
hierarchies, 217
implementation inheritance and, 210–213
implicit interface implementation, 99, 100
interfaces and, 94, 284
loosely coupled, 209
mixin, 118–122
multiple inheritance, 96
multiple interface implementation, 95
naming conventions, 178
NullUser, 105, 107
Order, 264
organizing with Stairway pattern, 65
polymorphism and, 101, 102
proxiable, 284
segregating interfaces and, 263–266, 267
single responsibility principle and, 169
Stopwatch, 198
as systems under test (SUTs), 126
tightly coupled, 209
client awareness, 209
clients
providing implementations to, 263–269
sample application showing dependencies, 44–51
software dependencies and, 44
cloning a repository, 382
closed for modification rule
exceptions to, 209
open/closed principle definition, 208
CLR (Common Language Runtime)
duck-typing and, 115–118
resolving assemblies and, 67
Cockburn, Alistair, 213
code. *See also* refactoring code; unit tests
backing up, 379
as client or service, 44
compiling, 384
contracts, 232–238
maladaptive, 37–41
production, 125, 130
code reviews. *See* peer reviews
code samples in this book, xxii, xxiii, 381
code smells
described, 57
inappropriate intimacy and, 59
refused bequest, 162
coding for change, 5
coding to an interface, 60
color schemes, Scrum boards and, 17
Command/Query Responsibility Segregation (CQRS), 90, 91, 275
command/query separation, 89, 90
comments for contracts of methods and classes, 232
Common Language Runtime (CLR)
duck-typing and, 115–118
resolving assemblies and, 67
compiling code, 384
components, 81
Composite pattern, 185–189
composition
as blackbox reuse, 110
root, dependency injection and, 306, 311
compromises. *See* technical debt
concessions. *See* technical debt
conditional expressions, replacing with polymorphism, 154–157
configuration files
assembly resolution process and, 68
mapping interfaces to implementations, 296
ConfigurationManager class, 287, 297
connection factories, 301, 302
Console.ReadKey(), 46
constants, replacing magic numbers with, 153, 154
constructing
dependent classes, 291
implementations, for clients, 263–268
object graphs, 287–293
constructors
classes and, 57
dependency injection and, 287–291
Illegitimate Injection and, 310–312
refactoring for abstraction and, 179
replacing with factory classes, 159
replacing with factory methods, 157–159
containers
ambient, 307
conventions and, 317–321

containers *(continued)*
injecting, 309
Inversion of Control (IoC), 291–295
Unity, 292, 293, 296, 297, 308
continuous integration (CI)
build process, 384, 385
server, unit test coverage and, 40
Contract class, 232, 235, 236, 237
Contract.Ensures method, 235
Contract.Invariant method, 236, 238
Contract.Requires method, 233, 234
contracts
code, 232–238
data invariants, 223, 224
vs. encapsulation, 224
postconditions and, 222, 223
preconditions and, 220–222
rules, 218, 225–231
contravariance, 242–244, 246
control, inversion of. *See* Inversion of Control (IoC)
controllers, dependency injection and, 312–317
conventions, dependency injection and, 317–321
correlation vs. causation, metrics and, 40
cosmetic issues, 16
cost of changes, xv
covariance, 239–243, 246
CQRS (Command/Query Responsibility Segregation), 90, 91, 275
cranes vs. skyhooks, 38, 39
create, read, update, and delete (CRUD) operations
interface segregation and, 251–260
two-layer architecture and, 85
cross-cutting concerns, 87, 88, 89, 255
CRUD (create, read, update, and delete) operations
interface segregation and, 251–260
two-layer architecture and, 85
cyclic dependencies, 53
cyclic digraphs, 53
cyclomatic complexity, unit testing and, 40, 41

D

daily Scrum meetings, 8, 31, 32, 33
data access layer, two-layer solutions and, 83–86
data invariants
code contracts and, 236
described, 223, 224
Liskov contract rules and, 229–231
debugging
assemblies, viewing loaded, 47
dependencies between assemblies, 67
Fusion log, 68–70
declarative vs. imperative registration, 295–297
declaring interfaces, 94–96
Decorator pattern
described, 184, 185
interfaces and, 251
decorators
asynchronous, 200–202
branching, 193, 194
caching, 257–261
composite, 185–188
CRUD interfaces and, 252–261
events, 203
lazy, 194, 195
logging, 195, 196
for multiple interfaces, 261–263
predicate, 189–193
profiling, 196–200
properties, 203
decoupling code by using Service Discovery, 73
default references, 49
defect cards, 16
defect fixes
open/closed principle and, 208, 209
writing tests for, 148, 149
defensive programming. *See* splitting interfaces
deferring construction to run time, 291
definition of done (DoD), 21, 22
delegates vs. interfaces, 193, 279
delegating
to abstractions, importance of, 169
to abstractions, refactoring for, 177–183
interface methods to methods of another object, 109
to interfaces, 212, 214
replacing inheritance with, 163–166
tasks to other methods, 173–176
delegation, pass-through, 256
demos, sprint, 33, 357, 358, 375, 376

dependencies
  chain of, 52, 58, 291
  combining decorators and, 263
  cyclic, 53–55
  defined, 44
  first party, 44–48
  framework, 48, 49
  interface inheritance and, 212, 214
  layering and, 83
  modeling, 51–55
  NuGet and, 77–81
  predicate decorators and, 190–192
  purpose of, 43
  refactoring for abstraction and, 178, 179
  resolving, 67–76
  sample application, 44–51
  services and, 70–77
  testability and, 59
  third party, 50, 51
  types of, 43
dependency digraphs, 51
dependency injection (DI). *See also* Inversion of Control (IoC)
  ASP.NET MVC and, 312–317
  classes that make use of, 61, 62
  composition root and, 311
  constructors and, 287–291
  controllers and, 312–317
  conventions and, 317–321
  described, 281
  method injection, 290
  object lifetime, 298–304
  Poor Man's Dependency Injection, 287–290
  property injection, 291
  Service Locator anti-pattern, 306–310
  task list application introduced, 281–289
  Windows Forms and, 315, 316
deployment packages, continuous integration and, 384
dereferencing null, 103, 108
design patterns. *See* patterns
developer role, Scrum process and, 326, 327
diamond inheritance problem, 96
DI (dependency injection)
  ASP.NET MVC and, 312–317
  classes that make use of, 61, 62
  composition root and, 311
  constructors and, 287–291
  controllers and, 312–317
  conventions and, 317–321
  described, 281
  method injection, 290
  object lifetime, 298–304
  Poor Man's Dependency Injection, 287–290
  property injection, 291
  Service Locator anti-pattern, 306–310
  task list application introduced, 281–289
  Windows Forms and, 315, 316
digital Scrum boards, 21
digraphs
  cyclic vs. acyclic, 53
  defined, 51
directed graphs, 51–55
discoverable services, 71–74
DiscoveryClient class, 73
DiscoveryEndpoint, 73
Dispose method, 293, 298–300, 304
distributed source control, 379–383
DLR (Dynamic Language Runtime), 115
domain model
  Command/Query Responsibility Segregation (CQRS), 90, 91, 275
  logic layer and, 87
DomainException, 146, 147
done
  definition of, 21, 22
  swimlane for, 18
duck-typing, 113–118
dynamic keyword, 115
Dynamic Language Runtime (DLR), 115
dynamic proxies, testing with, 140

E

edges, 51
effort, estimated vs. actual, 23, 35
encapsulating
  branching tests, 280
  null users, 107
  variant behavior with Strategy pattern, 112
encapsulation vs. contracts, 224
Ensures method, 235

Entourage anti-pattern
- dependency injection and, 284
- described, 63–65
- Illegitimate Injection and, 311
- solving with Stairway pattern, 85

epics vs. features, 13
estimating
- features, 29
- stories, 30, 31, 35
- story points, 330–335

events
- decorating, 203
- publishing and subscribing to, 261–263

exceptions
- catching, 238, 246–249
- enforcing preconditions with, 220, 221
- purpose of, 246, 371
- wrapping, 146, 149

expected behavior
- asserting in unit tests, 128
- Liskov substitution principle, 217
- naming test methods and, 132
- refactoring code and, 152
- test-driven development and, 130
- variance and, 239

ExpectedExceptionAttribute, 144, 145, 147
explicit interface implementation, 97–101
extension
- methods, 118–120
- open/closed principle and, 208
- points, 209–214
- points, adaptive code and, 322

external dependencies. *See* third-party software dependencies
external mocking frameworks, 140, 141
Extreme Programming (XP), user stories and, 13

F

factories
- connection, 301, 302
- injection, 299
- isolating, 304

factory classes, replacing constructors with, 159
Factory Isolation pattern, 304, 305
factory methods, replacing with constructors, 157–159
Factory pattern, 301, 302, 314, 345–347
failing unit tests
- bug fixes and, 208
- described, 130, 139
- writing for bug fixes, 148

fakes, testing with, 137–140
fast-track items, 18
features
- burnup charts, 25–27
- estimating, 29
- prioritizing, 29
- Scrum board cards and, 12, 13

fire-and-forget methods, 202
first-party software dependencies, 43, 44–48
fluent interfaces, 123, 124
foreach loops, CLR duck-typing support and, 116–118
framework dependencies, 48, 49, 66
Fusion log, 68–70

G

GAC (global assembly cache), 67, 68
generalizing specialists, 8
generic
- controllers, 265
- interfaces, 252, 265
- type parameters, 242–245

Git, 379–383
GitHub, 381
Given, When, Then pattern, 126
global assembly cache (GAC), 67, 68
goals, defining for sprints, 337
Goldilocks Zone, extension points and, 214
graphs
- modeling dependencies in, 51–55
- object, 287–291, 312, 320

green icon, as indicator of successful unit test, 130
greenfield projects, 27
guard clauses
- vs. code contracts, 232
- data invariants and, 223, 230, 231
- postconditions and, 222
- preconditions and, 220, 221, 226

H

Happy Path, need for further tests after finding, 143
hierarchies
  contravariance and, 242–244
  covariance and, 239–242
  inheritance, 217, 218
Hollywood Principle, 308
horizontal scaling
  tiers and, 83
  vs. vertical scaling, 373
horizontal swimlanes, 18
hosted services
  dependencies and, 70–77
  discovery of, 71–74
HTTP verbs, 75
Hype Cycle, 327

I

IAccountRepository interface, 137
IAction interface, 280
IComponent interface, 187, 194
IContravariant interface, 243
ICovariant interface, 240
IDisposable interface, 293, 299–302, 304
IFluentInterface, 123, 124
IFunction interfaces, 280
Illegitimate Injection, 310–312
imperative vs. declarative registration, 295–297
implementation inheritance
  designing for, 212, 213
  extension points and, 210, 211
  vs. interface inheritance, 212
implementations
  blackbox vs. whitebox reuse, 110
  explicit, 97–101
  implicit, 99, 100
  inability to enhance, 58
  vs. interfaces, 56, 124
  leaf, 267
  Register, Resolve, Release pattern and, 293–295
  separating from interfaces, 63–65, 84
implicit implementation, 99, 100
Impromptu Interface, 116
in keyword, 243, 244, 245
inappropriate intimacy, code smells and, 57, 59
inheritance
  Class Adapter pattern and, 109
  contravariance and, 242–244
  covariance and, 239–243
  exceptions, 246–249
  hierarchies, 217, 218
  implementation, 210–213
  interface, 212
  multiple, 96
  replacing with delegation, 163–166
  segregation and, 272–274
  as whitebox reuse, 110
initial backlog, 328–335
initialization method, 149, 150
injection
  constructor, 308
  of containers, 309
  factories, 299
  illegitimate, 310–312
injection, dependency
  ASP.NET MVC and, 312–317
  classes that make use of, 61, 62
  composition root and, 311
  constructors and, 287–291
  controllers and, 312–317
  conventions and, 317–321
  described, 281
  method injection, 290
  object lifetime, 298–304
  Poor Man’s Dependency Injection, 287–290
  via property injection, 291
  Service Locator anti-pattern and, 306
  task list application introduced, 281–289
  Windows Forms and, 315, 316
In-Progress swimlanes, 18
instantiating objects, code smells and, 57
IntelliSense, contracts and, 232
interface contracts, 237, 238
interface inheritance, extension points and, 212
interface keyword, 94
interface segregation
  authorization, 274
  caching decorator and, 257–261
  client construction, 263–268
  defined, 251

interface segregation *(continued)*
  example, 251–257
  splitting for architectural need, 275–279
  splitting for client need, 268–275
Interface Soup anti-pattern, 267, 268
interfaces. *See also* user interface
  assemblies and, 63, 84
  classes and, 94, 284
  code adaptability and, 284
  coding to, 60
  combining as mixins with Re-Motion Re-mix, 120–122
  Composite pattern and, 186
  contracts and, 219
  decorating multiple, 261–263
  defined, 57
  vs. delegates, 193, 251, 279
  described, 93
  explicit implementation, 97–101
  external dependencies and, 66
  fluent, 123, 124
  generic, 252
  IAccountRepository, 137
  IAction, 280
  IComponent, 187, 194
  IContravariant, 243
  ICovariant, 240
  IDisposable, 293
  IFunction, 280
  vs. implementations, 56, 124
  implementing in a single class, 266, 267
  IPaymentStrategy, 206
  IPredicate, 192, 280
  IServiceLocator, 306, 308
  ITargetInterface, 118, 122
  ITask, 279
  members and, 251
  mocking, 140–142
  organizing with Stairway pattern, 65, 284
  overuse of, 113
  providing to clients, 263–269
  refactoring toward abstraction and, 177–180
  Register, Resolve, Release pattern and, 293–295
  repository, 136
  splitting for architectural need, 275–279
  splitting for client need, 268–275
  syntax, 94–96
Invariant method, 236
invariants, 223, 224, 229–231, 236, 244, 245
  guard clauses and, 223
Inversion of Control (IoC)
  composition root and, 311
  containers and, 291–295
  described, 291, 292
  vs. Poor Man's Dependency Injection, 292
inverting class hierarchies with contravariance, 242–244
IoC (Inversion of Control). *See* Inversion of Control (IoC)
IPaymentStrategy interface, 206
IPredicate interface, 192, 280
IronPython, as dynamically typed language, 115
IServiceLocator interface, 306, 308
IsNull property anti-pattern, 105–107
isolating factories, 304, 305
ITargetInterface, 118, 122
ITask interface, 279
iterations. *See* sprints

J

JIT (just-in-time) model, 67

K

keywords, dynamic, 115
known endpoints, service references and, 70

L

lambda expressions
  Factory Isolation pattern and, 305
  mocking and, 141
  object lifetime and, 299
lambda factories, declarative registration and, 297
layering
  API layer, 84
  described, 81
  patterns, 82–87
  vs. tiers, 83
  wrapping exceptions and, 146

lazy decorators, 194, 195
Lazy<T>, 194
leaf implementations, 267
leaky abstractions, 85
libraries
  code contracts, 232
  Impromptu Interface, 116
  Log4Net, 181, 182
  Prism, 122
  Re-motion Re-mix, 118, 120–122
lifetime of objects, 298–304
line of best fit, 24
Liskov substitution principle (LSP)
  contravariance, 242–245
  covariance, 239–243
  described, 217, 218
  exceptions, 246–249
  invariance, 244, 245
  rules, 218, 225–231
log files, 68–70
Log4Net, 181, 182
logging decorators, 195, 196
logic layer, 86, 87
loops, 54, 55
loose mocks, 141
loosely coupled classes, 209
LSP (Liskov substitution principle)
  contravariance, 242–245
  covariance, 239–243
  described, 217, 218
  exceptions, 246–249
  invariance, 244, 245
  rules, 218, 225–231

M

magic numbers, replacing with constants, 153, 154
maladaptive code, 37–41
mapping
  composition root and, 311
  conventions and, 318
  dependency injection and, 321
  interfaces to implementations, 296
markdown transforms, 367–370
marketable features, 12
Martin, Robert C., 208
meetings
  for full sprint, 36
  planning, for sprint, 337–339
  retrospective, for sprint, 358–362
Mercurial vs. Git, 379
method injection, 290
methods. *See also* decorators
  abstract, 211
  asymmetric layering and, 89–91
  asynchronous, 200
  chaining, 123, 124
  Contract.Ensures, 235
  Contract.Invariant, 236, 238
  Contract.Requires, 233, 234
  Dispose, 293, 298–300, 304
  Ensures, 235
  extension, 118–120
  factory, 157–159
  fire-and-forget, 202
  IDispose.Dispose(), 298–304
  Liskov contract rules and, 225–231
  naming, 219
  postconditions and, 222, 223
  preconditions and, 220–222
  private, 236
  Register, 293–295
  RegisterTypes, 318
  Release, 293–295
  request-response, 202
  Requires, 233, 234
  Resolve, 293–295
  signatures, 98–101, 219
  static, 158
  synchronous, 200
  tracking execution time, 197
  virtual, 210, 211, 284
metrics
  correlation vs. causation, 40
  maladaptive code and, 39
  monitoring project progress with, 22–27
  story points, 23
Meyer, Bertrand, 207
Microsoft AppFabric AutoStart, 74
Microsoft Moles, 59

Microsoft .NET Framework
advantages of, 196
Impromptu Interface library, 116
software dependencies and, 43, 48, 49
Microsoft .NET Framework Reflection API, 116
minimum marketable feature (MMF)
defined, 12
vs. epics and themes, 13
minimum viable release (MVR), 12
mixins, 118–122
MMF (minimum marketable feature)
defined, 12
vs. epics and themes, 13
mocking frameworks, 39, 140, 141
mocks
Illegitimate Injection anti-pattern and, 311
setting up, 149
testing with, 140–143
modeling dependencies, 51–55
Model-View-Controller (MVC), dependency injection and, 312–317
Model-View-ViewModel (MVVM) pattern, 282
MongoDB, 276
monitoring progress with charts and metrics, 22–27
Moq, 140, 141
MSTest, 129, 149
multicast network messages, 71
multilayer solutions. *See* layering
multiple inheritance, 96, 118
multiple interface implementation, 95
MVC (Model-View-Controller), dependency injection and, 312–317
MVR (minimum viable release), 12
MVVM (Model-View-ViewModel) pattern, 282

N

.NET Framework. *See* Microsoft .NET Framework
network boundaries, tiers and, 83
NHibernate, 276
niko-niko calendar, 32
nodes, 51
noise-to-signal ratio, contracts and, 237
NuGet
dependencies and, 77–81
testing with Moq and, 140
Null Object pattern, 103–105, 108
NullReferenceException, 103, 108, 144, 145
NullUser class, 105, 107

O

Object Adapter pattern, 110, 111
object graphs
constructing, 287–291
conventions and, 320
resolution root and, 312
object lifetime
IDispose.Dispose() method and, 298–304
release method and, 293
Object property, 141
object-oriented programming (OOP), polymorphism and, 101
Object/Relational Mapper (ORM), 50
observer effect, 22
OCP (open/closed principle). *See* open/closed principle (OCP)
OOP (object-oriented programming), polymorphism and, 101
open for extension rule, 208
open/closed principle (OCP)
bug fixes and, 208, 209
client awareness and, 209
extension points, 209–214
Martin definition, 208
Meyer definition, 207
Order class, 264
ORM (Object/Relational Mapper), 50
out keyword, 240, 241, 245
over-specification, 142, 143

P

package weight, 224
packages. *See also* dependencies
Chocolatey, 80, 81
NuGet, 78–80
pair programming, 353, 370–373
parameters, naming, 219
pass-through delegation, vs. interface segregation, 256
patterns. *See also* anti-patterns; interfaces
Adapter, 109–111, 261

Agile processes and, 1
Arrange, Act, Assert (AAA), 126–130
Class Adapter, 109, 110
Command/Query Responsibility Segregation (CQRS), 90
Composite, 185–189
Decorator, 184, 185, 251
described, 56
Factory, 301, 302, 314, 345–347
Factory isolation, 304, 305
history of, 102
layering, 82–84
Model-View-ViewModel (MVVM), 282
Null Object, 103–105, 108
Object Adapter, 110, 111
overuse vs. underuse, 102
Poor Man's Dependency Injection, 287
Register, Resolve, Release, 293–295
Repository, 241
Responsible Owner, 302–304
Stairway, 65, 66, 284
Strategy, 111–113, 204–206
Template Method, 211
peak of inflated expectations, 327
peer reviews, 340–357, 380
performing unit tests. *See* unit tests
persistent storage, CRUD operations and, 251, 252
pigs and chickens, 9
Plain Old CLR Object (POCO), 285
planning poker, story point estimation and, 331–335
plateau of productivity, 327
PO (product owner) role, 7, 325, 326
POCO (Plain Old CLR Object), 285
poker, planning, story point estimation and, 331–335
polymorphism
covariance and, 239, 240
described, 101, 102
replacing conditional expressions with, 154–157
Poor Man's Dependency Injection (Poor Man's DI)
composition root and, 311
described, 287–290
post mortems. *See* sprint retrospectives
postconditions
code contracts and, 235
described, 222, 223
Liskov Substitution Principle and, 227–229
PostSharp, 88
preconditions
arranging for unit tests, 126, 127
code contracts and, 232–234
of constructors, 143
contracts and, 224
described, 220–222
enforcing with exceptions, 221
enforcing with preconditions, 220
Liskov Substitution Principle and, 225–227
predicate decorators, 189–193
predicted variation, 213
prioritizing features, 29
Prism (Windows Presentation Foundation/Model-View-ViewModel library), 122
private methods, 236
product backlogs
described, 27
responsibility for setting priorities, 7, 27
Scrum process and, 4
product owner (PO) role, 7, 325, 326
production code
defined, 125
test-driven development and, 130
profiling decorators, 196–200
proof of concept, refactoring toward abstraction and, 177
properties
decorating, 203
signatures, 99
property injection, 291
property setters, 230, 231
protected variation, 213, 214
prototypes, refactoring toward abstraction and, 177
proxiable classes, 284
proxies
classes, 70
discovery, 71
services, 70, 71
prudent technical debt, 20
publishing events, 261–263
pull requests, 380

Q

Quality Assurance (QA), swimlanes for, 18
queries, separating from commands, 89–91

R

Rapid Application Development (RAD), 196
readability, refactoring code for, 153, 154
reading data, segregating from writing data, 269–274
reckless technical debt, 20
recursion, 55
red, green, refactor, 130–135
red icon, as indicator of failed unit test, 130
refactoring code. *See also* dependency injection (DI); interface segregation
  for abstraction, 177–183
  for clarity, 172–176
  defined, 125
  process described, 151
  for readability, 153, 154
  replacing conditional expressions with polymorphism, 154–157
  replacing constructors with factory methods, 157–159
  replacing inheritance with delegation, 163–166
  replacing magic numbers with constants, 153, 154
  samples, 347, 348, 350–353
  technical debt and, 57
  test-driven development and, 130–135
references
  converting third-party to first-party, 181
  defaults by project type, 49
  NuGet and, 77–81
  resolution process, 67
  services, 70, 71
  third-party, 50
Reflection Emit, 116
refused bequests, 162
Register method, 293–295
Register, Resolve, Release pattern, 293–295
registering interfaces to their implementations, 293–295
RegisterTypes method, 318
registration
  conventions and, 317–321
  imperative vs. declarative, 295–297
release
  defined, 11
  minimum viable, 12
  planning, 29
  scrum process and, 4
Release method, 293–295
Re-motion Re-mix library, 118, 120–122
repaying technical debt, 20
repositories, 379–382
repository interfaces, 136
Repository pattern, 241
request-response methods, 202
Requires method, 233, 234
resolution root, 312
Resolve method
  Inversion of Control and, 292
  Register, Resolve, Release pattern and, 293–295
resolving dependencies, 67–76, 291
responsibilities. *See* single responsibility principle (SRP)
Responsible Owner pattern, 302–304
RESTful services, 74–77
retrospectives, for sprints, 34, 35, 358–363, 376–378
return types, variance rules for, 218
return values, postconditions and, 222, 223
reuse, blackbox vs. whitebox, 110
rigidity, 37, 38
roles and responsibilities, Scrum process and, 7–9, 325–328
rules, Liskov substitution principle, 218, 225–231

S

scaling applications by using tiers, 83
Scrum
  Agile method and, 4
  calendar, 36
  defined, 3
  documentation and, 5, 9
  monitoring project progress, 22–27
  problems with, 37–41
  process overview, 4
  roles and responsibilities, 7–9, 325–328
  software dependencies and, 43
  user stories and, 13, 328–335
  variants of, 6
  vs. waterfall, 4–6
Scrum boards
  avatars, 17
  cards, 10–17

color schemes, 17
example of, 10
importance of, 10
location, 9
swimlanes, 18
Scrum master (SM) role, 8, 326
Seemann, Mark, 320, 321
segregation. *See* interface segregation
Service Discovery, 71–74
Service Locator anti-pattern, 306–310
service proxies, 70, 71
service references, 70, 71
ServiceDiscoveryBehavior, 73
ServiceException, 146, 147
services
dependencies and, 70–77
sample application showing dependencies, 44–51
software dependencies and, 44
unit testing examples for, 135–144
setup method, 149, 150
signature clash, explicit implementation and, 98
signatures, methods for, 98–101, 219
single responsibility principle (SRP)
abstraction, refactoring for, 177–183
clarity, refactoring for, 172–176
Decorator pattern and, 184
delegating tasks to other methods, 173
described, 169
problem example, 169–172
skyhooks vs. cranes, 38, 39
slicing user stories vertically, 15
SLOC (Source Lines of Code), 39
slope of enlightenment, 327
SM (Scrum master) role, 8, 326
software dependencies. *See* dependencies
software development, user stories and life cycle, 14
software testers, Scrum process and, 8. *See also* testing; unit tests
source control, 379–383
Source Lines of Code (SLOC), 39
SourceTree, 381
splitting interfaces. *See also* interface segregation
architectural need, 275–279
client need, 268–275
sprint demos, 33, 357, 358, 375, 376
Sprint Handover Day, 36
sprint retrospectives, 34, 35, 358–363, 376–378
sprint zero adaptive sample, 325–335
sprints
burndown charts, 24, 25
described, 28
goals, 337
meetings, 29–31
planning, 30, 31, 337–339, 365, 366
product owner responsibilities during, 7
Scrum master responsibilities during, 8
Scrum process and, 4
software dependencies, effect on, 43
velocity and, 23, 24
SSADM (Structured Systems Analysis and Design Methodology), 9
stable interfaces, 214
Stairway pattern, 65, 66, 85, 178, 284
static classes, 308
static contract verification, 232
static methods, 158
static typing, 115
Stopwatch class, 197–200
storage, persistent, 251, 252
stories. *See* user stories
story points
as progress metric, 23
claiming, 18
estimating, 330–335
triangulating, 35
user stories and, 14, 15
velocity and, 23
Strategy pattern, 111–113, 204–206
strict mocks, 141
Structured Systems Analysis and Design Methodology (SSADM), 9
subclasses
abstract methods and, 211
Liskov contract rules and, 225–231
subscribing to events, 261–263
substitution. *See* Liskov substitution principle (LSP)
subtypes
Liskov substitution principle and, 218
polymorphism and, 239
supertypes
contracts, 218
polymorphism and, 239

SUT (system under test), 126–130. *See also* testing; unit tests
swarming fast track items, 18
swimlanes, on Scrum board, 18
switch statements, 204–206
synchronous methods, 201
system requirements for code samples in this book, xxii
System.Diagnostics.Contracts namespace, 232
System.Diagnostics.Stopwatch class, 197
System.Diagnostics.Contracts namespace, 233
system under test (SUT), 126–130. *See also* testing; unit tests

T

task management application example, 281–287
tasks
  assigning, 14
  user stories and, 14
TCP vs. UDP, 74
TDD (test-driven development), 130–135
technical debt
  decision to take on, 370, 373
  defined, 16
  good vs. bad, 19
  quadrant, 19–20
  refactoring code and, 57
Template Method pattern, 211
TEntity types, 252, 260
test analyst role, Scrum process and, 328
testability
  dependencies and, 59
  mixins and, 119
  unit tests and, 126
test runners, 128, 129
test-driven development (TDD), 130–135
testing. *See also* unit tests
  coverage, 40
  with fakes, 137–140
  maladaptive code and, 38, 39
  with mocks, 140–143
  Scrum process and, 8
TestInitialize attribute, 149
themes vs. features, 13
third-party libraries
  Log4Net, 181, 182
  Re-motion Re-mix, 118, 120–122
third-party software dependencies, 43, 50, 51, 66
three-layer architecture, 86, 87, 90
throwing exceptions, enforcing precondition contracts by, 220, 221
tiers vs. layers, 83
tightly coupled classes, 209
tools for adaptive programming, 379–385
transactional consistency, 91
Trey Research sample application, 325–328
trifle, compared to well-designed software, 15
trough of disillusionment, 327
try/catch block, 147, 148
try/finally block, 303
two-layer solutions, 83–86, 90
Typemock, 59
types
  Composite pattern and, 187, 188
  extending, 118–120
  generating new at run time by using Impromptu Interface, 116
  generating new on the fly with Re-Motion Re-mix, 120–122
  mapping to interfaces, 292
  variance, 218
type-sniffing, mixins and, 122
typing
  duck, 113–118
  dynamic vs. static, 115
  weak, 321

U

UDP vs. TCP, 74
unauthorized interfaces, 274
undirected graphs, 51
unit tests
  Arrange, Act, Assert (AAA) pattern, 126
  arranging the preconditions, 126, 127
  asserting expected behavior in, 128
  continuous integration and, 384
  coverage, 40
  defined, 125
  failing, 139
  with fakes, 137–140

Illegitimate Injection and, 311
maladaptive code and, 38
with mocks, 140–143
parts, 126
performing the testable acts, 127
runners, 128, 129
running, 128–130
samples, 341–344
Unity containers, 292, 293, 296, 297, 308
untestability, maladaptive code and, 38
user interface
defects in, 16
layer, 83, 84
user stories
defined, 3
initial backlog of, creating, 328–335
sample, creating rooms for categorizing conversations, 340–348
sample, list of rooms, 349–353
sample, sending messages, 356, 357
sample, viewing messages, 353–355
Scrum board cards and, 13
slicing vertically, 15
tasks and, 14
using block, 299, 302, 303

V

variables
naming, 154
replacing magic numbers with, 153
variance
contravariance, 242–244
covariance, 239–242
defined, 239
invariance, 244, 245
rules, 218
velocity, measuring, 23, 24
vertical scaling
tiers and, 83
vs. horizontal scaling, 373
vertical slices, 15, 370
viewing assemblies that have been loaded into memory, 47
view models. *See* Model-View-ViewModel (MVVM) pattern
virtual methods, 210, 211, 284

W

waterfall method, xv, 4–6
Web Services Definition Language (WSDL) file, 70
weight, encapsulation and, 224
whitebox reuse, 110
wikis, Scrum and, 6
Windows Forms, dependency injection and, 315, 316
wrapping exceptions, 146, 149
writing data, segregating from reading data, 269–274
WSDL (Web Services Definition Language) file, 70

X

XML, declarative registration and, 296, 297