



P r o f e s s i o n a l E x p e r t i s e D i s t i l l e d

Refactoring with Microsoft Visual Studio 2010

Evolve your software system to support new and ever-changing requirements by updating your C# code base with patterns and principles

Peter Ritchie

[PACKT] enterprise 
PUBLISHING professional expertise distilled

www.allitebooks.com

Refactoring with Microsoft Visual Studio 2010

Evolve your software system to support new and ever-changing requirements by updating your C# code base with patterns and principles

Peter Ritchie



BIRMINGHAM - MUMBAI

Refactoring with Microsoft Visual Studio 2010

Copyright © 2010 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: July 2010

Production Reference: 1190710

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849680-10-3

www.packtpub.com

Cover Image by Sandeep Babu (sandyjb@gmail.com)

Credits

Author

Peter Ritchie

Reviewers

Atul Gupta

Anand Narayanaswamy

Vivek Thangaswamy

Hima Bindu Vejella

Acquisition Editor

Rashmi Phadnis

Development Editors

Neha Patwari

Ved Prakash Jha

Technical Editor

Neha Damle

Indexer

Monica Ajmera Mehta

Editorial Team Leader

Gagandeep Singh

Project Team Leader

Lata Basantani

Project Coordinator

Shubhanjan Chatterjee

Proofreaders

Lesley Harrison

Claire Cresswell-Lane

Aaron Nash

Graphics

Nilesh Mohite

Production Coordinator

Adline Swetha Jesuthas

Cover Work

Adline Swetha Jesuthas

About the Author

Peter Ritchie is a software development consultant. Peter is president of Peter Ritchie Inc. Software Consulting Co., a software consulting company in Canada's National Capital Region specializing in Windows-based software development management, process, and implementation consulting. Peter has worked with such clients as Mitel, Nortel, Passport Canada, and Innovapost, from mentoring to architecture to implementation. Peter's range of experience ranges from designing and implementing simple stand-alone applications to architecting n-tier applications spanning dozens of computers; from C++ to C#.

Acknowledgement

Any sort of body of work is like a child: it takes a village raise a body of work. This body of work is no different; it could not have existed without a huge village of software development constituents. Many people from this village have influenced me directly and many more have influenced me indirectly. There are too many to faithfully acknowledge in a single place.

Any sort of book on Refactoring is based on the work of Martin Fowler and William Opedyke. This book could not have existed in the state it has without their work. Refactoring itself is based on the techniques and methodologies developed or promoted by Ward Cunningham and Kent Beck.

Some of the refactorings use design techniques right out of Domain Driven Design. Eric Evens organized and systematized patterns and practices on good object-oriented design.

I have to acknowledge the early years of the ALT.NET movement and the people involved in it. ALT.NET promoted a more scientific view of software development, promoting generally accepted principles, methodologies, and community over not-invented-here, cowboy development, and working in a vacuum. I can't possibly list all the people who have been involved with ALT.NET, but some of those people that I've had the pleasure of being involved with or influenced by include (in no particular order): David Larabee, Scott Bellware, Jeremy Miller, Greg Young, Donald Belcham, James Kovacs, Jean-Paul Boodhoo, Kyle Baley, Karl Seguin, Oren Eini, Steven List, Adam Dymitruk, Udi Dahan, Glenn Block, Derek Whittaker, Justice Gray, Roy Osherove, Scott Allen, Scott Koon, Brad Wilson, and many, many others.

Much thanks to the people at Packt and the technical reviewers that provided many other points of view to what I had written. Thanks to Bill Wagner for his feedback and advice.

Also, many thanks to Charlie Calvert, Mark Michaelis, and Bill Wagner for our collaborations on community. It promoted and facilitated my views on being involved with and giving back to the software development community.

Finally, I have to acknowledge my wife Sherry, who's had the patience and support that allows me to follow my software development interests that take up so much of my spare time, like writing this book.

About the Reviewers

Atul Gupta is the Principal Technology Architect at the Microsoft Technology Center, Infosys Technologies. With close to 15 years of experience working on Microsoft technologies, Atul is currently a Principal Technology Architect at Infosys' Microsoft Technology Center. His expertise spans User Interface technologies, and he currently focuses on Windows Presentation Foundation (WPF) and Silverlight technologies. Other technologies of interest to him are Touch (Windows 7), Deepzoom, Pivot, Surface, and Windows Phone 7.

His prior interest areas were COM, DCOM, C, VC++, ADO.NET, ASP.NET, and AJAX. He has authored papers for industry publications and websites, some of which are available on Infosys' Technology Showcase. Along with colleagues from Infosys, Atul is also an active blogger. Being actively involved in professional Microsoft online communities and developer forums, Atul has received Microsoft's Most Valuable Professional award for multiple years in a row.

Anand Narayanaswamy, Microsoft MVP, is the author of *Community Server Quickly* (www.packtpub.com/community-server/book) published by Packt Publishing. He works as a freelance writer based in Trivandrum, India besides devoting time for blogging and tweeting. He also works as a technical editor for ASPAlliance.com. He had worked as a technical editor/reviewer for various publishers such as Sams, Addison-Wesley, Mc Graw Hill, and Packt. He runs www.learnxpress.com and www.dotnetalbum.com.

First, I would like to thank the almighty for giving me the strength and energy to work every day. I specially thank my father, mother, and brother for providing valuable help, support and encouragement. I also thank Shubhanjan Chatterjee, Neha Patwari, and Peter Ritchie for their assistance, co-operation, and understanding throughout the review process of this book.

Vivek Thangaswamy has been working as a solution developer in Software Technologies for more than six years now. He has worked for many top notch clients across the globe. Vivek started programming in a DOS world, moved to C, C++, VC++, J2EE, SAP B1, LegaSuite GUI, WinJa, JSP, ColdFusion, VB 6, eventually to .NET in both VB.NET and C# worlds and also in ASP.NET / MS SQL Server and more into Windows Mobile platforms. He also worked in Microsoft's latest trendsetter in Enterprise Collaboration Microsoft Office SharePoint Server accompanied with VSTO and .NET 3.0 frameworks. He started working in SharePoint from the version 2003 to up to date version. Now he is more into Mobile platform Research and Development. Different domains and industries knowledge and experience eCommerce, ERP, CRM, Transportation, Enterprise Content Management, Web 2.0 and Portal. Expert in SAP B1, and SugarCRM consulting. Focusing on Java ME, Windows Mobile, JavaFX Mobile and Android, basically, what Vivek does is answer more out in the newsgroups over and over, plus adds to its blogging about Microsoft Technologies, wraps it in a very readable and interesting format and more in technical writing. For his good technical knowledge, passion about the Microsoft Technologies, community involvement and contribution, he has also been awarded the Microsoft Most Valuable Professional award for ASP.NET (once) and SharePoint (twice). He is the lead technology consulting advisor for Arimaan Global Consulting (www.arimaan.com).

Vivek completed his Bachelor Degree in Information Technology (B.Tech), from one of the oldest and finest universities in the world, University of Madras and has an MBA (Master of Business Administration) in Finance from one of the largest Open universities in the world, IGNOU.

Writing is a passion for Vivek, and he has written many technical articles and whitepaper based on different technologies and domains. He also authored a technical book on Microsoft technology VSTO 3.0 for Office 2007 Programming by *Packt Publishing* and has been a reviewer for Microsoft Office Live Small Business: Beginner's Guide by *Packt Publishing*.

I dedicate this book to my Arimaan Global Consulting technical team members, for their excellence and support.

Hima Bindu Vejella, a B.Tech graduate, Microsoft MVP since 2006, .NET Rock Star, working as Team Manager at Prokarma Softech Hyderabad, has eight years of experience in software development using Microsoft Technologies. Hima Bindu Vejella, Active community leader, all time winner in Community-Credit since 2006, author at aspalliance, dotnetslackers, and simpletalk. She is also speaker and book-reviewer at DotnetUserGroupHyderabad India Lead and Moderator at syntaxhelp, Technical Member at dotnetspider. She has spoken at more than 200 sessions at various colleges and events online and offline. She has taken sessions on MVP awareness, VS 2010, VS2010 at MNCs, UG Meets, Events and at corporate companies. Visit her blog at <http://himabinduvejella.blogspot.com>.

She is regular columnist as international author for Mundo.NET, Portugal magazine, author at ASP Alliance and dotnetslackers. Her recent series of articles for VS2010 are like a white paper on VS2010 features are published in MSDN blog. She is founder and moderator of **MUGH (Microsoft User Group Hyderabad)**, and most active UG community leader in India. She is contributed to syntaxhelp, submitted more than 500 code snippets on various technologies like ASP.NET, C#, VB, SharePoint, WP, LINQ, and so on. She is not only active in Indian MVPs but also internationally doing a lot for the community. She is associated with many online technical communities and has helped lot of people in finding solutions to their problems. She can be reached at himabvejella@gmail.com. She believes in "Aim to go where you have never been before and strive to achieve it".

I would like to thank my beloved husband Mr. Vamshi, parents, in-laws, and Sai, a wonder kid, for being supportive all the time while I was spending my personal time working on the laptop on weekends.

Table of Contents

Preface	1
Chapter 1: Introduction to Refactoring	7
What is refactoring?	8
Why the term refactoring?	10
Unit testing—the second half of the equation	11
Simple refactoring	12
Technical debt	15
In the software development trenches	15
The option of rewriting	16
Working refactoring into the process	20
What to refactor	21
Refactoring to patterns	22
Refactoring to principles	22
Code smells	23
Complexity	23
Performance	24
Kernel	24
Design methodologies	25
Unused and highly-used code	25
Refactoring in Visual Studio® 2010	26
Static code analysis	26
Code metrics	27
Summary	28
Chapter 2: Improving Code Readability	31
Built-in Visual Studio® refactorings	32
Rename identifier refactoring	33
Rename field	33
Rename property	35

Rename method	36
Rename local variable	36
Rename class	37
Extract Method refactoring	37
Encapsulate Field refactoring	39
The smell of code	41
Duplicate code smell	41
Duplicate code in a class	42
Duplicate code in multiple classes	47
Duplicate code in construction	53
Advanced duplicate code refactoring	54
Long method smell	56
Code comments smell	56
Dead code	60
Intention-revealing design	60
You ain't gonna need it	61
KISS principle	62
Keeping track of code changes	64
Check-in often	65
Removing unused references	65
Summary	66
Chapter 3: Improving Code Maintainability	67
Code maintainability	67
Automated unit testing	69
Feature Envy code smell	81
Design for the sake of reuse	83
Don't repeat yourself	84
Inappropriate Intimacy code smell	84
Lazy Class code smell	87
Improved object-model usability	88
Contrived Complexity	90
Detecting maintainability issues	95
Summary	98
Chapter 4: Improving Code Navigation	99
Navigating object-oriented code	100
Convention over Configuration	101
Consistency	101
Conventions	102
Naming	102
Scoping types with namespaces	104
IDE navigation	106
Search	106

Class View	108
Solution Explorer	111
Class Diagram	114
Code Editor	116
Navigation with design patterns and principles	118
Summary	120
Chapter 5: Improving Design Correctness	121
Liskov substitution principle	122
Convert to Sibling refactoring	126
Refactoring to single class	128
Composition over inheritance	134
Refactoring virtual methods to events	138
Exceptions to preferring composition	140
Replace inheritance with delegation refactoring	140
Object-oriented design and object behavior	142
Refactoring to improved object-orientation	144
Move initialization to declaration	148
Value types	150
Refactoring to value type	150
Modeling business rules appropriately	152
Summary	153
Chapter 6: Improving Class Quality	155
Cohesion	155
Class cohesion	156
The Single Responsibility Principle	160
Refactoring classes with low-cohesion	160
Detecting classes with low-cohesion	163
Method cohesion	164
Refactoring methods with low-cohesion	165
Namespace cohesion	171
Refactoring namespaces with low-cohesion	171
Assembly cohesion	173
Refactoring assemblies	174
Coupling	174
Refactoring subclass coupling	175
Refactoring content coupling	176
Interface-based design	176
Delegates	179
Events	180
Refactoring external coupling	183

Dependency cycles	186
Proper dependency design	187
Summary	188
Chapter 7: Refactoring to Loosely Coupled	189
What is loosely coupled?	190
What are coupling and dependencies?	190
Tightly-coupled	191
Dependency Inversion principle	191
Inversion of Control	192
Dependency Injection	193
Working with IoC containers	196
Tightly coupled to creation	199
Factory Pattern	200
Abstract Factory	201
Decorator pattern	203
Detecting highly-coupled	206
Types of coupling	207
Afferent coupling	207
Efferent coupling	208
Interface segregation principle	208
Drawbacks of loosely-coupled	214
Other methods of loose-coupling	214
Web services	215
Communication hosts	215
Summary	216
Chapter 8: Refactoring to Layers	217
Layers	217
Business logic and domain layers	219
Data Access Layers	221
Refactoring UI data access to Data Access Layer	221
Refactoring domain data access to Data Access Layer	233
Plain Old CLR Objects	236
User interface layers	237
Model View Presenter (MVP)	238
Additional layers	248
Summary	249
Chapter 9: Improving Architectural Behavior	251
Behavioral patterns	252
Don't Repeat Yourself (DRY)	253
Strategy pattern	253

Detecting need for strategy pattern	254
Refactoring to strategy pattern	255
Specification pattern	259
Detecting need for specification pattern	261
Refactoring to specification pattern	261
Publish/Subscribe paradigm	273
Observer pattern	274
Detecting the need for the observer pattern	275
Refactoring to the observer pattern	275
Summary	279
Chapter 10: Improving Architectural Structure	281
Structural patterns	281
Legacy code	282
Adapter pattern	283
Detecting need for the adapter pattern	283
Refactoring to the adapter pattern	284
Façade pattern	291
Detecting the need for façade	291
Refactoring to the façade pattern	292
Proxy pattern	299
Detecting need for proxy	300
Refactoring to proxy	300
Object/Relational Mapping	304
Problems with code generation	305
Detecting need to refactor to ORM	306
Refactoring to ORM	307
ORM sessions	312
Summary	312
Chapter 11: Ensuring Quality with Unit Testing	313
Change is not always good	314
Automated testing	314
Unit tests	314
Other testing	315
Mocking	316
Priorities	320
Code coverage	320
Mocking frameworks	321
Rhino Mocks	321
Moq	324
Unit testing existing legacy code	325
TypeMock isolator	327

Table of Contents

Unit testing in Visual Studio®	330
Test driven development	332
Unit testing frameworks	337
NUnit	337
XUnit	338
Automating unit-testing	339
Continuous Integration	340
Third party refactoring tools	341
Resharper	341
Refactor! Pro	341
Summary	342
Index	343

Preface

This book introduces the reader to improving a software system's design through refactoring.

It begins with simple refactorings and works its way through complex refactorings by building on the simple refactorings. You will learn how to focus changing the design of their software system and how to prioritize refactorings—including how to use various Visual Studio features to focus and prioritize design changes. The book also covers how to ensure quality in light of seemingly drastic changes to a software system. You will also be able to apply standard established principles and patterns as part of the refactoring effort with the help of this book.

What this book covers

Chapter 1, Introducing Refactoring, describes what refactoring is, its importance, and its priority in the software development effort. Comparison to re-writing and what "Technical Debt" is and how refactoring can be used to pay down technical debt is covered in this chapter.

Chapter 2, Improving Code Readability, begins detailing the refactorings built in to Visual Studio and how they can make code more readable. Code smells are introduced, and which code smells apply to readability, and how to detect and refactor them are detailed in this chapter.

Chapter 3, Improving Code Maintainability, continues to detail the refactorings built in to Visual Studio and how they can make code more maintainable. Code smells that apply to maintainability, how to detect and refactor them are detailed in this chapter. The importance of unit testing is covered in this chapter.

Chapter 4, Improving code navigation, continues with simple refactorings and how code can be refactored to improve its navigability in general and takes into account Visual Studio code navigation abilities.

Chapter 5, Improving design correctness, begins detailing complex refactorings. Design principles such as Liskov Substitution and Composition over Inheritance are introduced and how to perform refactorings related to these principles is covered in this chapter.

Chapter 6, Improving class quality, introduces code quality metrics like cohesion and coupling. Principles related to cohesion and coupling are introduced and refactorings that increase cohesion and decrease coupling are covered in this chapter.

Chapter 7, Refactoring to loosely-coupled, expands on coupling from the previous chapter and drills-down on loosely-coupled design. Principles related to loosely-coupled are introduced and complex refactorings related to loosening coupling are covered in this chapter.

Chapter 8, Refactoring to layers, continues with more complex refactorings that involve layered architectures. Typical layers, Model View Presenter, and Repository patterns and how and when to refactor to them are also detailed in this chapter.

Chapter 9, Improving architectural behavior, details complex refactorings to improve architectural behavior. Design behavior patterns, when and how to refactor to them are detailed in this chapter.

Chapter 10, Improving architectural structure, continues with architectural-related complex refactorings. Object-Relational Mapping (ORM) and refactoring Repository implementations are included in this chapter.

Chapter 11, Ensuring Quality with Unit Testing, details the importance of unit testing. How unit testing applies to refactoring, examples of unit testing to support the refactoring effort, and legacy code are also detailed in this chapter.

What you need for this book

We will use examples in C# in Visual Studio 2010, but the concepts can be applied to any version of Visual Studio since version 2005.

Who this book is for

This book is primarily for developers who want to refactor their code in Visual Studio. However, the book can be used by anyone using Visual Studio. Developers, designers, and architects who are eager to improve the performance of their craft will find this book useful because it details refactoring existing code to use recognized and established patterns and principles to improve code structure and architectural behavior. The book assumes that the reader knows both Visual Studio and C#. No previous knowledge of refactoring is required.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."


A block of code is set as follows:


```
foreach (InvoiceLineItem invoiceLineItem in
    InvoiceLineItems)
{
    invoiceSubTotal +=
        (float)((decimal)(invoiceLineItem.Price
            - invoiceLineItem.Discount)
            * (decimal)invoiceLineItem.Quantity);
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
foreach (InvoiceLineItem invoiceLineItem in
    InvoiceLineItems)
{
    invoiceSubTotal +=
        (float)((decimal)(invoiceLineItem.Price
            - invoiceLineItem.Discount)
            * (decimal)invoiceLineItem.Quantity);
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

 [Warnings or important notes appear in a box like this.]

 [Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.


To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

[	<p>Downloading the example code for the book</p> <p>You can download the example code files for all Packt books you have purchased from your account at http://www.PacktPub.com. If you purchased this book elsewhere, you can visit http://www.PacktPub.com/support and register to have the files e-mailed directly to you.</p>]
---	---	--	---

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Introduction to Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

- Martin Fowler

This chapter begins our journey together into refactoring. In this chapter, we'll provide some introductory information on what refactoring is, its importance, and why we'd want to do it. We'll also describe some refactorings and some of the side-effects of not performing regular refactoring.

The following are a list of topics for this chapter:

- What is refactoring?
- Why the term refactoring?
- Simple refactoring
- Complex refactoring
- Technical debt
- The option of rewriting

What is refactoring?

Although the task of refactoring has been around for some time, and the term **refactoring** and the systematic approach associated with it have also been around for a long time; Martin Fowler and all were the first to popularize refactoring and begin organizing it more systematically.

Refactoring is a very broad area of software development. It could be as simple as renaming a variable (and updating all the uses of that variable), or it could refer to breaking a class that has taken on too much responsibility into several classes, like implementing a pattern. Refactoring applies to all complex software projects that require change over time.

A pattern is a [description of] a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

– Christopher Alexander

Refactoring is changing the design of a software system without changing its external behavior. Changes to internal structure and interfaces that don't affect the system's external behavior are considered refactoring.

The term refactoring was coined by William Opdyke in what seems to be an evolution of factoring. Factoring was a common term used in the Forth programming language (and mathematics) to mean decomposition into constituent parts. The term refactoring has been around since at least the early 1990's. Few developers would argue that refactoring isn't something they do on a day-to-day basis. Although refactoring is a fairly simple concept, many programmers don't associate changing code without changing external behavior as being refactoring.

Refactoring has been an integral part of software development in the **Extreme Programming (XP)** methodology since Kent Beck introduced it in 1999.



Kent Beck introduced XP in the book *Extreme Programming Explained* circa 1999.



XP mandates a **Test-Driven-Development (TDD)** process, where only enough code is written to implement a single feature and to pass at least a single test. The code is then refactored to support implementing another feature and pass all the existing tests. The new feature is then implemented and passes at least a single new test and all current unit tests. Extreme Programming also mandates *continuous refactoring*. We will discuss the importance of tests when refactoring in Chapter 3.

Some of the first tools to support automated refactoring came out of the Smalltalk community. The "Refactoring Browser" was one of the first user interfaces that provided abilities to refactor Smalltalk code. Now refactoring tools are commonplace in almost all programming language communities. Some of these tools are stand-alone; while some are add-ins to existing tools; while some refactoring abilities are built into other tools and applications where refactoring isn't their main purpose. Visual Studio® 2010 fits in the second two categories. Visual Studio® 2010 is an integrated development environment (IDE) designed specifically to allow developers to perform almost all the tasks required to develop and deliver software. Visual Studio®, since version 2005, has included various refactoring abilities. Visual Studio® has also included extensibility points to allow third-party authors to write add-ins for it. Some of these add-ins have the purpose of giving users of Visual Studio® 2010 specific refactoring abilities. Add-ins such as Resharper, Refactor! Pro, and Visual AssistX add more refactoring abilities for Visual Studio® 2010 users.

Refactoring has become more mainstream with Visual Studio® users in the past few years because of the built-in automated refactoring abilities of many IDEs, including Visual Studio® 2010. It's now almost trivial, for example, to rename a class in a million line software project and update dozens of references to that class with a few mouse clicks or keystrokes. Before the support for this simple refactoring came about, this problem would have involved searching for text and manually changing text or using a brute force search-and-replace to replace all instances of the class name and then rely on the help of the compiler to tell you where replacements weren't actually a use of the class name. While it's almost unheard of for IDEs to not have some sort of (at least rudimentary) simple automated refactoring abilities, developers will always have to manually perform many complex refactorings (although they may consist of some simple refactorings that *can be automated*).

Without automatic refactoring abilities and tools, developers feel friction when performing simple refactoring. Changing the order of parameters in a complex million-line software project, for example, is tedious and error prone. Encountering a spelling mistake in the name of a method that is referenced by dozens of other methods in dozens of other files is time-consuming. Without automated tools, our maintenance problems cause even more friction; simple tasks like changing the order of parameters or fixing spelling mistakes are simply avoided. This prevents a code base from improving in its maintainability and it becomes even more fragile and even more likely to be neglected (have you ever tried to find a specific method with text search only to find out someone misspelled it?)

It's the ease with which simple refactorings can be performed that has elevated "refactoring" in to the lexicon of almost every programmer. Yet, there is so much more to the act of refactoring than just simple refactorings that can be accomplished automatically by software.

The common thread in all refactoring is the goal of the refactoring. The goal can be as simple as making the code easier to read and maintain, or to make the code more robust; or the goal may be as complex as improving componentizing code modularity to make it more decoupled and to make it easier to add new behavior. But, systematic refactoring is the acceptance that the act of writing software is not atomic; it cannot be done in a single step and will evolve over time as our understanding of the problem domain improves and/or our customer's understanding of their needs is discovered and communicated.

Why the term refactoring?

So, why bother with a specific term for this type of modifying code? Isn't all modifying code simply *modifying code*? A systematic approach to the different types of editing and writing code allows us to focus on the side-effects we're expecting as a result of changing the code. Making a change that includes fixing a bug, refactoring, and adding a feature means that if something doesn't work then we're unsure which of our edits caused the problem. The problem could be that we didn't fix the bug correctly, that there was a problem with our refactoring, or that we didn't add the feature properly. Then there's the possibility that one of the edits interacted with the other. If we do encounter an adverse side-effect (also known as, "a bug") in this process, we have an overly large combination of possible causes to evaluate.

It's much easier to focus on one type of task at a time. Make a bug fix, validate the bug fix didn't cause any expected side-effects. When we're sure that the bug fix works, we move on to adding new functionality. If there are unexpected side-effects while we're adding new functionality, we know that the side-effect was caused either by the code that we have just added to implement the new feature, or the way that code interacts with the rest of the system. Once we know the new functionality is working correctly, we can reorganize the code through refactoring. If we encounter any unexpected side-effects through the refactoring, then we know that the problem comes from either the code that was refactored or some way that the refactored code interacts with the rest of the system—further minimizing the domain in which the unexpected side-effect could exist. This systematic approach reduces the time and work involved in writing software.

It's rare as software designers and programmers that we know at the start of a project exactly what the end-user requires. Requirements can be unclear, wrong, incomplete, or written by someone who isn't a subject matter expert. Sometimes this leads us to make an educated guess at what an end-user requires. We may create a software system in the hope of someone finding it useful and wanting to purchase it from us. Sometimes we may not have direct access to end-users and base all our decisions on second-hand (and sometimes third-hand) information. In situations such as these, we're essentially betting that what we're designing will fulfill the end-user's requirements. Even in a perfect environment, concrete requirements change. When we find out that the behavior does not fulfill the end-user's real requirements, we must change the system. It's when we have to change the behavior of the system that we generally realize that the design is not optimal and should be changed.

Writing software involves creating components that have never been written before. Those components may involve interaction with third-party components. The act of designing and writing the software almost always provides the programmer with essential knowledge of the system under development. Try as we might, we can almost never devise a complete and accurate design before we begin developing a software system. The term **Big Design Up Front (BDUF)** describes the design technique of devising and documenting a complete design before implementation begins. Unless the design repeats many aspects of an existing design, the act of implementing the design will make knowledge about the system illicit, which will clarify or correct the design. Certain design techniques are based on this truth. Test-Driven Development, for example, is based on realizing the design as the code is written – where tests validate the code in implementing requirements of the system.

Regardless of the design technique; when a design needs to change to suit new or changing requirements, aspects of the existing design may need to change to better accommodate future changes.

Unit testing—the second half of the equation

In order to support the refactoring effort – and to be consistent with changing design and not external behavior – it's important that all changes be validated as soon as possible. Unit tests validate that code continues to operate within the constraints set upon it.

Unit tests validate that code does what it is supposed to do; but unit tests also validate that any particular refactoring hasn't changed the expected behavior and side-effects of that code. They also validate that no new and unexpected side-effects have been introduced.

It's important that refactoring be done in conjunction with unit tests. The two go hand-in-hand. The refactorings described in this book will be focused on the details of the refactoring and not about how to unit test the code that is being refactored. Chapter 11 details some strategies for approaching unit-testing in general and in circumstances particular to refactoring.

Simple refactoring

Simple refactorings are often supported by tools—IDEs, IDE add-ons, or stand-alone tools. Many simple refactorings have been done by programmers since they started programming.

A simple refactoring is generally a refactoring that can occur conceptually in one step—usually a change to a single artefact—and doesn't involve a change in program flow. The following are examples of simple refactorings that we'll look at in more detail:

- Renaming a variable
- Extracting a method
- Renaming a method
- Encapsulating a field
- Extracting an interface
- Reordering parameters

Renaming a variable, is a simple refactoring that has a very narrow scope—generally limited to a very small piece of code. Renaming a variable is often done manually because of its simplicity. Prior to having automated refactorings, the rename variable refactoring could often be done reliably with search and replace.

The **Extract method** refactoring is a type of composing method refactoring. Performing an extract method refactoring involves creating a new method, copying code from an existing method and replacing it with a call to the new method. Performing this refactoring can involve use of local variables outside the scope of the original code that would then become new parameters added to the new method. This refactoring is useful for abstracting blocks of code to reduce repetition or to make the code more explicit.

Another simple refactoring is **Rename method**. Renaming method is a simplification refactoring. It has a scope whose change can impact much of a code base. A public method on a class, when renamed, could impact code throughout much of the system if the method is highly coupled. Performing the refactoring rename method involves renaming the method, then renaming all the references to that method through all the code.

Encapsulating a field is an abstraction refactoring. Encapsulating a field involves removing a field from the public interface of a class and replacing it with accessors. Performing an encapsulate field refactoring may involve simply making the field private and adding a getter and a setter. All references to the original field need to be replaced with calls to the getter or the setter. Once a field is encapsulated, its implementation is then abstracted from the public interface of the class and can no longer be coupled to external code – freeing it to evolve and change independently of the interface. This abstracting decreases coupling to implementation and increases the maintainability of code.

Another simple abstraction refactoring is **Extract interface**. Performing an extract interface refactoring involves creating a new interface; copying one or more method signatures from an existing class to the new interfaces, then having that class implement the interface. This is usually done to decouple use of this class and is usually followed up by replacing references to the class with references to the new interface. This refactoring is often used in more complex refactorings, as we'll see in later chapters.

Reordering parameters is a simple refactoring whose process is well described by the name. Performing reorder parameters, as its name describes, involves reordering the parameters to a method. This refactoring is useful if you find that the order of the parameters of a method make it more prone to error (two adjacent parameters of the same type) or that you need to make the method signature match another (maybe newly inherited) method. If the method is referenced, the order in which the arguments are passed to the method would need to be reordered. Although a simple refactoring, conceptually this refactoring could lead to logic errors in code if not fully completed. If parameters that were reordered in the method signature had the same type, all references to the method would be syntactically correct and the code would recompile without error. This could lead to arguments being passed as parameters to a method that were not passed to the method before "refactoring". If this refactoring is not done properly it is no longer a refactoring because the external behavior of the code has changed! Reordering parameters is different from the previously mentioned simple refactorings, because if not completed properly they would all result in a compiler error.

Removing parameters is another simplification refactoring. It involves removing one or more parameters from a method signature. If the method is referenced, these references would need to be modified to remove the argument that is no longer used. This refactoring is often in response to code modifications where method parameters are no longer used. This could be the result of a refactoring or an external behavior change. With object-oriented languages, removing a parameter could cause a compiler error as removing the parameter could cause the method signature to be identical to an existing method overload. If there were an existing overload, it would be hard to tell which method references actually referenced the overload or the method with the newly removed parameter. In cases such as these, it's usually best to revert the change and reevaluate the two methods.

These simple refactorings are conceptually easy and aren't difficult to perform manually. Performed manually, these refactorings could easily involve many lines of code with a lot of repetitive code changes. With many repetitive actions and the possibility of introducing human error, many developers may tend to avoid performing these manual refactorings despite being simple. These refactorings have the potential to evolve the code in ways that make it easier to maintain it and add features to it. This makes for more robust code and enables you to more quickly respond to new requirements.

These simple refactorings are actually supported by Visual Studio® 2010. Visual Studio® 2010 automates these refactorings. If you rename a method with the rename method refactoring in Visual Studio® 2010 and there are hundreds of references to the method, Visual Studio® 2010 will find all references to them in the current solution and rename them all – automatically. This drastically reduces the friction of performing many of the refactoring building blocks.

In Chapter 2, we began performing these simple refactorings with the built-in refactoring tools in Visual Studio® 2010.

Simple refactorings are refactorings that are easy to understand, easy to describe, and easy to implement. Simple refactorings apply to software projects in just about every language – they transcend languages. Refactorings are like recipes or patterns; they describe (sometimes merely by their name) how to improve or change code. This simplicity with which they can describe change and the simplicity by which they are known has led to several taxonomies of refactorings. Most taxonomies catalogue simple refactorings; but some taxonomies can be found that include complex or specialized refactorings. Some examples of online taxonomies offer centralized databases of refactorings:

<http://www.refactoring.com/catalog/>

<http://industriallogic.com/xp/refactoring/catalog.html>

Technical debt

As we implement the required functionality and behavior to fulfill requirements, we have the choice of not fully changing the rest of the system to accommodate the change. We may make design concessions to get the feature done on time. Not improving the design to accommodate a single change may seem benign; but as more changes are made to a system in this way, it makes the system fragile and harder to manage. Making changes like this is called incurring technical debt. The added fragility and unmaintainability of the system is a debt that we incur. As with any debt, there are two options: pay off the debt or incur interest on the debt. The interest on the technical debt is the increased time it takes to correctly make changes with no adverse side-effects to the software system and/or the increased time tracking down bugs as changes made to a more fragile system that cause it to break. As with any debt, it is not always detrimental to your project, there can be good reasons to incur debt. Incurring a technical debt, for example, to reach a high-priority deadline, may be good use of technical debt. Incurring technical debt for every addition to the code base is not a good use of technical debt.

In the software development trenches

I've been a consultant in the software development industry for 15 years. I've worked with many different teams, with many different personalities, and many different "methodologies". It's more common than not for software development teams to tell me of nebulous portions of code that no one on the team is willing to modify or even look at. It performs certain functionality correctly at the moment, but no one knows why. They don't want to touch it because when someone has made changes in the past it has stopped working and no matter how many duct tape fixes they make to it, it never works correctly again.

Every time I hear this type of story, it always has the same foot note: no one has bothered to maintain this portion of code and no one bothered to try to fully understand it before or after changing it. I call this the "House-of-Cards Anti-pattern". Code is propped up by coincidence until it reaches a pre-determined height, after which everyone is told to not go near it for fear that the slightest movement may cause it to completely collapse. This is a side-effect of **Programming by Coincidence**. Had this nebulous code constantly been refactored to evolve along with the rest of the system or to improve its design, it's quite likely that these regions of code would never have become so nebulous.



Programming by Coincidence is a Pragmatic Programmer term that describes design or programming where code results in positive side-effects but the reason the code "works" is not understood. More information on this can be found here: <http://www.pragprog.com/the-pragmatic-programmer/extracts/coincidence>

Unlike the financial debt basis for this metaphor, it's hard to measure technical debt. There are no concrete metrics we can collect about what we owe: the increased time it takes to make correct changes to a hard-to-maintain code base without unexpected consequence. This often leads members of the development community to discount keeping a code base maintainable in favor of making a change with complete disregard for overall consequences. "It works" becomes their catch-phrase.

The option of rewriting

I'm not going to try to suggest that – while as developers we "refactor" all the time – it's the only option for major changes to a system. Rather than systematically evolve the code base over time, we could simply take the knowledge learned from writing the original code base and re-write it all on the same platform. Developers often embrace re-writing because they haven't been maintaining large blocks of the code, for various reasons. They feel that over time the changed requirements have caused them to neglect the underlying design and it has become fragile, hard to maintain, and time-consuming to change. They feel if they could just start anew they could produce a much better quality code base.

While re-writing may be a viable option and will most likely get you from point A to point B, it tends to ignore many aspects of releasing software that some developers tend to not want to think about. Yes, at the end of the re-write we might end up with a better code base and it might be just as good as having refactored it over the same amount of time; but there are many consequences to committing the software to a re-write.

Committing the software to a re-write does just that, commits the team to a re-write. Writing software means moving from less functionality to more functionality. The software is "complete" when it has the functionality that the users require. This means between point A and B the new software is essentially non-functional. At any given point of time, it can only be expected to perform a subset of the final functionality (assuming it's functional at all at that given point in time). This means the business has to accept the last released version of the software during the re-write. If the software is such a bad state as to be viewed as requiring a re-write then that usually means that last released version is in a bad state and user is not overly satisfied with it.

Re-writes take a long time. Developers can re-use the concepts learned from the original code and likely avoid some of the analysis that needs to go into developing software; but it's still a monumental task to write software from scratch. Re-writes are like all software development, the schedule hinges on how well the developers can estimate their work and how well they and their process are managed. If the re-write involves the same business analysts, the same developers, the same management, and the same requirements that the original – admittedly flawed – software was based upon, the estimates should be suspect. If all the same inputs that went into the flawed software go into the re-write, why should we expect better quality? After all, the existing software started out with a known set of requirements and still got to its existing state. Why would essentially performing the same tasks with the same process result in better software?

So, we know we are going to get to point B from point A, but now we know the business can't have new software until we get to point B and we now realize that we're not really sure when we're going to get to point B. The time estimates may not be scrutinized at the beginning of the re-write; but if the developers are unable to keep to the schedule they will most certainly be scrutinized when they fail to maintain the schedule. If the business has been without new features or bug fixes for an extended period of time, telling them that this timeframe is expanding indefinitely (and that is how they'll view it because they were already given a timeframe, and now it's changed) will not be welcome news.

I know what you're thinking, we can deal with the fact that the re-write means the software can't respond to market changes and new requirements for an unknown and significant amount of time by having two code streams. One code stream is the original code base and the other is the re-write. One set of developers can work on the original code base, implementing bug fixes and responding to market changes by adding features; and another set of developers can work on the rewrite. And this often happens, especially due to the business's response to the pushed timeframe. If they can't get their re-write right away, they'll need something with new features and bug fixes to keep the business afloat to pay for the re-write. If two code streams aren't created the software developers are right back to where they started, shoe-horning new features and bug fixes into the "pristine" code base. This is one of the very reasons why the developers feel the original code base had problems, the fact that the new features and bug fixes had caused them to neglect the design.

Maintaining two code streams make the developers feel like they're mitigating the risks involved with having to implement new features and bug fixes into their new design. It seems like everyone they should be happy. But, how is this really any different? Why would the business even accept the re-write if they're already getting what they want? How would these new features not make their way into the re-write? This is actually the same as having one code stream; but now we have to maintain two code streams, two teams, and manage change in two code bases and in two different ways. So, have we made the problem worse?

It's easy for developers to neglect the design of the system over time, especially if the design process is based on Big Design Up Front. With Big Design Up Front, the software development lifecycle stages are assumed to progress in a way similar to a waterfall, one stage can't continue until the previous stage has completed, for example, development can't begin until "Design" has been completed. This means that adding new behavior to an existing design means going back to the Design stage, coming up with a new design: documenting it, reviewing it, and submitting it for approval.

With a process like this, the management team quickly realizes that they can't make money if every evolution of the software requires a Big Design Up Front and they plead with developers to make small changes without Big Design Up Front to be responsive to customer and market demand – despite mandating Big Design Up Front at the start of the project. Developers are almost always happy to oblige because they don't like dealing with processes, and generally like dealing with documentation even less. Going back to analyzing and writing designs isn't what they really want to do, so they're happy to be asked to effectively just write code. But, the problem is that they often fail to make sure the rest of the software can accommodate the change. They simply duct tape it to the side of the existing code (like adding a feature, making sure it passes one or two change-specific use cases, and moving on). It's after months of doing this that the code base becomes so brittle and so hard to manage that no developer wants to look at it because it means really understanding the entire code base in order to make a change that doesn't cause something else to break.

A developer faced with a process like this is almost forced to fall back on a re-write in order to implement any major additions or changes to the software because that's essentially how the lifecycle is organized and what it promotes. They're certainly not presented with a process that guides or even rewards the developer for keeping on top of the entire code base. The reward system is based almost solely on delivery of functionality. It's not the developer's fault that the lifecycle is forcing them into a corner like that. Or is it?

Many developers fall back on management's lifecycle. "I'm just doing what I'm told" or "I didn't define the process". This is avoiding ownership in their work. The developer is avoiding any ownership of their work because they haven't been delegated responsibility for part of what goes into their work – the process. This is partially the fault of the developer, they need to take ownership of their work (the code) and understand what it is that only they really have control over. Getting into this situation is also the fault of management (or mismanagement) because it assumes that management only manages tasks and time estimates and leaves the details of "software development" to the software developers but ties their hands behind their backs by imposing a process by which they need to follow. Refactoring is a tool to help developers take ownership of the code – especially code they didn't originally author.

Unless we have an almost perfect handle on the code base that we're working with, we're unlikely to know all the implicit functionality of the system. Implicit functionality is unintended functionality that users have come to rely upon or can come to rely upon. This functionality isn't necessarily a bug (but users can come to rely upon bugs as well); but is concrete behavior of the system that users are using. This is never documented and it is a side-effect of the way the software was written (that is, not the way it was designed). I've actually worked with end-users that complained when a bug was fixed, because they communicated amongst themselves a process that included a work around for the bug. They were annoyed that they had to change this process because they were so used to doing it a certain way (the work around had essentially become a reflex to them).

The reason a re-write may be appealing is because the developers don't have a complete handle on the code base and therefore cannot have a complete handle on the implicit behavior. A re-write will neglect these implicit behaviors and almost always draw complaints from users. While re-writing is a viable technique, the applicable circumstances are rare. If a re-write is proposed for a project you are on, be sure it is thoroughly evaluated.

Working refactoring into the process

Writing software doesn't work well if you don't accept that the industry changes rapidly and that the software needs to respond to these rapid changes and those that its users require. If the act of writing a piece of software took a couple of days (or even a couple of weeks) we could get away with not having to evolve existing source to produce changes to software. If it were just industry and requirements changes, software development may be able to keep up with these demands. But, our industry contains many competitors, all vying for the same pool of new and existing customers. If a competitor releases software with new features that the software we're writing doesn't have, our software would be obsolete before it even got released. As software developers, we need to embrace change and be as responsive to it as possible. Refactoring helps us to achieve this.

It's inevitable; someone will ask for a new feature that the existing design hadn't accounted for. One way of dealing with this, as we've discussed, is to go back to the drawing board with the design. This is too costly due to the consequences of that fall-out. The opposite side of the pendulum swing is to shoe-horn the change in, ignore the design, and hope there are no consequences. There's a happy medium that can accommodate both changes to the design and make the customer happy. Simply account for refactoring work and inform the customer of the slight increase in time estimates. If this is done consistently, the design is kept up-to-date with what it is required to support and maintains a fairly constant ability to respond to change. Each new feature may require design changes, but now we've spread those design changes over time so that one single new feature doesn't require abandoning the entire design. Each new feature has a constant and limited design evolution aspect to it.

To clarify with a metaphor: there are two ways of maintaining your car. One is to be entirely responsive, add gas when the gauge reaches E and go to the mechanic when the red light comes on. When the red light comes on, this means there's trouble. The owner of a car can avoid this red light through regular maintenance. Changing the oil at prescribed intervals, performing routine checks at scheduled times, and so on, go a long way in maintaining the health of your car so that the red light never goes on and you avoid costly repair bills. Software development is much the same; you can neglect regular and preventative maintenance of your code base by ignoring technical debt. Ignoring technical debt could result in a costly "repair bill" if you're forced to pay your technical debt when you haven't scheduled it. "Regular maintenance" comes in the form of constant preventative refactoring. Time should be scheduled for refactoring code outside of *in response to adding a new feature*. Developers should regularly read the code looking for ways to refactor it and improve its structure. This has two benefits. One is that they're constantly up-to-speed on the code. Even if there are infrequent changes to portions of the code, they're still familiar with it because they're reading and understanding it on a periodic basis.

The other benefit is that the code is being kept up-to-date with the rest of the design and its changes. Keeping the code up-to-date in this way generally means the latest technologies and patterns are incorporated into the code in a timely fashion, keeping it that much more robust and reliable. When customers are informed of the amount of time work entails and this amount of time is consistently met, they're much more accepting of this than to be informed of the amount of time it takes for a re-write.

What to refactor

With most software development projects, there's a reasonably constant influx of new requirements or change requests. Depending on the system and the input from your user community, some or many of these requests may require code that is significantly outside what the design currently handles. These requests may be enough to keep your refactoring plate full for a long time.

But, what if your project is already seeing the side-effects of brittle design and hard-to-maintain code? Implementing requirements often introduces bugs in seemingly disparate areas of the code that take less than trivial amounts of time to fix that make estimating the effort to implement requirements less than accurate. Finding where to make code changes to implement requirements may not be obvious, or the changes end up requiring far-reaching changes to code across much of the project. Working on a complex software project often means this can be an everyday fact of life. But, working on a software team where most changes involve modifying code across much of the code base can introduce a friction that makes the time it takes to implement requirements much longer than necessary. Some people may think this is just a part of the software development process; but, because you're reading this book, you don't believe that.

New requirements that clearly don't fit in to the current design offer a clear hint at where to focus our refactoring effort. So, we know that refactoring may produce a more maintainable and robust code base; but, where do we start?

Refactoring to patterns

Many object-oriented code bases are perfectly functional. But these code bases don't consistently attempt to reuse concepts or attempt to use formally accepted patterns. A pattern is a description of communicating objects and classes that are customized to solve a general design problem in a particular context. One way to clean up code is to attempt to refactor individual parts of the code to use specific and applicable patterns. This is an excellent way of better understanding the intention of the code. Once specific patterns are implemented, the code then becomes that much more understandable. Industry-standard terms and code is then strewn throughout the code rather than project-specific buzzwords. The code becomes easier to consume by newcomers and thus easier to maintain. Chapters 5 through 9 deal specifically with examples of code that use concepts found in several common patterns and show how to refactor the code to make the pattern explicit and thus improve the chances of not repeating code.

Just as refactoring to patterns may make code easier to understand, less likely to repeat itself, and easier to maintain in general; forcing patterns into code for the sake of patterns will make code harder to understand and maintain. It's important to be sure that refactoring to a particular pattern adds value to the readability, understandability, and maintainability of the code. Shoe-horning a pattern where it does not belong will have the opposite effect. Make sure your desire to use a pattern hasn't overshadowed its suitability.

Refactoring to principles

In the race to get products out the door or to meet specific deadlines, programmers can often lose sight of principles in favor of functionality. While this isn't all bad, it could leave the code as a procedural ball of mud, or hard to maintain and understand in many other ways. After deadlines have been met, this is often a good time to reflect on the code, its design and structure, as it applies to principles. The code may not be overly object-oriented, for example. The code and the architecture may benefit from a review for SOLID principles. **SOLID** is an acronym for **Single Responsibility principle**, **Open-Closed principle**, **Liskov Substitution principle**, **Interface Segregation principle**, and the **Dependency Inversion principle**. There are several other object-oriented principles geared towards reducing coupling and complexity in source code and help keeping software development efforts focused more on adding value and less on responding to issues. Refactoring to principles is discussed in detail in Chapters 5 through 7.

Refactoring to principles and patterns is a design change. Although changes of this nature have positive side-effects (the impetus to implement them) they may also come with negative side-effects. A design change may decrease performance, for example. Any design change should be evaluated to ensure there are no unacceptable negative side effects before being accepted.

Code smells

Kent Beck introduced the term "code smells" (and formalized it along with Martin Fowler) to provide a way of communicating ways to detect potential problems that can occur in code. These problems are generally adverse side-effects of code that effectively works in at least one context. As with real-world smells, some smells may be tolerable by some people and not by others. There are no good smells with code smells, only varying degrees of bad smells.



Kent Beck and Martin Fowler formalized the term code smells in the book *Refactoring: improving the design of existing code*.

Code smells allow us to easily define common anti-patterns that can be prioritized by opinion or context. Depending on the person or the project in which they work, a code smell can be distinctly prioritized. Some people and teams may feel certain smells are unacceptable (maybe they consider them as *stenches*); while others feel the removal of a smell is just a nice-to-have; while still others may feel the same smell is nary an issue.

Code smells are an easily categorized, but personal means of describing the side-effects that someone discovered (and documented) of a particular type of code. So, code smell possibilities and the degree to which you may find them applicable are endless. Chapters 2 and 3 go into more detail about code smells and specific examples of refactoring code in response to common code smells that are generally accepted as having a high return on investment when properly removed.

Complexity

Reducing complexity is always a good reason to change code. Complex code is hard to understand; if code is hard to understand, it's hard for just anyone (or anyone at all) to fix bugs in the code or to add features to the code. But, how do you focus your efforts on complex code and how do you find complex code—after all, you want to fix code and not spend all your time searching for code to fix. Fortunately, there are many tools out there that will tell you how complex your code is.

You can get a gut feeling about how complex some code is by simply reading it. But, if you can't read all your code, code metrics can help. Software metrics are numerical measurements calculated by the structure and content of source code. Many software metrics focus on putting a numeric value on the complexity of code. Metrics like Depth of Inheritance, Class Coupling, and Lines of Code can help tell you how complex regions of code are. More obviously, metrics like Maintainability Index and Cyclomatic Complexity specifically address code complexity.

Chapter 6 goes into more detail about specific refactorings in response to specific complexity metrics.

Performance

Focusing on complexity can often indirectly help with the performance of code. But, you often want to focus specifically on performance of code. Fixing performance issues is almost refactoring by definition—you want to change the code without changing the external behavior of the code. Many applications have functionally with obvious need of performance improvements; and these are obvious areas of code you should focus upon. But, how do you measure improvement; and if you don't have obvious non-performance features, how do you find areas of code to focus your performance-related refactoring efforts? Luckily, there are many tools out there to gather performance metrics about executed code.

Kernel

Many software systems have a core subsystem that performs the majority of the work that the system does. If a subsystem was designed this way, it's likely called a kernel. Depending on the methodology used by the team, a Domain Model may exist that is very similar to a kernel—the code that clearly defines all the domain-specific entities and their logic that remain constant regardless of the type of the front-end, how integration points are implemented, and so on.

Focusing on the core of the system and making sure it's not complex, easy to understand, easy to change, and easy to add features too goes a long way in improving the responsiveness of a software team. This core of code often deals with proprietary information—the reason the business exists. People with experience with information and logic like this are usually hard to find. You may not be able to simply publish an ad for "C# developers with experience in Telematics" and expect to find many people locally to fill the position. Keeping the kernel simple to understand means you can get people without years of experience in your domain to change and add to the code.

Design methodologies

It's common for source code that has been around for any length of time to have been worked on by team members that have come and gone. Some of those team members may have been influential with regard to the design methodology used (or there may have been some "rogue" developers that deviated from the design methodology accepted by the rest of the team). With some design methodologies, this may not have a noticeable effect on the design of the code; but some design methodologies have fairly distinct effects on the design of the code. Domain-driven design, for example, suggests that domain entities be explicit in the code—usually an entity-to-class relationship. These entities often are completely decoupled from the rest of the system (user-interface, infrastructure, utility methods, and so on.) If the design of the system is to remain domain-driven, you may find that some classes may need to move to a different place, be decoupled from other classes, and so on. Depending on the level to which domain-driven has been implemented (or lacking thereof) the code may need to be more organized into layers. Chapter 8 details refactoring to layers. Other techniques attributed specifically to domain-driven design are detailed in Chapters 8 and 10. Specific patterns have been attributed to domain-driven design and details of those patterns can be seen in the chapters dealing with refactoring to patterns: Chapters 5 through 9.

Unused and highly-used code

Identifying how frequently code is used helps to tell you whether the refactoring effort will result in tangible results. One of the easiest refactorings is to simply delete unused code. But, without scouring the code, line-by-line, how do you find unused code or focus on highly-used code?

Luckily there are tools that will tell you how used code is. These tools are called Code Coverage tools. Much like performance metrics tools, they monitor executing code and tell you how frequently code, methods, and classes are used. Some static analysis tools can tell you about code, methods, and classes that are not referenced by other code—giving you information about unused code. This type of unused code will help focus your unused-code refactoring efforts, but can't tell you about all unused code. Code, methods, or classes may still be referenced by other code but may never be executed. Code Coverage tools will help tell you about this other type of unused code.

Refactoring in Visual Studio® 2010

The Visual Studio® 2010 IDE is much more than a solution/project management system with a text editor. It offers much functionality to aid the user in refactoring their code. Visual Studio® 2010 has had features for common refactorings since Visual Studio® 2005, such as extract method, rename member, encapsulate field, extract interface, and so on. (more detail about these refactorings can be seen in following chapters). Visual Studio® 2010 expands the refactoring palette that Visual Studio® 2010 has to offer by adding refactorings like Generate from Usage, TODO: get final list of any new refactorings.

Refactoring with Visual Studio®'s built-in automated refactorings doesn't have to be the limit of your refactoring efforts with Visual Studio® 2010. Depending on the edition of Visual Studio® 2010 you're using, you can work with other features of Visual Studio® 2010 to focus, prioritize, or introduce areas of your code to refactor.

Visual Studio® 2010 Premium offers several features that when coupled with a tangible refactoring effort, will help improve the quality and maintainability of your code base.

Static code analysis

Static code analysis is the analysis of software by an automated tool that does not involve execution of the code. This is most commonly analysis of the source code. The code is analyzed for common simple anti-patterns, logic errors, structure of individual statements, and so on. Visual Studio® introduced static code analysis in version 2005. The static code analysis in Visual Studio® was based on a Microsoft Internal project started by Brad Abrams and Krzysztof Cwalina called FxCop. FxCop was originally written as a means to analyze .NET software and enforce the guidelines detailed in the "Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries" book. FxCop is different from most static analysis tools available at the time in that it analyzed compiled code. This means it can't analyze code comments. FxCop was eventually subsumed by Visual Studio® and renamed to Code Analysis. Although FxCop is still available, it appears that it is no longer being maintained and any new functionality added to Code Analysis is not being merged into FxCop.



More information on FxCop can be found at <http://go.microsoft.com/fwlink/?LinkId=180978>

Visual Studio® 2010 Code Analysis includes 200+ rules for reporting possible design, localization, globalization, performance, security, naming, interoperability, maintainability, portability, and reliability improvements. Every software development team should review the rules available in Visual Studio® and decide what subset (or subsets, depending on types of applications) of rules should be applicable. The rules should then be prioritized so that the team is not overwhelmed with trying to remove many Code Analysis rule violations. The team should decide which rules can be used to help focus the refactoring effort. For example, the rule CA1502 Avoid Excessive Complexity might be corrected by one or more Extract Method refactorings.

Code metrics

Code metrics, although another form of static analysis, is a separate feature of Visual Studio® 2010. Code metrics is more commonly called 'software metrics' in the software industry. To a certain extent, some of the rules in Code Analysis are based on specific software metrics (like CA1502 Avoid Excessive Complexity – which is based on the Cyclomatic Complexity software metric). Software metrics are specific definitions about how to measure software in a certain way. These measurements generally deal with quality. Quality in software is measured in efficiency, complexity, understandability, reusability, testability, and maintainability.

Most software metrics are not easily actionable. For example, what is specifically actionable from a Maintainability Index of 55? With the Maintainability Index, the lower the metric the higher the estimated maintainability – but deciding what course of action to take based solely on 55 is nearly impossible. Some metrics may be easily actionable, like 1000 lines of code for `ClassX` may need `ClassX` to be refactored into multiple classes. We'll look more closely at how you can use code metrics to help focus the refactoring effort in Chapter 3.

Software metrics can be used in conjunction with other observances to focus where refactoring should be prioritized. Given `ModuleX` and `ModuleY` that both contain code that *has known performance issues*; if `ModuleX` has a Maintainability Index of 75 and `ModuleY` has a Maintainability Index of 50, it may be obvious that `ModuleX` needs more work and could benefit from refactoring first. But, it's a double-edged sword; you could easily decide that because `ModuleY` is more maintainable (because it's Maintainability Index is lower) that refactoring efforts should focus on code in `ModuleY` because you may be able to get a faster return on investment.

Chapters 6 and 7 include details about making use of software metrics to help prioritize refactoring.

Summary

Software projects are generally about writing and managing multi-release software with no fixed end-of-life and that evolves noticeably over time. This chapter has outlined how systematic refactoring efforts help in the evolution and management of software to avoid it from becoming brittle and keeping it robust in the face of change. Refactoring helps software developers be more amenable to accepting new requirements.

Writing software, although a science, is not always done perfectly. Many priorities affect how software is written. Sometimes, software isn't designed or written to use the best technique. Software that doesn't use the best technique (but is otherwise functional) is considered to have acquired technical debt. An increased debt load can lead to bankruptcy. By focusing code improvements on refactoring to reduce debt, software developers can avoid reaching a debt load that requires a re-write.

Refactoring efforts can have a specific impetus, or they may simply be part of a general effort to improve the maintainability of the software under development. There are many aspects that can help focus the refactoring efforts whether the impetus is specific or not.

In some code-bases there may be no specific focus on design – focusing almost solely on functionality. Code-bases like this have inconsistent design styles and are hard to read, understand, and modify. By refactoring to specific design principles, code can be modified to improve its consumability, robustness, and changeability.

Refactoring to patterns help in maintaining software that is easily understandable and consumable to peers in the software industry. Use of patterns reduces the concepts consumers of code are required to understand before they can understand the overall intent of the code.

Refactoring code to remove code smells can help focus and prioritize the refactoring effort. By prioritizing standard smells and creating proprietary smells, maintainability of code can be enhanced focusing on improvements with the highest returns.

There are many aspects to how software is written and designed that is orthogonal to its functionality but directly related to its usability. Prioritizing refactoring efforts based on usability metrics like performance will give end users the same functionality, only faster.

There are many formulas for analyzing source code to gather specific metrics about the static structure of the code. These metrics are generally geared towards detecting attributes that are side effects of problems. These problems are often related to complexity. There are many tools that automatically gather metrics about code. Complex code has proven to be a major aspect contributing to hard-to-maintain software projects. Complex code reduces the ability of software to change to implement new features and increases the risk that these changes cause other problems. By gathering and prioritizing these metrics, code improvement can focus on refactoring complex code.

Focusing on improving code that when improved will result in the highest return, is an attractive option. By prioritizing change to areas of the code that are highly used or highly dependent can realize more noticeable benefits to the refactoring effort.

Refactoring is about preventative maintenance. As with any maintenance, a systematic approach to it can make the process more tolerable and in many ways it can make it enjoyable. Put some thought into what and where you want to improve your source code and you can avoid drudgery and focus on the value-added parts of your software.

2

Improving Code Readability

The readability of code is an important aspect of how quickly code can be consumed and understood. Most of the time this understanding is by someone else, but if you don't write code so that it's easy to consume and understand, you can be bitten by its lack of readability when you return to the code sometime after you read it. Although how easily code can be read can be dependent on the style of the code – the bracing standard, whitespace: tab versus spaces, and so on – "code readability" here is about how well the code details its intentions. While code style is an important aspect of reading and producing code, and working with a team of developers, any developer with much experience working within a team and with other people's code should be able to read code with various "styles". Besides, there's a myriad of tools to "prettify" code so that it abides by specific style guidelines or rules. Prettifying code is the process of changing the style of the code without changing the flow of the logic in which the code implements. This is similar to refactoring; but refactoring specifically changes the logic without changing the external behaviour. Prettifying code results in the binary executable output with the same executable content, whereas refactoring usually does not.

The readability of code is an important aspect to the ability of code to be evolved over time. If someone can't read the code, it's hard for them to change it in response to the changing environment to which it applies. The code is a view of the requirements for which it fulfils. The code documents what the system or application is supposed to do and how it should do it.

Although code that works provides functionality to users, if that code can't keep up with changing technologies, changing environments, or the changing atmosphere that the user requires, it – and its writers – will be replaced by something that can. There are many aspects to code that makes it inviting to change. If only one person understands the code, it's a risk, and at some point the code can't be changed if the single person that understands the code becomes too busy or is no longer available to change the code. For someone else to read, absorb, and comprehend hard-to-understand code and not change the code to something more useful is time-consuming. Time that doesn't go into directly adding value to a product is time and funds that are wasted.

If readability wasn't important for the capacity of code to be understood and its logic to be comprehended, there wouldn't be various tools and utilities to obfuscate source code.

In this chapter, we'll detail how to improve code readability with Visual Studio® in the following ways:

- Built-in Visual Studio® refactorings
- Rename field
- Rename property
- Rename method
- Rename local variable
- Rename class
- The smell of code
- KISS principles
- You ain't gonna need It
- Tracking code changes

Built-in Visual Studio® refactorings

The introduction of basic refactorings to Visual Studio® (2005) was a watershed moment for Visual Studio® users. It reduces many pain points for developers involving the difficulty of maintaining code and evolving it over time.

Visual Studio® has a few built-in abilities for you to refactor code to make it more readable. Visual Studio® 2010, regrettably, comes with the loss of one of our tried-and-true refactorings. The Visual Studio® 2010 refactoring companions drops from seven to six. The Promote Local Variable to Parameter refactoring is survived by Rename Identifier (local, method, member, and class), Extract Method, Encapsulate Field, Extract Interface, Reorder Parameters, and Remove Parameters.

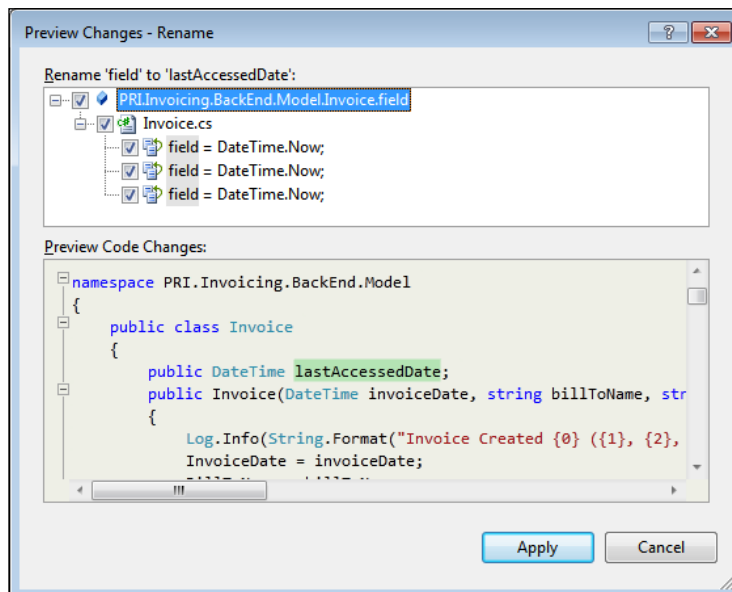
Refactoring code is much more than a small set of basic automatic refactorings; but most complex refactorings are based on basic refactorings. Some complex refactorings include an ordered aggregation of many basic refactorings.

Rename identifier refactoring

Probably the most common built-in refactoring you might use (whether you use it with existing code or not) is the Rename Identifier refactoring. The rename identifier refactoring in Visual Studio® 2010 is context sensitive. It shows up in many places in Visual Studio® as "Rename..." or sometimes just "Rename". Depending on what type of identifier you're renaming, you have different options for renaming it; so, I'll briefly describe each rename refactoring separately.

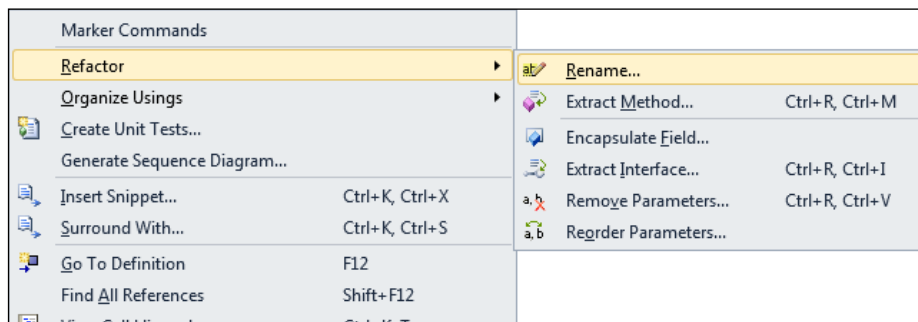
Rename field

If you're not used to using an automated refactoring tool then you've likely only ever renamed a field by simply typing a new name right into the source code. When you do this with Visual Studio® a smart tag is displayed underneath the identifier you just edited. Clicking on this smart tag (or entering the keystroke *Shift+Alt+F10* or *Ctrl+.*) will allow two items "Rename 'X' to 'Y'" and "Rename with preview...". "Rename 'X' to 'Y'" will go ahead and rename all current references from 'X' to 'Y'. "Rename with preview..." presents the "Preview Changes - Review" form that details all the places that the field is currently referenced and shows how the renamed reference will look. The preview form looks like this:

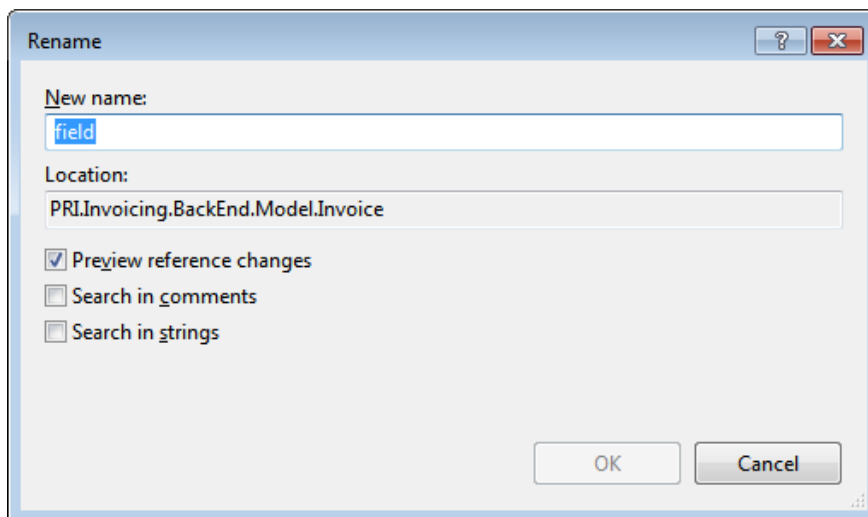


The top tree view details each place where a reference will be changed and the line of code that changes. By clicking on a particular node in that tree, it will show the context of the change in the **Preview Code Changes** region. You can opt to uncheck a particular node that you don't want to be renamed by un-checking the line in the tree view. This is useful if you've selected **Search in comments** or **Search and strings** and it has found a word that matches the name of your identifier that you don't really want changed.

The most common way of renaming a field without editing it manually is the use of the right-click context menu in the code editor. To rename a field in code, right-click the field and select **Refactor\Rename....**



The **Rename** form will be displayed:



When this form is first displayed, **New name** is populated with the current name of the field. Renaming an identifier causes all references to that identifier in the solution to be renamed as well. You can preview the renaming of all those references by leaving the **Preview reference changes** checkbox checked. If **Preview reference changes** is checked, the **Preview Changes–Rename** form will be displayed.

You can also position the caret over the field you want to rename (clicking it does this nicely) and select **Refactor\Rename** from the main menu. This brings up the same Rename form described earlier. If you display the Class View, you can also right-click the field in the lower Members pane and select **Rename...**, which brings up the Rename form described earlier.

If you're using an edition of Visual Studio® 2010 that supports class diagrams, you can also rename a field by right-clicking the field in the diagram or in the Class Details pane and selecting **Refactor\Rename...** This brings up the Rename form as we saw in the Code Editor. You can also click on the field in the Class Details pane and simply type a new name for the field. This bypasses the Rename form and renames all references to the field in the entire solution without preview and ignores comments and strings.

You can also rename a field from within the Object Browser. Although there is no Refactor item on the right-click context menu on a field in the Object Browser, the main Refactor menu contains **Rename...** when you select a field in the Object Browser.

If the field you wish to rename is a field added to a Form in the Form Designer, you can also rename the field from the Properties window. This performs a rename in the same way as renaming a field from the Class Details pane in the Class Designer – it does not bring up the Rename form and renames the field and all references and ignores comments and strings.

If the field you are renaming is an override or is overridden by another class, Visual Studio® 2010 will inform you of this fact when you attempt to rename the field (after pressing **OK** in the Rename form) and prompt you as to whether you'd like to continue or not.

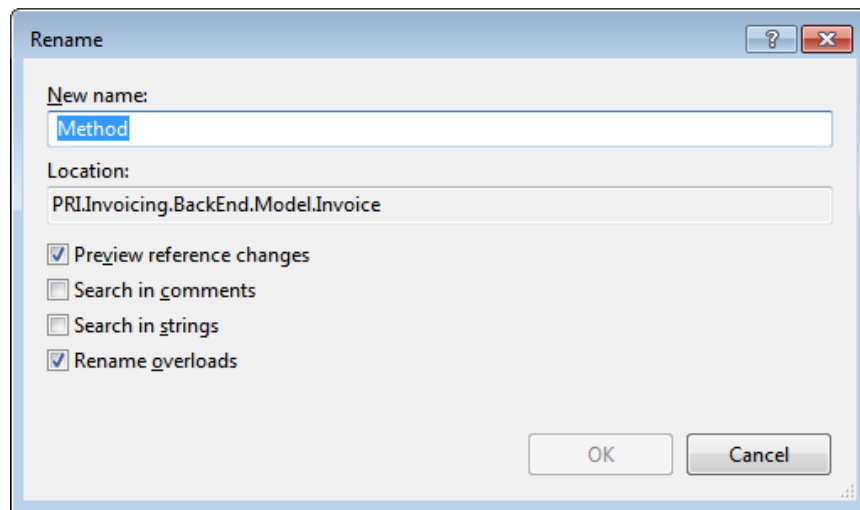
Rename property

Rename property works the same as Rename field. You can rename from almost all the same places: right-clicking the property in the Code Editor, the Class Designer, and the Class View, from the main Refactor menu when selected in the Code Editor, the Class Designer, the Class View, and the Object Browser. When designing a form you can't implement a form element as a property, so you can't rename properties from the Properties window within the Form Designer.

If the property you are renaming is an override or is overridden by another class, Visual Studio® 2010 will inform you of this fact when you attempt to rename the property (after pressing **OK** in the Rename form) and prompt you as to whether you'd like to continue or not.

Rename method

Methods are handled slightly differently than fields and properties. Unlike fields and properties, methods can have overloads. Although a method can be renamed in all the same places as a field or property, the Rename form allows you to specify whether to rename overloads as well, as shown in the addition of "Rename overloads" in the Rename form:



If the method you are renaming is an override or is overridden by another class, Visual Studio® 2010 will inform you of this fact when you attempt to rename the method (after pressing **OK** in the Rename form) and prompt you as to whether you'd like to continue or not.

Rename local variable

Since local variables are only visible in the Code Editor, the options for renaming a local variable are more limited. They may be renamed inline, by right-clicking the variable in the code and selecting **Refactor\Rename...** or by positioning the caret over the variable name in the declaration and selecting **Refactor\Rename...** from the main menu. Renaming a local variable performs just like renaming a field or property.

Rename class

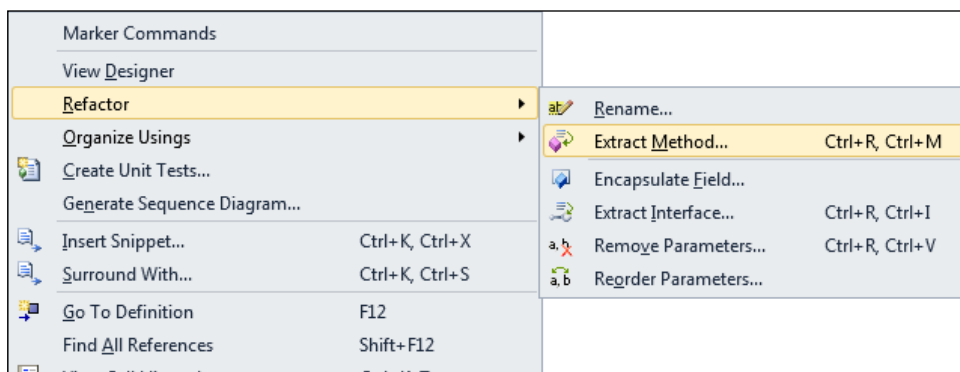
The options for renaming a class are more plentiful than renaming other identifiers. You can rename a class inline in the code editor (and get the same Smart Tag options), right click the identifier name in the class declaration and select **Refactor\Rename...**, select **Refactor\Rename...** from the main menu after positioning the caret over the class name in the declaration, from within the Class View, from within the Class Designer, and from within the Object Browser. Also, you can rename a class from within the Solution Explorer by renaming the file in which it resides if the name of the file matches the name of the class.

When renaming the name of a file in the Solution Explorer that contains a class by the same name, Visual Studio® 2010 will display the following prompt: "You are renaming a file. Would you also like to perform a rename in this project of all references to the code element `ClassName`?". Selecting **Yes** effectively performs a rename refactoring on the class name which includes the class definition and all references to that class name.

Extract Method refactoring

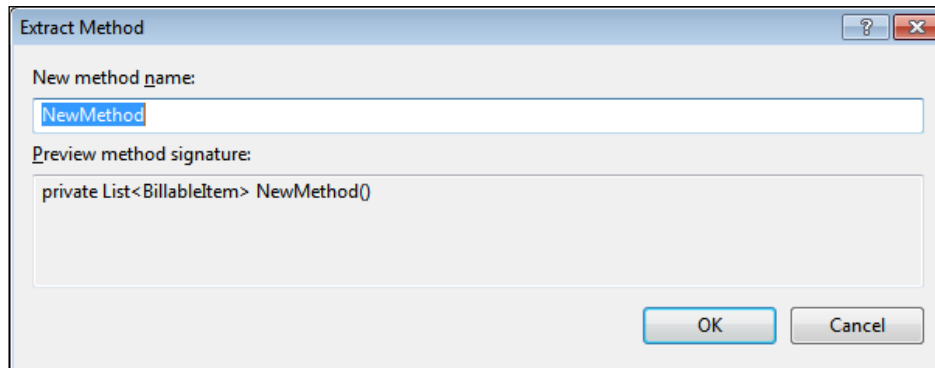
Another important automatic refactoring in your refactoring tool belt is the **Extract Method** refactoring. The **Extract Method** refactoring creates a new method in a class from a fragment of code from an existing method in the same class and replaces the existing code with a method call to the new method and includes passing any necessary parameter should the code not be self contained.

The Extract Method refactoring is context sensitive; it requires a valid code fragment selection. The Extract Method refactoring can be invoked by right-clicking a valid code fragment selection and choosing **Refactor\Extract Method...**



Extract method may also be invoked by choosing **Refactor\Extract Method...** from the main menu after selecting a valid code fragment, or (since the Extract Method refactoring has a keyboard shortcut in the default Visual Studio® 2010 C# keyboard layout) pressing *Ctrl+R, Ctrl+M*.

Upon invoking the **Extract Method** refactoring, you are presented with the **Extract Method** form:



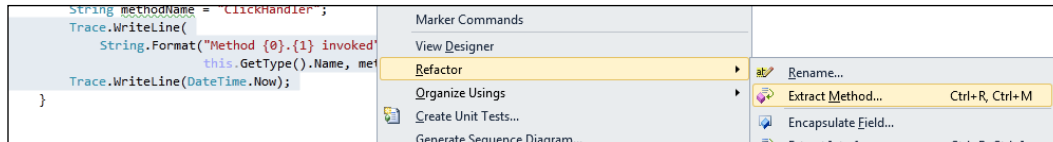
You can enter the name you would like to give your new method in the **New method name** field and the **Preview method signature** section displays an updated preview of the signature your method will take when created.

When you press **OK** Visual Studio® will create a new method, move the selected code to the body of the method, create any appropriate parameters, and return the appropriate value, if applicable. It will then place a call to the new method where the selected code fragment used to be, creating any local variables to receive the return value and passing any appropriate arguments to the new method.

For example, let's say we have a button `Click` event handler in our form class that when called outputs the fully-qualified name of the method and the date/time it was invoked to the debug output. We may do something like this:

```
private void button1_Click(object sender, EventArgs e)
{
    const String methodName = "button1_Click";
    Trace.WriteLine(
        String.Format("Method {0}.{1} invoked",
                      this.GetType().Name, methodName));
    Trace.WriteLine(DateTime.Now);
}
```

When we would like to do something else in this method, we may find that this code hinders the readability of the method and the generic nature of the code suggests moving it to its own method would be an improvement. We would then select the lines that call `Trace.WriteLine`, right click, and choose **Refactor\Extract Method...**:



If we entered `TraceMethodInvocation` for the name of the method, we'd end up with a new method that looks similar to the following:

```
private void TraceMethodInvocation(String methodName)
{
    Trace.WriteLine(
        String.Format("Method {0}.{1} invoked",
            this.GetType().Name, methodName));
    Trace.WriteLine(DateTime.Now);
}
```

And our `button1_Click` method would look like this:

```
private void button1_Click(object sender, EventArgs e)
{
    const String methodName = "button1_Click";
    TraceMethodInvocation(methodName);
}
```

No fuss, no muss; we now have a tidy reusable method `TraceMethodInvocation` that we can reuse and avoid repeating ourselves.

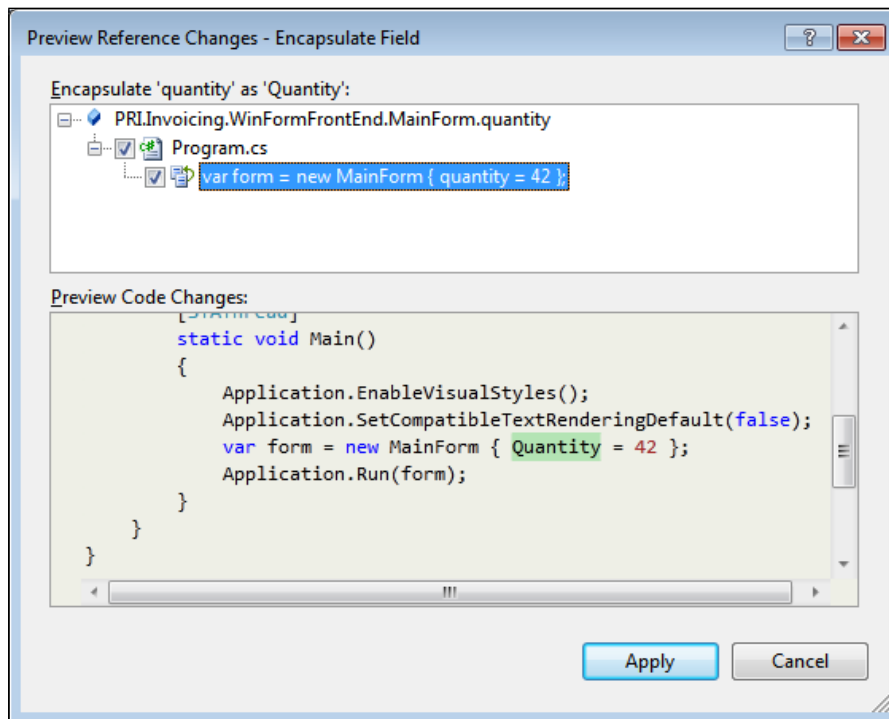
Encapsulate Field refactoring

The Encapsulate Field refactoring creates a new property with the same name as the field being encapsulated capitalized (and a number added, if a member of the same name exists; for example, if there was a property `Quantity` and field `quantity`, and I performed the Encapsulate Field refactoring on the field `quantity` it would result in a property named `Quantity1`). It also replaces all references to the field (excluding references to the field in the containing classes constructors) or only references to the field outside the containing class with the name of the new property. The resulting property makes use of the original field and if that field was public, it is made private.

Much the same as the Rename field refactoring, you have the option of previewing all the changes Visual Studio® will make, as well as whether or not to look for the name of the field in comments and in strings. For example, the following code creates an instance of a MainForm class and initializes the quantity field to 42:

```
var form = new MainForm { quantity = 42 };
```

If we choose to invoke the Encapsulate field refactoring and chose to preview the reference changes, the **Preview Reference Changes - Encapsulate Field** form would look like the following if we selected the node in Program.cs:



This shows that "quantity" will be replaced with "Quantity" in the object initializer for MainForm. Pressing **Apply** at this point will apply all the check changes.

The Encapsulate Field refactoring is useful if you're trying to abide by the *Framework Design Guidelines and Design Guidelines for Class Libraries* or responding to Code Analysis warning CA1051: Do not declare visible instance fields.



Krzysztof Cwalina and Brad Abrams (2005) *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley.

More information about field design guidelines in Design Guidelines for Developing Class Libraries can be found at <http://msdn.microsoft.com/en-us/library/ms229057.aspx>.

More information about CA1051 can be found at <http://msdn.microsoft.com/en-us/library/ms182141.aspx>.

The smell of code


We first introduced **Code Smells** in Chapter 1, if you're not familiar with **Code Smells**, it may sound strange. The metaphor "something doesn't smell right" is the gist of code smells. Experienced developers that read code can often notice subtle problems with the code that they may or may not be able to put their finger on directly. This is a "Code Smell". Code Smells are based on viewing the code, so it's arguably one of the major means of improving code readability. Many code smells are particularly geared towards the ability to read code. "Long Method" and "Large Class" code smells, for example, have a side effect of being hard to read because the reader needs to page up and down in the code to follow what it is doing.

Duplicate code smell

Duplicate code is probably the most prevalent code smell. Andy Hunt and Dave Thomas actualized the principle **Do Not Repeat Yourself** (or **DRY**) in the book *Pragmatic Programmer*. It states "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system". Duplicate code indicates that there is a prevalent concept in your domain that hasn't been made explicit in the code.

There are many reasons why we would want to avoid duplicate code. The simplest is that you simply have two of the same thing. When you have two versions of the same thing they can evolve independently. The evolution of one may need to be duplicated in the other, which can be easily missed in the duplicate code. You rarely think of trying to find similar code throughout your code base whenever you change a particular piece of code.

Duplication can be at the textual level: two regions of code contained are textually identical. Or, the duplication may be at a conceptual level: two regions of code do the same thing. Textually identical code is much easier to find. There are actually tools that look for duplicate code like this – tools like SolidsDD – Duplicate Code Detector and Clone Detective. Finding (or noticing) conceptually identical regions of code is much harder. We won't get into finding the duplicate code since we're focusing on refactoring code based on known issues.

 More information about SolidsDD – Duplicate Code Detector can be found at <http://www.solidsourceit.com/products/SolidSDD-code-duplication-cloning-analysis.html>
More information about Clone Detective can be found at <http://clonedetectivevs.codeplex.com/>

Fortunately it's easy to refactor duplicate code.

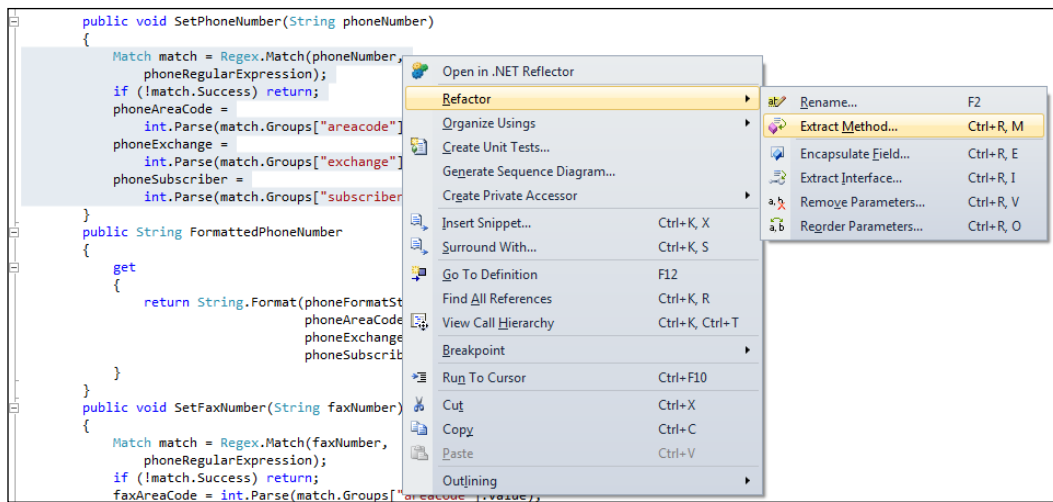
One form of code duplication is duplication that can be found within one class. Code has been duplicated in two or more methods within a class. This indicates that a new method should be added to the class and that method should be called instead of executing the duplicate code.

Duplicate code in a class

A common form of duplicate code is a logic that appears more than once within a single class. Part of the refactoring can be automated with the Extract Method refactoring. To complete the refactoring of duplicate code, the remaining code must be manually replaced with a call to the newly extracted method.

The following `ContactInformation` class is a container for contact information. In this snippet we're dealing with phone number and fax number information. The `SetPhoneNumber` and `SetFaxNumber` methods parse a string and set appropriate field values. Both of these methods perform very similar logic; they differ only in the destination fields. To refactor these methods to eliminate the duplicate code, we can perform the following steps:

1. Select the code within `SetPhoneNumber`.
2. Right-click the selection.
3. Choose **Refactor\Extract Method...**



4. Enter ParsePhoneNumber.
5. Edit ParsePhoneNumber to accept three ref int parameters: areaCode, exchange, subscriber.
6. Edit ParsePhoneNumber replacing phoneAreaCode with areaCode, phoneExchange with exchange, and phoneSubscriber with subscriber.
7. Edit SetPhoneNumber to add ref phoneAreaCode, ref phoneExchange, ref phoneSubscriber to the arguments passed to ParsePhoneNumber.
8. Edit SetFaxNumber to call ParsePhoneNumber with the faxNumber, faxAreaCode, faxExchange, and faxSubscriber fields:

```

/// <summary>
/// ContactInformation class that encapsulates informatino
/// about a contact like phone number and address
/// </summary>
public class ContactInformation
{
    private int phoneAreaCode, phoneExchange, phoneSubscriber;
    private int faxAreaCode, faxExchange, faxSubscriber;

    /// <summary>
    /// Sets the phone number for this contact
    /// </summary>
    /// <param name="phoneNumber"></param>
    public void SetPhoneNumber(String phoneNumber)
    {

```

```
// parse the phone number
Match match = Regex.Match(phoneNumber,
    phoneRegularExpression);
if (!match.Success) return;
phoneAreaCode =
    int.Parse(match.Groups["areacode"].Value);
phoneExchange =
    int.Parse(match.Groups["exchange"].Value);
phoneSubscriber =
    int.Parse(match.Groups["subscriber"].Value);
}

/// <summary>
/// Gets the phone number as formatted text
/// </summary>
public String FormattedPhoneNumber
{
    get
    {
        return String.Format(phoneFormatString,
            phoneAreaCode,
            phoneExchange,
            phoneSubscriber);
    }
}

/// <summary>
/// Sets the fax number for this contact
/// </summary>
/// <param name="faxNumber"></param>
public void SetFaxNumber(String faxNumber)
{
    // Parse the phone number
    Match match = Regex.Match(faxNumber, phoneRegularExpression);
    if (!match.Success) return;
    faxAreaCode = int.Parse(match.Groups["areacode"].Value);
    faxExchange = int.Parse(match.Groups["exchange"].Value);
    faxSubscriber =
        int.Parse(match.Groups["subscriber"].Value);
}

/// <summary>
/// Gets the fax number formatted as a text
```

```

    /// </summary>
    public String FormattedFaxNumber
    {
        get
        {
            return String.Format(phoneFormatString, faxAreaCode,
                                faxExchange, faxSubscriber);
        }
    }

    private const String phoneFormatString = "{0} {1}-{2}";
    private const string phoneRegularExpression =
        @"\\((?<areacode>[0-9]{3})\\) *(?<exchange>[0-9]{3})-
        (?<subscriber>[0-9]{4})";

    //...
}

```

The refactored class will look something like the following:

```

/// <summary>
/// ContactInformation class that encapsulates information
/// about a contact like phone number and address
/// </summary>
public class ContactInformation
{
    private int phoneAreaCode, phoneExchange, phoneSubscriber;
    private int faxAreaCode, faxExchange, faxSubscriber;

    /// <summary>
    /// Sets the phone number for this contact
    /// </summary>
    /// <param name="phoneNumber"></param>
    public void SetPhoneNumber(String phoneNumber)
    {
        ParsePhoneNumber(phoneNumber, ref phoneAreaCode,
                        ref phoneExchange, ref phoneSubscriber);
    }

    /// <summary>
    /// Gets the phone number as formatted text
    /// </summary>
    public String FormattedPhoneNumber
    {
        get
        {
            return String.Format(phoneFormatString,

```

```
        phoneAreaCode,
        phoneExchange,
        phoneSubscriber);
    }
}

/// <summary>
/// Sets the fax number for this contact
/// </summary>
/// <param name="faxNumber"></param>
public void SetFaxNumber(String faxNumber)
{
    ParsePhoneNumber(faxNumber, ref faxAreaCode,
        ref faxExchange, ref faxSubscriber);
}

/// <summary>
/// Gets the fax number formatted as a text
/// </summary>
public String FormattedFaxNumber
{
    get
    {
        return String.Format(phoneFormatString, faxAreaCode,
            faxExchange, faxSubscriber);
    }
}

private const String phoneFormatString = "{0} {1}-{2}";
private const string phoneRegularExpression =
    @"\((?<areacode>[0-9]{3})\) *(?<exchange>[0-9]{3}) -
    (?<subscriber>[0-9]{4})";

private static void ParsePhoneNumber(string number,
    ref int areaCode, ref int exchange, ref int subscriber)
{
    var match = Regex.Match(number, phoneRegularExpression);
    if (!match.Success) return;
    areaCode = int.Parse(match.Groups["areacode"].Value);
    exchange = int.Parse(match.Groups["exchange"].Value);
    subscriber =
        int.Parse(match.Groups["subscriber"].Value);
}

//...
}
```

Duplicate code in multiple classes

Duplicate code can also span multiple classes. Refactoring this is a little bit different than refactoring code within a single class. When extracting a method from a block of code, that method needs to be accessible by both classes in order to correctly remove the duplication. It may seem simple to just leave the extracted method in one of the classes and reference it from the other. But, if the logic spans two classes then it's likely that neither of the two classes should really have the responsibility of this logic.

If the two classes share a common base class then the base class could contain the logic. In which case, we'd perform the same steps as we did with code duplicated in a single class, then we'd perform **Pull Up Method** refactoring. We'd then replace the logic in the other original class with a method call to that new method.



Pull Up Method refactoring is when a method is moved from a subclass to a super class. This refactoring is not an automated refactoring built into Visual Studio® 2010.

If the two classes don't share a common base class then it might be worth analysing whether they should. If they should, we can perform the same refactoring steps.

It has been my experience that classes in this scenario are rarely suited to derive from the same super class. In this scenario you'll need a third, new, **Service** class to contain the method that will be extracted. A Service class is useful when consolidating duplicate code amongst projects. It may be necessary to create a new project to contain infrastructure-related types like Service classes.



A **Service** class is a class that contains no state and performs logic that normally wouldn't be the responsibility of other classes.

Performing this refactoring is very similar to our single class example, except that a new Service class is created and when the selected code is extracted to a method, a **Move Method** refactoring is manually performed on that method moving it to the new Service class and the method call resulting from the extract method refactoring is changed to call the method on the new service class. Since a Service class contains no state, the extracted method may be made static, in which case no instance of the Service class need be created.



Move Method refactoring is when a method is moved from one class to another unrelated class.

For example; with our original `ContactInformation` class example, if we had another class like the following:

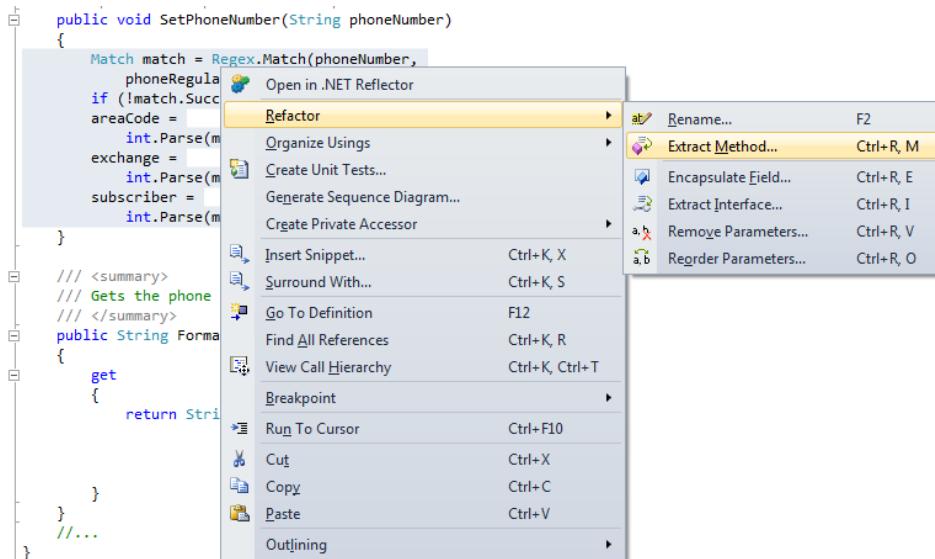
```
/// <summary>
/// Encapsulates cellular device information line phone number
/// </summary>
public class CellularDevice
{
    private int areaCode, exchange, subscriber;
    private const String phoneFormatString = "{0} {1}-{2}";
    private const string phoneRegularExpression =
        @"\\((?<areacode>[0-9]{3})\\) *(?<exchange>[0-9]{3}) -
            (?<subscriber>[0-9]{4})";

    /// <summary>
    /// Sets the phone number for this contact
    /// </summary>
    /// <param name="phoneNumber"></param>
    public void SetPhoneNumber(String phoneNumber)
    {
        Match match = Regex.Match(phoneNumber, phoneRegularExpression);
        if (!match.Success) return;
        areaCode =
            int.Parse(match.Groups["areacode"].Value);
        exchange =
            int.Parse(match.Groups["exchange"].Value);
        subscriber =
            int.Parse(match.Groups["subscriber"].Value);
    }

    /// <summary>
    /// Gets the phone number as formatted text
    /// </summary>
    public String FormattedPhoneNumber
    {
        get
        {
            return String.Format(phoneFormatString,
                areaCode,
                exchange,
                subscriber);
        }
    }
    //...
}
```

This class contains logic that parses a phone number in the same way as `ContactInformation`. In this case, our steps may look like this:

1. Select the code within `ContactInformation.SetPhoneNumber`.
2. Right-click the selection.
3. Choose **Refactor\Extract Method...**



4. Enter `ParsePhoneNumber`.
5. Edit `ParsePhoneNumber` to accept three `ref int` parameters: `areaCode`, `exchange`, `subscriber`.
6. Edit `ParsePhoneNumber` replacing `phoneAreaCode` with `areaCode`, `phoneExchange` with `exchange`, and `phoneSubscriber` with `subscriber`.
7. Create a new class named `PhoneNumberParser`.
8. Manually move `phoneRegularExpression` to `PhoneNumberParser`.
9. Manually move `ParsePhoneNumber` from `ContactInformation` to `PhoneNumberParser`.
10. Edit `ContactInformation.SetPhoneNumber` to invoke `PhoneNumberParser.ParsePhoneNumber` instead of just `ParsePhoneNumber`.
11. Edit `ContactInformation.SetPhoneNumber` to add `ref phoneAreaCode`, `ref phoneExchange`, `ref phoneSubscriber` to the arguments passed to `ParsePhoneNumber`.

12. Edit `ContactInformation.SetFaxNumber` to call `PhoneNumberParser.ParsePhoneNumber` with the `faxNumber`, `faxAreaCode`, `faxExchange` and `faxSubscriber` fields.
13. Edit `CellularDevice.SetPhoneNumber` to call `PhoneNumberParser.ParsePhoneNumber`.

The result of this refactoring may look something like the following:

```
/// <summary>
/// Service class to parse phone numbers
/// </summary>
public static class PhoneNumberParser
{
    private const string phoneRegularExpression =
        @"\\((?<areacode>[0-9]{3})\\) *(?<exchange>[0-9]{3}) -
            (?<subscriber>[0-9]{4})";

    /// <summary>
    /// Parse a phone number <paramref name="number"/> into
    /// <paramref name="areaCode"/>
    /// <paramref name="exchange"/>
    /// <paramref name="subscriber"/>
    /// </summary>
    /// <param name="number"></param>
    /// <param name="areaCode"></param>
    /// <param name="exchange"></param>
    /// <param name="subscriber"></param>
    public static void ParsePhoneNumber(string number,
        ref int areaCode, ref int exchange, ref int subscriber)
    {
        var match = Regex.Match(number, phoneRegularExpression);
        if (!match.Success) return;
        areaCode = int.Parse(match.Groups["areacode"].Value);
        exchange = int.Parse(match.Groups["exchange"].Value);
        subscriber =
            int.Parse(match.Groups["subscriber"].Value);
    }
}

/// <summary>
/// ContactInformation class that encapsulates informatino
/// about a contact like phone number and address
/// </summary>
public class ContactInformation
{
    private int phoneAreaCode, phoneExchange, phoneSubscriber;
```

```
private int faxAreaCode, faxExchange, faxSubscriber;

/// <summary>
/// Sets the phone number for this contact
/// </summary>
/// <param name="phoneNumber"></param>
public void SetPhoneNumber(String phoneNumber)
{
    PhoneNumberParser.ParsePhoneNumber(phoneNumber,
        ref phoneAreaCode, ref phoneExchange,
        ref phoneSubscriber);
}

/// <summary>
/// Gets the phone number as formatted text
/// </summary>
public String FormattedPhoneNumber
{
    get
    {
        return String.Format(phoneFormatString,
            phoneAreaCode,
            phoneExchange,
            phoneSubscriber);
    }
}

/// <summary>
/// Sets the fax number for this contact
/// </summary>
/// <param name="faxNumber"></param>
public void SetFaxNumber(String faxNumber)
{
    PhoneNumberParser.ParsePhoneNumber(faxNumber,
        ref faxAreaCode, ref faxExchange, ref faxSubscriber);
}

/// <summary>
/// Gets the fax number formatted as a text
/// </summary>
public String FormattedFaxNumber
{
    get
    {
        return String.Format(phoneFormatString, faxAreaCode,
            faxExchange, faxSubscriber);
    }
}
```

```
    }
    private const String phoneFormatString = "{0} {1}-{2}";
    private const string phoneRegularExpression =
        @"\\((?<areacode>[0-9]{3})\\) *(?<exchange>[0-9]{3})-(?<subscriber>[0-9]{4})";

    private static void ParsePhoneNumber(string number,
        ref int areaCode, ref int exchange, ref int subscriber)
    {
        var match = Regex.Match(number, phoneRegularExpression);
        if (!match.Success) return;
        areaCode = int.Parse(match.Groups["areacode"].Value);
        exchange = int.Parse(match.Groups["exchange"].Value);
        subscriber = int.Parse(match.Groups["subscriber"].Value);
    }
    //...
}
/// <summary>
/// Encapsulates cellular device information line phone number
/// </summary>
public class CellularDevice
{
    private int areaCode, exchange, subscriber;
    private const String phoneFormatString = "{0} {1}-{2}";
    private const string phoneRegularExpression =
        @"\\((?<areacode>[0-9]{3})\\) *(?<exchange>[0-9]{3})-(?<subscriber>[0-9]{4})";

    /// <summary>
    /// Sets the phone number for this contact
    /// </summary>
    /// <param name="phoneNumber"></param>
    public void SetPhoneNumber(String phoneNumber)
    {
        PhoneNumberParser.ParsePhoneNumber(phoneNumber,
            ref areaCode, ref exchange, ref subscriber);
    }

    /// <summary>
    /// Gets the phone number as formatted text
    /// </summary>
    public String FormattedPhoneNumber
    {
        get
        {
```

```
        return String.Format(phoneFormatString,
                               areaCode,
                               exchange,
                               subscriber);
    }
}
//...
}
```

Duplicate code in construction

It's not entirely uncommon for instance constructors to contain initialization code that is copied and pasted from one constructor to another. Generally, instance constructors don't have much logic so this may seem satisfactory – but now we know why duplicated code is not satisfactory. *Effective C#* has an item devoted to this topic titled *Utilize Constructor Chaining* that details that classes should utilize constructor chaining by placing appropriate construction logic in each constructor and chain to another constructor to let it continue with construction logic (and not use an initialization method).



Bill Wagner, *Effective C#: 50 Specific Ways to Improve Your C#* 2005, Addison Wesley.

This eliminates duplicate construction logic. The existence of several constructors implies that there is more than one field or property that can optionally have default values. **Accumulative Construction** is a form of constructor chaining that makes most efficient use of constructors without having to resort to initialization methods. Initialization methods are troublesome because only constructors can modify readonly members. Containing all initialization code within the constructors and ensuring that no initialization code is duplicated and instead changed to another constructor, you achieve a structure that results in one constructor being called after another until the object is fully constructed. I call this **Accumulative Construction**. With a properly designed class this doesn't get out of hand. If you find that it's extremely difficult to achieve this you've most likely got a Large Class smell and should look at the intended responsibilities of the class, the responsibilities it has taken on, and break up the class into multiple classes that each take on a coherent responsibility. See Long Method Smell.

Advanced duplicate code refactoring

Sometimes multiple regions of code that are conceptually similar are different in very distinct ways. For example, a class may contain a collection of objects that it iterates over and performs a different action in several places in your code. This could be as simple as a `foreach` loop, or it could be as complex as a `for` loop. Despite the simplicity of `foreach` and `for` loops, you may find that you need to iterate that collection in a different way at some point. When there are multiple loops in your code, it's time-consuming and error-prone to change them all. If the code that contained the loop was not coupled to how the collection was iterated, you'd have a much easier time modifying the iteration code. One way of performing this decoupling is to create a method that iterates the collection and invokes a delegate.

For example, let's say we have the following two methods in a class named `Invoice`: `CalculateTotalTax` and `CalculateTotal`. `CalculateTotalTax` calculates the total tax for this invoice and `CalculateTotal` calculates the total amount of this invoice.

```
float CalculateTotalTax()
{
    Decimal result = 0M;
    foreach (InvoiceLineItem invoiceLineItem in Items)
    {
        result += (Decimal)invoiceLineItem.CalculateTax();
    }
    return (float)result;
}

float CalculateTotal()
{
    Decimal result = 0M;
    foreach (InvoiceLineItem invoiceLineItem in Items)
    {
        result += (Decimal) invoiceLineItem.CalculateSubTotal();
    }
    return (float)result;
}
```

These two methods have one thing in common: they both iterate through the invoice's line items performing some action with each item. We could remove the responsibility of knowing how to iterate over all invoice line items from these methods and decouple it from `IEnumerable` by introducing a method that accepts an `Action<T>` delegate and invokes the delegate for each the iteration:

```
void PerformActionOnAllLineItems (Action<InvoiceLineItem>action)
{
    foreach (InvoiceLineItem invoiceLineItem in Items)
```

```
    {
      action(invoiceLineItem);
    }
  }
```

And we can then change our two methods as follows:

```
float CalculateTotal()
{
  Decimal result = 0M;
  PerformActionOnAllLineItems(
    ili =>
      result += (Decimal) ili.CalculateSubTotal());
  return (float)result;
}

float CalculateTotalTax()
{
  Decimal result = 0M;
  PerformActionOnAllLineItems(
    ili => result += (Decimal) ili.CalculateTax());
  return (float)result;
}
```

Or, if you prefer anonymous methods over lambdas:

```
float CalculateTotal()
{
  Decimal result = 0M;
  PerformActionOnAllLineItems(delegate(InvoiceLineItem ili)
  {
    result += (Decimal) ili.CalculateSubTotal();
  });
  return (float)result;
}

float CalculateTotalTax()
{
  Decimal result = 0M;
  PerformActionOnAllLineItems(delegate(InvoiceLineItem ili)
  {
    result += (Decimal) ili.CalculateTax();
  });
  return (float)result;
}
```

We've now taken two identical `foreach` loops that implicitly perform some logic on each line item and consolidated them into a single method whose name explicitly details its intention.

Unfortunately, we can't take advantage of any built-in automated refactorings in Visual Studio® 2010 to help us with this refactoring. This particular type of refactoring is entirely manual.

Long method smell

Doing one thing and one thing only is a vital rule of object oriented design. But, it doesn't mean that one thing should be constrained to one method. Methods that perform specific and complex logic can easily become long. The method works perfectly fine but in order to understand the method much reading is required.

In order to refactor a long method the logic should be analyzed, looking for anything that can stand on its own—or isn't directly the responsibility of the method under inspection. Those snippets of generic code can then be extracted into their own methods via Extract Method. The process of refactoring a large method is essentially the same as refactoring duplicate code, except you skip the step of replacing the second instance of the logic with a method call to a newly extracted method.

Once a long method has been refactored you may find that many of the extracted methods aren't necessarily a responsibility of the class they are in. You should review whether a new Service class should be created to be responsible for this related logic.

Code comments smell

Programmers are often taught that they must comment their code within methods. Some development teams mandate code comments. Comments aren't bad, but they can become a crutch for coders. Coders can rely on the comments to explain what the code does rather than making code easier to understand and easier to read.

Comments are just that, human-readable comments put into the code by the coder to make a comment about something. Most of the time the comments explain the code. Comments don't affect how software behaves. Comments aren't checked syntactically by the compiler and they are bound to the code by location. Comments can easily become out of date or dislocated from the code they originally provided information about, making the code actually harder to understand.

The code comment smell is not suggesting that comments should not be added to code at all. There are many reasons for comments that don't apply to explain how code works. A comment may be necessary to explain a decision that was made in the implementation of the code. Or, a comment may be necessary to point out a reference to external documentation that was used when writing the code. There are many reasons for comments; the point here is that they shouldn't be the sole mechanism for explaining code.

Tim Ottinger describes comments as *apologies*. They are apologies for not writing the code more clearly. Code that needs comments for it to be better understood should be refactored so the code is then self-explanatory. Fortunately, code with comments can be easily refactored with refactorings like Extract Method.

Comments should describe *why*, not *what*. The code should self-explain the *what* and a comment should be added if it's not clear *why* it was coded that way.

The first example of code comment smells refactoring that we will see is a commented block of code. The comment explains what the block of code does. The block of code can be refactored into a method with the Extract Method refactoring and the name of the method would be a summary of the comment. For example, the following code writes circle with the radius `radius` metrics area and circumference with an arbitrary `BinaryWriter`:

```
static void WriteCircleMetrics(BinaryWriter writer,
    float radius)
{
    // circle area
    writer.Write((float)(Pi * (decimal)Math.Pow(radius, 2)));
    // circle circumference
    writer.Write((float)(Pi * (decimal)(radius * 2)));
}
```

This would be refactored with the following steps:

1. Select the text `(float)(Pi * (decimal)Math.Pow(radius, 2))`.
2. Right-click the selection.
3. Choose **Refactor\Extract Method**.
4. Enter `CalculateCircleArea` and click **OK**.
5. Select the text `(float)(Pi * (decimal)(radius * 2))`.
6. Right-click the selection.

7. Choose **Refactor\Extract Method**.
8. Enter `CalculateCircleCircumference` and click OK.
9. Deleting the comments.

The refactored code would look something similar to the following:

```
public static float CalculateCircleArea(float radius)
{
    return (float)(Pi * (decimal)Math.Pow(radius, 2));
}

public static float CalculateCircleCircumference(float radius)
{
    return (float)(Pi * (decimal)(radius * 2));
}

static void WriteCircleMetrics(BinaryWriter writer,
    float radius)
{
    writer.Write(CalculateCircleArea(radius));
    writer.Write(CalculateCircleCircumference(radius));
}
```

This particular type of refactoring lends itself to situations where you want to reuse logic. The assumption in the above is the code to calculate the area and circumference appeared in multiple places and the refactoring results in several places where `CalculateCircleArea` and `CalculateCircleCircumference` are being invoked. Had that not been the case, the code could have been refactored to use local variables instead to avoid the need for comments:

```
static void WriteCircleMetrics(BinaryWriter writer,
    float radius)
{
    float area = (float)(Pi * (decimal)Math.Pow(radius, 2));
    writer.Write(area);
    float circumference = (float)(Pi * (decimal)(radius * 2));
    writer.Write(circumference);
}
```

The variable names now perform the same task the prior comments did, but now they are syntax checked by the compiler. The drawback of this method is that there is no built-in automated refactoring for this in Visual Studio® 2010, and code like this is more likely to be refactored to not needlessly require local variables.

A comment can be used to provide information about many things. The comment could be explaining a **Magic Number**.



A **Magic Number** is a constant literal value in code. This value is generally numeric (hence the "Number" part of Magic Number) but could apply to any non-numeric literal value (like strings). It is "Magic" because its existence in the code has special meaning that isn't explicitly clear.

Refactoring a magic number is straightforward. A constant field can be added to a class and all instances of the magic number can be replaced with the constant field. Unfortunately Visual Studio® 2010 doesn't have a built-in automated refactoring feature to convert a literal value to a constant member field. So, the process basically involves performing a search-and-replace. I recommend using regular expressions to limit the search to the whole word. For example, if I'm replacing the number with the constant `TheAnswer` then I would enable regular expressions in the Find and Replace form and search for "`<42>`" so I don't find numbers like 1424. The following two methods use the literal `3.14159265358979323846264338327m` for PI.

```
public static float CalculateCircleArea(float radius)
{
    // 3.14159265358979323846264338327m is PI
    return (float)(3.14159265358979323846264338327m *
        (decimal)Math.Pow(radius, 2));
}

public static float CalculateCircleCircumference(float radius)
{
    // 3.14159265358979323846264338327m is PI
    return (float)(3.14159265358979323846264338327m *
        (decimal)(radius * 2));
}
```

Once refactored, the code would end up looking similar to the following:

```
private const decimal Pi= 3.14159265358979323846264338327m;

public static float CalculateCircleArea(float radius)
{
    return (float)(Pi * (decimal)Math.Pow(radius, 2));
}

public static float CalculateCircleCircumference(float radius)
{
    return (float)(Pi * (decimal)(radius * 2));
}
```

The example of creating a constant field for Pi may not be the most useful example because there is `Math.PI` in the framework. The value-added here is that our `Pi` uses decimals and is thus more accurate.

Dead code

Dead code is code that isn't used by any other code. When you're writing frameworks and class libraries it can be difficult to track down dead code because there's no automated way of finding dead code in Visual Studio®. Dead code in frameworks and class libraries is usually restricted to private methods that aren't being used, or blocks of code that don't get executed because parameters it depends upon are never passed into the code. For the sake of maintainability, it's best to remove dead code.

The Code Analysis feature aids in finding dead code by warning about unused local variables and unused private code.

Intention-revealing design

Much of what we've detailed thus far has been an effort to refactor code to be more intention-revealing in order to improve readability and thus maintainability. Intention-revealing design is designing code that can stand on its own with little or no documentation on how it works or what it does. Variables should have a name that explains what is stored in the variable and how it is used or "Rename refactoring" should be performed on them. Methods should have a name that explains what will be performed when the method is executed and how it should be used, or a Rename refactoring should be performed on them. Methods should do one thing that is a responsibility of the class that contains the method, or a new class should be created and a Move Method refactoring performed on the method. Classes should have a name that explains what the class models and hints at how it should be used.

You ain't gonna need it

"You ain't gonna need it" (or YAGNI for short) is a principle that gained an established foothold in the Extreme Programming community. The principle centers around the tendency of some software to increase functionality for reasons other than the features. "Featuritis" – the symptom of some software to increase features simply to satisfy a check-list or some market collateral – is related to *you ain't gonna need it*.

"You ain't gonna need it" gives us a criteria by which to focus our refactoring efforts. If we know we don't need a particular feature, we can make better decisions about what may or may not be dead code. Code that supports a particular feature, for example, that we don't need will become dead code if we decide we *ain't gonna need* that feature.

Code in a code base that supports a feature that isn't needed, or performs actions that "might be needed in the future" increases the amount of code that needs to be read. In order to effectively evolve a code base over time, the code within it needs to be understood. Having more code than necessary to read and understand reduces the readability of code as a whole.

Detail focusing on value-add and not what "might be". Focus on requirements and don't increase scope just because a concept might include something.

For example, upon analysis of a system it may have been decided that a Customer class is required to model the system. Upon writing the Customer class the developer decided that Customer is a type of person and created the base class Person. Since this class modeled a person, the developer added applicable properties like age, weight, and height. If we were providing a framework that provides a generic way of modeling a person in many different contexts these properties may be useful. In this particular context, we're only modeling a customer, and as cliché as it may sound, we don't need to view them as a person. Creating the Person base class and adding properties to it that weren't going to be needed by the system is a waste of time. That code needed to be written, tested, possibly documented, and debugged, all for no value added. When changes to the system need to be made, the person doing the change needs to read and understand this unneeded code before they can accurately and reliably make the modification. This reduces the maintainability of the system because changes to the system take longer than needed.

You ain't gonna need it isn't limited to unnecessary features or features with little functional value; it can also apply to the way developers design software. In environments where communication with the end-user or the target audience doesn't exist, developers can work in a fictional world where they're trying to fulfill the requirements of a mythical user. This mythical user demands everything. Since this isn't a real person, they can't be asked if they need something, or what priorities to use for functionality that may be actually needed. Developers will tend to design software that tries to do everything for everyone. This is a problem because, with this attitude, software tends not to do anything very well and is so bloated it's hard to.

Every developer is guilty of some level of over-designing. It's easy to view something in a different context to which it is going to be used and add sensible features and functionality because it makes sense in that other context. Writing functional and usable software is hard work. We need to narrow our efforts on what the users need and the users need to prioritize what they want. Requests without prioritization are requests that are all prioritized as low priority.

You ain't gonna need it brings our refactoring efforts to the functional level. We need an established communication channel with our users. Ideally, we need a relationship with them. Sometimes this can be hard, but the rewards are priceless. If we know what the user really wants (which may not be what they've asked for) and we know how to communicate with the user to gather that information, we can focus our efforts on features and code that provides the most value. We may *think* a particular feature, particular code, or particular logic is interesting, but we're projecting our needs onto the software that we're writing to fulfil someone else's needs. This leads to dissatisfaction of users and can lead to communication problems and worse still, cancelation of software development efforts because of the opinion that the software being produced isn't adding value to the user.

Refactoring *you ain't gonna need it* is effectively the same as refactoring dead code.

KISS principle

KISS, "Keep It Simple Stupid" (or the less pejorative "Keep It Short and Simple") is a principle that suggests favoring simplicity over complexity. Simple code is easier to read and consume.

There are many code smells that deal with detecting complexity and refactoring it to simpler code. These smells deal with specific areas of complexity such as class-level complexities, method-level complexities, and so on. Simplicity can apply in many other places.

Computer language syntax has a subset that deals with a more general context and sometimes a subset that deals with more specific contexts. C# is no exception. There have been additions to the language for specific scenarios. Lambdas, for example, are a means towards functional programming that promote side-effect-free programming and avoid having to specify the details of the algorithm. Lambdas are a key part of the LINQ story. While you can use LINQ without lambdas, it's hard to avoid lambdas with complex LINQ statements.

Lambdas are a form of anonymous methods; so they can be used wherever anonymous methods can be used, which in turn can be used wherever a delegate is expected. But, because the syntax allows it in many places doesn't mean that's an appropriate use of it and more importantly, means the code is very readable. The following code (and I've encountered similar code to this in the wild – despite its contrived appearance) is perfectly legal:

```
public void Execute(int x, int y, String text)
{
    Action<int, int, String> validator =
        (xPos, yPos, textValue) =>
    {
        if (xPos > maxWidth)
            throw new ArgumentOutOfRangeException("xPos");
        if (yPos > maxHeight)
            throw new ArgumentOutOfRangeException("yPos");
        if (textValue == null)
            throw new ArgumentNullException("textValue");
    };
    validator(x, y, text);
    //...
}
```

The `Execute` method starts with some code to validate the arguments it was given. This particular implementation assigns a statement lambda to a delegate named `validator`. The validation is then executed when the `validator` delegate is invoked.

Although the addition of `validator` makes the intention of the validation code explicit; its use introduces lots of extra syntax that adds no value. Sure, as a programmer you get to use syntax with that new-car-smell, but you sacrifice readability for absolutely no extra value.

It's much more readable to use standard validation idioms, such as:

```
public void Execute(int x, int y, String text)
{
    if (x > maxWidth)
```

```
        throw new ArgumentOutOfRangeException("x");
    if (y > maxHeight)
        throw new ArgumentOutOfRangeException("y");
    if (text == null)
        throw new ArgumentNullException("text");
}
```

These idioms are easily recognizable because they are common and thus much more readable.

There are other ways of making code overly complex.

Keeping track of code changes

Code that is changing is code that can't be used. Your refactoring efforts may get pre-empted at any time with changes that are important to the viability of the software product. Without some ability to save all your changes and return the code to the state it was in when it was given to the customer, you run the risk of accidentally publishing an incomplete refactoring resulting in a defect (which could result in lost customer data). Sure, you could back up the code that was used on the last build and work with a copy of the code. This is almost do-able if you're the only developer. But, if you have more than one release of your software this means having a copy for each release. If you have a customer that finds a defect in a release older than the last release and the fix entails making a change to code that has already been changed; you have to deal with manually merging the code into the latest code. If you work with a team of developers, or plan on working with a team of developers, this strategy is simply not feasible.

It's extremely important to use a tool to manage changes to the code it has to be something that not only keeps track of what was done, who did it, and when it was done, but also keeps track of milestones and the differences from version to version. Source Code Control systems are an important part of software development. They keep track of changes, let you manage code streams and the changes to code streams, manage merging, and will handle locking if needed.

Check-in often

Now that you're using a Source Code Control system, it's important to maximize the use of that system. There's a principle called "check-in often". This principle effectively means, that as soon as you have something working, you should check it in. The Source Code Control system acts as a backup, so if you check-in often you're less likely to lose a change. But, it's more than that. The more often you check-in, the better granularity you have in your changes. If you made change A, change B, then change C, and realized that change B is actually better than change C, and you had checked in each of those changes then you could simply go to your Source Code Control system and ask for change B. You could always rely on memory to go back to change B, and that might work if change C was made fairly close in time change B. But, if you're like me, sometimes the time between changes can get long enough for me to forget the context and the details of what I was changing and it's almost impossible for me to truly remember how to get the source code back to the state it was in at the end of change B. Splitting hairs, maybe; but we're not dealing with our own property or our own time. This is the property and time of our employer; we should be doing everything we can to protect this property and not waste their expensive time.

Removing unused references

Another code change that improves readability in some cases is removing unused references. Visual Studio® includes the ability to optimize `using` directives and remove unused `using` directives.

When you create a Windows Form class, for example, it automatically includes `using` directives that the Visual Studio® team thought would be useful to a majority of users. There are eight `using` directives added to the top of your Windows Form class. The code generated by Visual Studio® only really needs one `using` directive (`using System.Windows.Forms;`) but likely will use two (`using System;` as well). Depending on your design philosophies and the requirements of your class, you may be unlikely to need the other six `using` directives. Removing unused `using` directives does not change the way the code behaves externally.

To remove unused references simply right-click on any `using` directive and choose **Organize Usings/Remove Unused Usings** and all unnecessary `using` directives will be removed. At this point you can then analyze what assemblies are referenced in the project and decide to remove them (and gain a slight improvement in your application's load time) without causing compile errors.

Summary

How readable the code is affects how easily the code can be read, refactored, or updated. Refactoring to address readability is useful for code bases or parts of the code that people find difficult to change because it's hard to understand.

Visual Studio® 2010 contains several built-in automated refactorings that help refactor a code base to improve code readability. We've detailed several aspects of hard-to-read code and detailed the refactorings that can improve readability. We have also detailed that making changes to code requires tracking. By using a Source Code Control system the changes to the code base are tracked, and that if Check In Often is followed, the code base can be returned to a state before any change.

We've introduced code smells as a way of detecting problems in code. We only scratched the surface of code smells, there are dozens of code smells and more are continually being defined.

3

Improving Code Maintainability

In the previous chapter, we discussed code readability and how refactoring with a goal to improving readability helps make code easier to understand and easier to change. In this chapter, we'll continue on a similar note with code maintainability. We'll continue our exploration of code smells and we'll discuss various topics related to code maintainability and how to refactor to improve code maintainability, including the following:

- Automated unit testing
- Feature Envy code smell
- Contrived Complexity code smell
- Don't Repeat Yourself
- Inappropriate Intimacy code smell
- Lazy Class code smell
- Detecting maintainability issues

Code maintainability

Code maintainability is very dependent on code readability. Code readability is based on some fairly hard and fast rules such as *fit methods within one or two screens*, *use intention revealing names*, *one class per file*, and so on. Other aspects of code maintainability involve what the code base is actually trying to do.

I'm a firm believer in *if it ain't broken, don't fix it* and it may seem like much of what we talk about with refactoring is trying to fix something that isn't broken. If what we're refactoring wasn't evolving over time and was functionally complete and working, I would agree. But, it's unheard of that software doesn't change after a particular release. As a consultant I often parachute into projects for a limited time and leave; so, technically I'm not *modifying the code and evolving the design over time*. *But, I believe it's important to think of the other developers and try to leave the campsite in a better state than when I arrived.* It's easy to be selfish and see only the short-term life of what you're working on; but, if you're reading this book, you're better than that.

What makes code more maintainable, to a certain degree, differs from project to project. There are some common areas that cause maintainability problems on software projects and we'll include those as we focus on refactoring to improve maintainability.

We know that our code base is going to evolve over time, and we should have a pretty good idea where it's going to evolve and how it's going to evolve. If you've been on the project for some time, you've likely got a gut feeling about where the project is headed. You've also have had a pretty good idea of where there are problem areas and have at least a high-level view of what is causing some of those problems. If you're new to the project, you've likely been asked, at least at a conceptual level, how the software needs to change. You've probably also been told about some problem areas. Reading this book, you've probably got a particular project in mind (maybe one you're currently working on) and may have very specific areas you're looking to improve. I obviously can't specifically address those first, but I'll try to start with the most common areas.

Some of what makes a code base readable also overlaps with what makes a code base more maintainable. Just as code smells helped us with readability, code smells will also help us with maintainability. To recap: code smells are specific symptoms that indicate specific potential problems in code that someone previously has fixed. The Large Method smell, for example, addresses readability, but dealing with a Large Method can also deal more specifically with maintainability. In this chapter, we'll also delve deeper into other code smells like Contrived Complexity, Feature Envy, and Inappropriate Intimacy.

This is probably a good point to define what we mean by maintainability. Maintainability implies that you have to maintain the code; you have to maintain the code to fix bugs, add features, and to support fixing bugs and adding features. Maintainability is about code's ability to accept change without introducing adverse side-effects. Clearly if we're fixing bugs and adding features we're expecting specific side-effects. So, refactoring for maintainability means changing the code to better accept change without introducing unwanted side-effects. We've all worked on code that seems fragile—you make what seems to be an innocuous change in one part of the code and something somewhere else breaks (or otherwise has an adverse side-effect). Or, we've worked with code that is really hard to make a simple change to because it means changing so many pieces of code that (at least when we're making the change) seem like they shouldn't need to be changed. So, what we're really trying to do when we refactor for maintainability is to reduce coupling. There are other areas of maintainability, but decoupling gives us the most value from our effort.

At this point we've described various ways of modifying code without modifying how it behaves. We're gradually describing more far-reaching refactoring that starts to delve more into design changes rather than code reorganization. But, what if we don't do something right and we actually introduce an adverse side-effect? Now is probably a good time to discuss ensuring how our refactoring efforts result in bugs for users.

Automated unit testing

Automated unit testing validates and verifies that individual units of code perform correctly based on constraints, requirements, pre-conditions, and post-conditions. It's rare that developers write code without doing any sort of testing. If you're reading this book, you're not a developer that doesn't test their code, right? We've got a code base that we can be reasonably sure has been developer-tested to at least some level. But, we're going to make changes to that code and we want it to act exactly how it did before we made those changes. So, how do we assure that? Automated unit testing allows us to create tests that verify that our code is doing what it's expected to do and run the tests any time we want. Once we can run them any time we want, we can continually run them (nightly, on every check-in/commit, build, and so on).

Automated unit testing allows us to know when a change we've made has created an adverse side-effect. Until we've performed all the refactorings we'd like to reduce fragility, we're likely going to introduce adverse side-effects during our refactoring efforts—despite our good intentions and our advanced skill level.

This may be starting to sound like much work. From a frame of mind not accustomed to explicit and tasked refactoring efforts and existing automated unit-testing, this whole refactoring thing is intimidating. It's important to remember that refactoring (including unit-testing) supports the whole development effort. Refactoring should never overshadow the fact that we're trying to deliver software, despite its level of maintainability.

Refactoring should always try to avoid blocking the evolution of the software; it needs to promote the evolution of the software. There may be times when the work involved in a particular refactoring could take an extended period of time, but the work should be organized as atomically as possible. That means changes shouldn't take too much time and shouldn't leave the software in a non-working state for very long. While the software is being refactored and in an unknown state it should never be checked-in to the source code control system in a way that affects the main stream or branch and other developers. It's important to remember that our refactoring efforts are gradual. You should never expect to be able to perform all your refactoring at once (or that no further refactoring will be needed once you've done the refactoring you want to do today). Unit tests that test all the code don't need to exist before we refactor and we don't need to finish all our refactoring before we fix bugs and add features (regardless of how nice that would be).

Before we perform a refactoring we need to know how the code that will be the focus of our refactoring works and performs today. If there is not an existing test, we need to then create a test that detects what the code has done in order to test that it still does that. The physical act of writing unit tests may introduce a need to further refactor our code in order to focus our testing or to actually perform a certain test. This is to be expected and is often a good way of focusing our refactoring efforts. If you're still not sure where to start refactoring, try writing a unit test to verify and validate a certain piece of code – like a method. If you find you can't focus the test on that specific method without getting a bunch of other code you don't want or don't need to test, refactor the code so you can.

Much of the effort to create unit tests for code rely on the code being maintainable; so, completing unit tests often fulfil our more maintainable code desire. Let's have a look at some code we would like to be able to test and see how we can refactor it to create individual tests that verify and validate specific post-conditions.

Let's go back to an invoicing example. To produce an invoice based on a collection of billable items involves a series autonomous logic. Depending on the type of customer or the type of services, an invoice will (among other things) need to sub-total each item based on price and quantity, create an invoice sub-total, calculate taxes, and calculate a final total. We can easily validate and verify that an invoice has done these tasks correctly by simply reading an invoice. Doing this depends on producing a readable invoice. *We* could write a unit test that takes readable invoice with known input values, performs some **Optical Character Recognition (OCR)** on it, determines where the output values are, reads the output values, then validates if the output values are correct. But, that's a huge amount of work and is likely to be pretty slow. If we want to automatically run that test during, say, the build or check-in, we could severely impact our development process. We definitely don't want to do that. A better way is to simply run code giving it a known input and getting an output and validating that output against that known value. In order to do that, our code needs to be structured in a way that makes that possible. Let's say we're unlucky and someone has written invoice generation like this:

```
/// <summary>
/// Generate a viewable invoice to the display
/// device represented by <paramref name="graphics"/>
/// </summary>
/// <param name="graphics"></param>
public void GenerateReadableInvoice(Graphics graphics)
{
    graphics.DrawString(HeaderText,
        HeaderFont,
        HeaderBrush,
        HeaderLocation);

    float invoiceSubTotal = 0;
    PointF currentItemLocation = LineItemLocation;
    foreach (InvoiceLineItem invoiceLineItem in InvoiceLineItems)
    {
        float lineItemSubTotal =
            (float)((decimal)(invoiceLineItem.Price
            - invoiceLineItem.Discount)
            * (decimal)invoiceLineItem.Quantity);

        graphics.DrawString(invoiceLineItem.Description,
            InvoiceBodyFont,
            InvoiceBodyBrush,
            currentItemLocation);

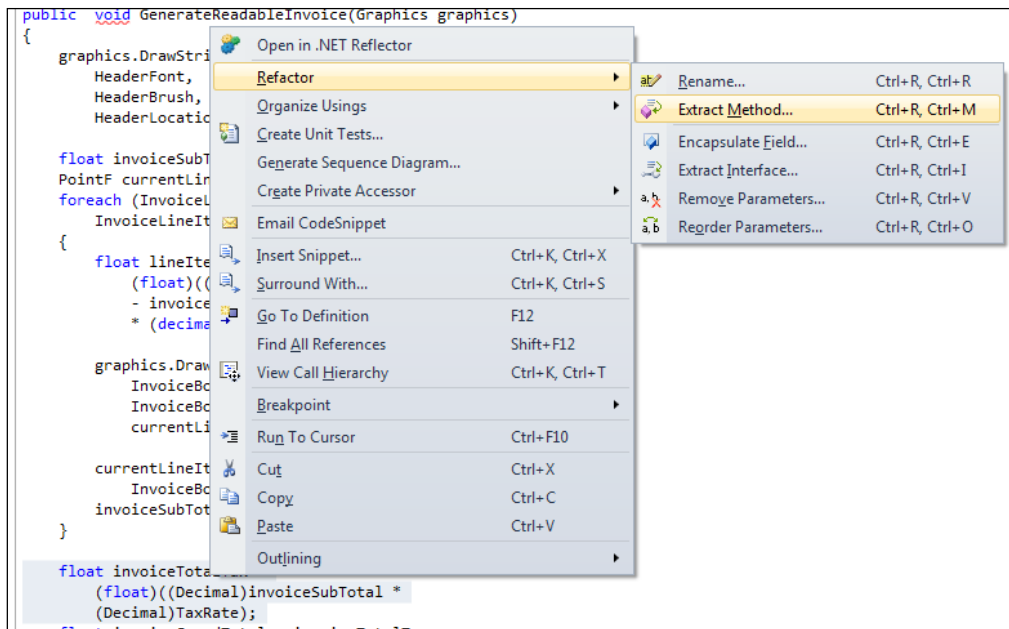
        currentItemLocation.Y +=
            InvoiceBodyFont.GetHeight(graphics);
        invoiceSubTotal += lineItemSubTotal;
    }
}
```

```
    }  
    float invoiceTotalTax =  
        (float)((Decimal)invoiceSubTotal *  
            (Decimal)TaxRate);  
    float invoiceGrandTotal = invoiceTotalTax +  
        invoiceSubTotal;  
    graphics.DrawString(String.Format("Invoice SubTotal: {0}",  
        invoiceGrandTotal - invoiceTotalTax),  
        InvoiceBodyFont, InvoiceBodyBrush,  
        InvoiceSubTotalLocation);  
    graphics.DrawString(String.Format("Total Tax: {0}",  
        invoiceTotalTax), InvoiceBodyFont,  
        InvoiceBodyBrush, InvoiceTaxLocation);  
    graphics.DrawString(  
        String.Format("Invoice Grand Total: {0}",  
            invoiceGrandTotal), InvoiceBodyFont,  
            InvoiceBodyBrush, InvoiceGrandTotalLocation);  
    graphics.DrawString(FooterText,  
        FooterFont,  
        FooterBrush,  
        FooterLocation);  
}
```

`GenerateReadableInvoice` draws a very simple invoice to a particular `Graphics` object. It draws a header, subtotals each line item, draws each line item with subtotal, calculates and draws the invoice total, calculates and draws total tax and draws a footer.

This method is currently functional. But, we can't perform any automatic validation or verifications of the calculations because they are intermixed with logic to draw to a `Graphics` object. The `Graphics` class is sealed, so we couldn't even implement a spy class and pass an instance to the method and spy on what is sent to `DrawString`.

If we want to automatically validate and verify the calculations performed when we generate an interface, we'll have to change the design of some implementation details. First, we need to decouple the calculations from the drawing logic. The easiest way to separate the calculation logic from the drawing logic is to use the Extract Method refactoring to extract a few methods. We can select the code that calculates a line item subtotal, right-click, and choose **Refactor\Extract Method...**



Enter the name of the method `CalculateInvoiceTotalTax`. We can then do the same with the code that calculates the invoice total tax and the code that calculates the invoice grand total. We'd then have something like this:

```
public void GenerateReadableInvoice(Graphics graphics)
{
    graphics.DrawString(HeaderText,
        HeaderFont,
        HeaderBrush,
        HeaderLocation);

    float invoiceSubTotal = 0;
    PointF currentItemLocation = LineItemLocation;
    foreach (InvoiceLineItem invoiceLineItem in InvoiceLineItems)
    {
        float lineItemSubTotal =
            CalculateLineItemSubTotal(invoiceLineItem);

        graphics.DrawString(invoiceLineItem.Description,
            InvoiceBodyFont,
            InvoiceBodyBrush,
            currentItemLocation);

        currentItemLocation.Y +=
            InvoiceBodyFont.GetHeight(graphics);
        invoiceSubTotal += lineItemSubTotal;
    }
}
```



```
    }
    float invoiceTotalTax =
        CalculateInvoiceTotalTax(invoiceSubTotal);
    float invoiceGrandTotal =
        CalculateInvoiceGrandTotal(invoiceSubTotal,
            invoiceTotalTax);
    graphics.DrawString(String.Format("Invoice SubTotal: {0}",
        invoiceGrandTotal - invoiceTotalTax),
        InvoiceBodyFont, InvoiceBodyBrush,
        InvoiceSubTotalLocation);
    graphics.DrawString(String.Format("Total Tax: {0}",
        invoiceTotalTax), InvoiceBodyFont,
        InvoiceBodyBrush, InvoiceTaxLocation);
    graphics.DrawString(
        String.Format("Invoice Grand Total: {0}",
            invoiceGrandTotal), InvoiceBodyFont,
            InvoiceBodyBrush, InvoiceGrandTotalLocation);
    graphics.DrawString(FooterText,
        FooterFont,
        FooterBrush,
        FooterLocation);
}

private static float CalculateInvoiceGrandTotal(
    float invoiceSubTotal, float invoiceTotalTax)
{
    float invoiceGrandTotal = invoiceTotalTax +
        invoiceSubTotal;
    return invoiceGrandTotal;
}

private float CalculateInvoiceTotalTax(float invoiceSubTotal)
{
    float invoiceTotalTax =
        (float)((Decimal)invoiceSubTotal *
            (Decimal)TaxRate);
    return invoiceTotalTax;
}

private static float CalculateLineItemSubTotal(
    InvoiceLineItem invoiceLineItem)
{
    float lineItemSubTotal =
        (float)((decimal)(invoiceLineItem.Price
```

```

    - invoiceLineItem.Discount)
    * (decimal)invoiceLineItem.Quantity);
    return lineItemSubTotal;
}

```

This is functionally equivalent to what we had before, but now the logic to perform the calculations has been separated from the logic that performs the drawing of the invoice.

This is a form of **Separation of Concerns**. We're dealing with two concerns here: one is the drawing of the invoice, and the other is the calculations. Separation of Concerns (or SoC) involves designing a physical separation between the concerns. In our example, our physical separation is separation by method. This separation could easily have been by class.

Before we can actually test these new methods, we first need to change their access modifiers to `public` from the `private` default that the extract method feature generated. We can now create an automated test for these new methods. If you prefer using Visual Studio® Unit Test, you can right click a method like `CalculateInvoiceGrandTotal` and choose **Create Unit Tests...** In the Create Unit Test form, you can accept the default selection of the `CalculateInvoiceGrandTotal`, or also select the other new methods. Go ahead and also check off `CalculateInvoiceTotalTax` and `CalculateLineItemSubTotal` and press **OK**. Visual Studio® 2010 should generate a new test class and add some test methods that look something like the following:

```

/// <summary>
///A test for CalculateInvoiceGrandTotal
///</summary>
[TestMethod()]
public void CalculateInvoiceGrandTotalTest()
{
    float invoiceSubTotal = 0F; // TODO: Initialize to an
    // appropriate value
    float invoiceTotalTax = 0F; // TODO: Initialize to an
    // appropriate value
    float expected = 0F; // TODO: Initialize to an appropriate
    // value
    float actual;
    actual =
        Invoice.CalculateInvoiceGrandTotal(invoiceSubTotal,
            invoiceTotalTax);
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive(
        "Verify the correctness of this test method.");
}

```

```
    }

    /// <summary>
    ///A test for CalculateInvoiceTotalTax
    ///</summary>
    [TestMethod()]
    public void CalculateInvoiceTotalTaxTest()
    {
        IEnumerable<InvoiceLineItem> invoiceLineItems = null;
        // TODO: Initialize to an appropriate value
        Invoice2 target = new Invoice(invoiceLineItems); // TODO:
        // Initialize to an appropriate value
        float invoiceSubTotal = 0F; // TODO: Initialize to an
        // appropriate value
        float expected = 0F; // TODO: Initialize to an appropriate
        // value
        float actual;
        actual = target.CalculateInvoiceTotalTax(invoiceSubTotal);
        Assert.AreEqual(expected, actual);
        Assert.Inconclusive(
            "Verify the correctness of this test method.");
    }

    /// <summary>
    ///A test for CalculateLineItemSubTotal
    ///</summary>
    [TestMethod()]
    public void CalculateLineItemSubTotalTest()
    {
        InvoiceLineItem invoiceLineItem = null; // TODO: Initialize
        // to an appropriate value
        float expected = 0F; // TODO: Initialize to an appropriate
        value
        float actual;
        actual =
            Invoice.CalculateLineItemSubTotal(invoiceLineItem);
        Assert.AreEqual(expected, actual);
        Assert.Inconclusive(
            "Verify the correctness of this test method.");
    }
}
```

We now have template unit tests that, by default, signal that the tests are inconclusive. Since we haven't provided the inputs and the expected results, these tests have no way of validating and verifying that our methods work. We now need to create any dependant objects, set the input values and the expected values. Our fictitious tax rate is an arbitrary 5%, so in order to finalize our tests we'd end up with code like this:

```
/// <summary>
///A test for CalculateInvoiceGrandTotal
///</summary>
[TestMethod()]
public void CalculateInvoiceGrandTotalTest ()
{
    float invoiceSubTotal = 123F;
    float invoiceTotalTax = 6.15F;
    float expected = 129.15F;
    float actual =
        Invoice.CalculateInvoiceGrandTotal (invoiceSubTotal,
            invoiceTotalTax);
    Assert.AreEqual (expected, actual);
}

/// <summary>
///A test for CalculateInvoiceTotalTax
///</summary>
[TestMethod()]
public void CalculateInvoiceTotalTaxTest ()
{
    IEnumerable<InvoiceLineItem> invoiceLineItems =
        new List<InvoiceLineItem> {
            new InvoiceLineItem() {
                Price = 123, Discount = .10F, Quantity = 2
            }
        };
    Invoice target = new Invoice2 (invoiceLineItems) {
        TaxRate = .05F
    };
    float invoiceSubTotal = 221.4F;
    float expected = 11.07F;
    float actual =
        target.CalculateInvoiceTotalTax (invoiceSubTotal);
    Assert.AreEqual (expected, actual);
}

/// <summary>
```

```
///A test for CalculateLineItemSubTotal
///</summary>
[TestMethod()]
public void CalculateLineItemSubTotalTest()
{
    InvoiceLineItem invoiceLineItem = new InvoiceLineItem() {
        Price = 123, Quantity = 2, Discount = .05F
    };

    float expected = 245.9F;
    float actual =
        Invoice.CalculateLineItemSubTotal(invoiceLineItem);
    Assert.AreEqual(expected, actual);
}
```

Now, as the `Invoice` class evolves over time, we know that any adverse side-effects to `CalculateLineItemSubTotal`, `CalculateInvoiceTotalTax`, and `CalculateInvoiceGrandTotal` will be detected as soon as possible. This new design means the `Invoice` class is more maintainable because it is more modular and more object-oriented by clearly delineating behaviour in methods and any new adverse side-effects will be pointed out as quickly as possible.

We effectively took a single method and refactored it into four methods. We essentially performed a refactoring in response to a Large Method code smell. Dealing with Large Method, as we've seen here, makes code much more maintainable by modularizing logic, through more methods. We can test the logic contained in those methods independently and more easily reuse that logic.

The ease with which someone can add a bug into the code base can depend on the design of the API at their disposal. An API that is hard to understand or hard to remember is an API that is easy to make a mistake with. An API should be intuitive. That is, the developer shouldn't require a lot of conscious thought to program with the API—especially conscious thought about the API as it applies to the domain in which it services.

The `System.Drawing` namespace contains examples of some intuitive APIs. For example, the `Graphics.DrawLine` effectively has two overloads. One that deals with two-dimensional point values and one that deals with numeric coordinates. `DrawLine` actually has four, but the two that deal with points have parameters in the same, consistent order (one takes `Point` object arguments, the other `PointF` object arguments) and the two that deal with numeric coordinates have parameters in the same, consistent order (`int` and `float` values). This means, regardless of whether I'm using `PointF` or `Point` objects, I would *call* `DrawLine` the same way:

```
graphics.DrawLine(Pens.Black, topLeftPoint, bottomRightPoint);
graphics.DrawLine(Pens.Black, firstPoint, secondPoint);
```

And regardless of whether I'm using `int` or `float` values, again I'd call `DrawLine` the same way:

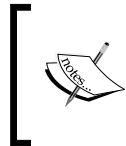
```
graphics.DrawLine(Pens.Black, leftCoordinate,
    topCoordinate, rightCoordinate, bottomCoordinate);
graphics.DrawLine(Pens.Black, firstX, firstY,
    secondX, secondY);
```

Where the arguments are defined as follows:

```
Point topLeftPoint, bottomRightPoint;
PointF firstPoint, secondPoint;
int leftCoordinate, topCoordinate;
int rightCoordinate, bottomCoordinate;
float firstX, firstY, secondX, secondY;
```

The programmer doesn't have to think about how to call the method depending on their arguments: it's intuitive.

An API that requires a developer to read lots of documentation before they can attempt to use the API is an API that is difficult to use. An API that is difficult to use is prone to error. Usually these are silly errors that could have easily been avoided if the API didn't let them do what they did. An important design guideline to help make an API easier and less prone to error is to practice **Intention Revealing Naming**.



Intention Revealing Naming involves using names that reveal what the variable contains and how it is used, or method names and signatures that reveal what the method does and how it should be used.

When names are intention revealing, they are self-documenting. Most modern Integrated Development Environments include some sort of statement completion—a feature that reveals available identifiers based on the context of what is being typed. This allows programmers to let the editor tell them what is available to them while they type. If all the identifiers that are presented to the programmer reveal nothing about what they are or what they do, the collection of revealed identifiers is useless to them. For example, the following class and method don't reveal much about what they do or how they could be used:

```
public class Customers
{
    public IEnumerable<Customer> Get(String name)
    {
        //...
    }
}
```

Sure, `Get` is a valid name. But, presented with "Get" from IntelliSense after typing "customer" doesn't offer much in the way of information for the programmer. With regard to the name parameter given to `Get`, what if a customer of that name doesn't exist? What if there are more than one customer that have that name? The name of the class in conjunction with the name of the method doesn't reveal what it might do. The following is much more intention revealing:

```
public class CustomerRepository
{
    public IEnumerable<Customer> FindCustomersByLastName(
        String name)
    {
        //...
    }
}
```

First, we've given our class name a more descriptive name. What we're operating on is a repository of customers—it contains some number of customers. Next, we've added some detail to our method name, not only revealing more about how the method is used but also more about what it returns. There's a common idiom where using a "Find" prefix when your method may return zero or more results instead of "Get" prefix (which would always return one result or an exception if it could not). This naming now leaves us free to extend our API in a consistent and easy-to-understand way with methods such as `FindCustomersByLastName` (which would be really hard to do or be consistent before with "Get"). We could also add related methods like `FindCustomerByCustomerNumber` that would be very consistent with the other methods and thus easier to consume. The use of plural and singular words when dealing with results that specifically return zero, one or more, or zero or one, is subtle but it makes the methods much more intention revealing.

Fortunately, it's easy to evolve our code base into something more intention revealing. As we saw in Chapter 1, the fact that a particular class or a particular method of a class may be used dozens of times, means we can get Visual Studio® 2010 to change every single instance of a class name, and every single reference to a method within the entire solution. What once used to be tedious and prone to error (although the compiler tells you right away when you've missed something) is now a breeze with built-in refactoring to rename a class and a class member.

Feature Envy code smell

When one class relies on another class more than itself, it has a code smell called **Feature Envy**. A class with feature envy of another class means it is highly coupled to that class. Any change to the class being envied means it affects the class with the envy.

Feature Envy can be easy to detect in some circumstances, and hard in others. A class that is only used by one other class and the other class simply delegates to that class for most of its functionality has Feature Envy. This is easy to detect. A base class that has only one subclass and the subclass modifies or extends very little behaviour of the base is generally Feature Envy. This is harder to detect. Feature Envy is subjective. If there is value to having another class, despite that class relying on another class for most of its functionality, and that other class isn't used by other classes, it might still be beneficial.

Feature Envy affects maintainability because logic spread amongst more than one class has an explicit delineation of that logic. We respect that published interface and, as we should, push back against change to that interface without good reason. When you're looking at two classes, we can often not see the forest. We shouldn't see the forest at that level. We can only assume that published interface is used outside the context in which we're looking. With this restriction, it's not as easy to make arbitrary changes to these envied classes.

Essentially what should be an implementation detail has been made public, to be made dependant on by other classes. It's best to find and refactor Feature Envy as quickly as possible before it has been made arbitrarily dependent upon. There's nothing wrong with being depended upon, but it needs to be planned.

So, what can we do about Feature Envy? Visual Studio® 2010 doesn't have anything specific in its refactoring abilities to deal with Feature Envy directly. If the class being envied is autonomous – not the base class to the class with envy – then the easiest thing to make the class an implementation detail is to make it a private nested class of the envying class. That doesn't necessarily solve the problem, but it means no unplanned dependencies can be made on this class. This can be done by moving the envied class within the scope of the envying class and changing its access to `private`.

If the class being envied is actually the base class of the envying class then we can use the **Push Down** refactoring. We haven't discussed the Push Down refactoring, and it's not a built-in refactoring that Visual Studio® 2010 implements. Push Down refactoring moves a member from a super class to a subclass. Let's have a look at the Push Down refactoring. Let's say we have the following classes:

```
public abstract class Shape
{
```



```
    public float Width { get; set; }
    public float Height { get; set; }
    public float Diameter { get; set; }
}

public class Circle : Shape
{
    public float Radius { get { return Diameter/2; } }
}
```

The `Shape` class is an abstraction that encapsulates something that acts like a shape. For our purposes a shape has width, height, and diameter. Upon reflection, for whatever reason (besides the obvious) we've decided that `Diameter` shouldn't belong to `Shape` and should belong to `Circle`. We can implement this change with the Push Down refactoring. We can move the `Diameter` property from the super class to the sub class. This can be performed by drag-drop or cut/paste. This results in the following:

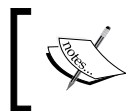
```
public abstract class Shape
{
    public float Width { get; set; }
    public float Height { get; set; }
}

public class Circle : Shape
{
    public float Radius { get { return Diameter/2; } }
    public float Diameter { get; set; }
}
```

The `Diameter` property is now no longer a property of the super class `Shape`, but a property of the sub class `Circle`. We're now saying that the `Shape` abstraction doesn't need to encapsulate `Diameter` and that it is a property of a `Circle`.

The problem with refactorings like Push Down that aren't directly supported by Visual Studio® 2010 is that there is no automated refactoring of references. In the scope of the `Shape` and `Circle` classes that we've presented here, we only need to cut the `Diameter` property from `Shape` class and paste it into the `Circle` class. In the real world, `Shape` and `Circle` are used by the code base and `Circle.Diameter` is referenced by other code. To complete our refactoring we'd likely have to change all references to `Shape.Diameter`. If we're truly refactoring and `Circle.Diameter` isn't used then it is **Dead Code** and should simply be deleted. If `Circle.Diameter` is referenced, the easiest way to complete the refactoring is to use the **Find All References** feature of Visual Studio® and get a list of all references to `Circle.Diameter` before removing and moving it to `Shape.Diameter`. Then, go through each reference in the Find All References list changing each one to reference `Shape.Diameter`.

Design for the sake of reuse



Object-oriented Myopia is when the process or the principle of objectifying concepts becomes more important than the value it adds to the code base.

It's sometimes easy to get into what I call **Object-oriented Myopia** and either try to force concepts into specific classes or to design classes that ignore the domain and let another context influence the design of our classes. Depending on the context, shapes may be a perfect example. In the "real world" we have polygons, squares, rectangles, circles, ellipses, parallelograms, trapezoids, rhomboids, and so on. As designers, we can go overboard objectifying concepts we know but are never needed in the domain we're designing for – we tend to project a framework onto our domain model. We sometimes think that our domain model needs to support more than our domain. This is rarely due to bad intentions; possibly driven by the **Fallacy of Reuse**.

Udi Dahan coined the term *Fallacy of Reuse* – a form of object-oriented myopia. This is when making something reusable becomes more important than fulfilling requirements and implementing a model that echoes the problem domain. Essentially, reuse needs to be designed; it needs to happen on purpose. Reuse by accident occurs when you try to make something reusable but it can't be reused right away. When it does get reused, it's often by accident; it can't be used by design because it was based on fictional usage. A design might allow something to be reused; but to design it to be reused in a way it needs to be reused is better.



Udi Dahan, *The Fallacy of Reuse*, <http://www.udidahan.com/2009/06/07/the-fallacy-of-reuse/>

Don't repeat yourself

Don't Repeat Yourself or **DRY** is about not doing the same thing twice. Reuse is great, but reuse for the sake of reuse defeats the purpose. If something doesn't have to be reused, don't make it reusable. Don't make a framework where a framework is not needed, don't make a class hierarchy where none is needed, don't try to anticipate solutions to fictional requirements, and don't do work that doesn't add value.

Inappropriate Intimacy code smell

In Object-Oriented Design, encapsulation is the practice of hiding implementation details. A well encapsulated class doesn't make its implementation details public. This is sometimes also referred to as Data Hiding or Information Hiding. I prefer not to use these terms because they tend to allow designers to focus only on hiding data (which is often defined differently by different people) and not all implementation details. Let's look at an example of **Inappropriate Intimacy**:

```
public class Customer
{
    private List<PhoneNumber> phoneNumbers = new
        List<PhoneNumber>();
    public List<PhoneNumber> PhoneNumbers
    {
        get
        {
            return phoneNumbers;
        }
    }
    //...
}
```

This is a simple class that intends to implement a customer abstraction. In this particular case, our customer only has zero or more phone numbers. We've implemented a collection of phone number objects with the `List<T>` collection type. We've then provided a `PhoneNumbers` property that provides read-only access to this collection (the collection contents can be modified, but the collection can't be replaced).

This seems innocuous enough, but what we've done is to inadvertently make public our implementation details. The type of the `PhoneNumbers` property is `List<PhoneNumber>`, the same as our private field. This is a contract with other classes that use this class that they'll receive a `List<PhoneNumber>` object when they access the `PhoneNumbers` property. This forces the `Customer` class to create an object of type `List<PhoneNumber>` to fulfil this request. If the `Customer` class later decides that `List<T>` is not appropriate for what it needs to do, it's out of luck. You may think this is not that big a deal, we could do something like the following:

```
public class Customer
{
    private Stack<PhoneNumber> phoneNumbers =
        new Stack<PhoneNumber>();
    public List<PhoneNumber> PhoneNumbers
    {
        get
        {
            return new List<PhoneNumber>(phoneNumbers);
        }
    }
    //...
}
```

This uses the collection type `Stack<T>` instead of `List<T>`, and the `PhoneNumbers` property simply creates a new `List<PhoneNumber>` object, populating it with the content of the `Stack<T>` object – all the while seemingly retaining our contract. The problem with this is that it breaks the contract; it doesn't allow other classes to add or remove `PhoneNumber` objects from our `Customer` class. While this particular implementation may be good, it's different from the original contract.

This obviously causes us maintenance problems because it limits what we can do with our class. We're forced to use a certain concrete type. Unfortunately, the only way to fix this is to break the contract and refactor the return type to something abstract. For example:

```
public class Customer
{
    private List<PhoneNumber> phoneNumbers =
        new List<PhoneNumber>();
    public ICollection<PhoneNumber> PhoneNumbers
    {
        get
        {
            return phoneNumbers;
        }
    }
}
```

```
    }  
    //...  
}
```

This implementation allows similar functionality (`Add`, `RemoveAt`, and so on) while encapsulating our implementation details. The class is now free to implement a collection of `PhoneNumber` objects in different ways, like `Dictionary<T>` or even a class that encapsulates a collection that doesn't keep all objects in memory and pages objects to a storage device.

Unfortunately Visual Studio® 2010 doesn't include a built-in automated refactoring for this, so we must perform the refactoring by manually changing the signature of the property, then changing all references to the property that use the `List<PhoneNumber>` type directly. Changing all the references can be done by using the Find All References feature to get a list of references that we can simply double-click and change the reference.

This is a rather simple example of Inappropriate Intimacy. And, in fact, there is a Code Analysis rule that warns against this. Code Analysis will warn you that you're exposing a concrete type that derives from an interface and suggests that you expose the interface instead.

When we implement a class it's almost impossible to avoid exposing implementation decisions to some extent. For example, a method that returns an `Int32` exposes the fact that it chose 32-bit integer to implement at least part of that method. With low-level base types like `Int32`, `Int64`, `String`, `Single`, `Double`, or `Decimal` this isn't a problem; and in fact it's usually a good idea—these base types have built-in support by the CLR and we simply couldn't write something better. But, there are all sorts of decisions we can make to implement a particular class or methods of that class that aren't necessarily useful to expose. Exposing implementation details will still obviously allow you to implement working code that others can use. But, as we've discussed, exposing those implementation details like that couples the implementation details to external code. This limits the maintainability of your class because of your restricted options.

This isn't limited to framework or base-class library types. We could, very easily, create a class to implement a specific algorithm, structure, protocol, concept, and so on. If the implementation detail isn't relevant to client code to a class that uses that implementation detail, it shouldn't be made public like that.

Lazy Class code smell

The **Lazy Class** smell involves classes that do very little, or nothing useful at all. There are some basic design patterns that specifically implement classes that themselves do very little, or nothing at all. But, their value is recognizability. Patterns like the Null Object pattern specifically implement classes that do nothing in order to act as an easy to recognize or read placeholder.

Outside of these circumstances, classes that do nothing or little at all and don't act as some sort of place holder really don't add much value to the source code. They hinder maintainability because their interface imposes a contract on those that use it and is public for anything to become coupled to.

A Lazy Class that isn't used is easy to deal with. Simply consider it dead code and remove it. Lazy Classes that are used are a little more problematic. A Lazy Class used by only one other class is only slightly less easy to deal with. These instances of Lazy Classes can simply be subsumed by the class using them. In a case like this, the class using the Lazy Class is delegating logic to another class when it should be part of its private implementation details.

A Lazy Class that is a sub class of a super class can be dealt with simply by using the **Collapse Hierarchy** refactoring – moving the logic from the sub class into the super class and simply removing the sub class. There's no built-in automated refactoring for this, so this process must be done manually in Visual Studio® 2010.

Lazy Classes that are used by more than one other class present a bit of a problem. One way of dealing with it is simply the same as a single dependant class: just subsume the logic being used into the other classes. This, of course, will work; but we risk violating Don't Repeat Yourself. If that is part of a phased approach to refactoring away the Lazy Classes, this may be acceptable. The repetition becomes an implementation detail of several classes and can be dealt with without affecting other classes by evolving it over time. This can often be done independently of one another.

If you've recognized the Lazy Class and noticed that it is being used by several other classes, you will likely recognize how to refactor away the Lazy Class in one phase. After all, we're dealing with a class that does very little.

Let's look at an example of a Lazy Class and the process of refactoring it:

```
public class PastDueInvoice : Invoice
{
    DateTime PastDueDate
    {
        get; set;
    }
}
```

The `PastDueInvoice` is an `Invoice` that is past due. It may seem logical for this to be its own class, but it doesn't do enough to warrant the added maintainability of another class. In this case, simply collapsing the hierarchy will be enough to complete this refactoring. In order to do this, the `PastDueDate` is moved to `Invoice`, and all uses of `PastDueInvoice` are changed to `Invoice`.

Lazy classes are often a side-effect of Object-oriented Myopia. Through transference, in our haste to implement a *perfect* object model we introduce scope creep. We introduce fictitious requirements that make sense in the context we're in but are unnecessary. This could be the introduction of a hierarchy that doesn't add value or simply separating logic into another class but contains very little logic. This needless complexity introduces explicit interfaces and contracts that need to be maintained over time and reduce the flexibility of the code base to change, reducing maintainability.

Improved object-model usability

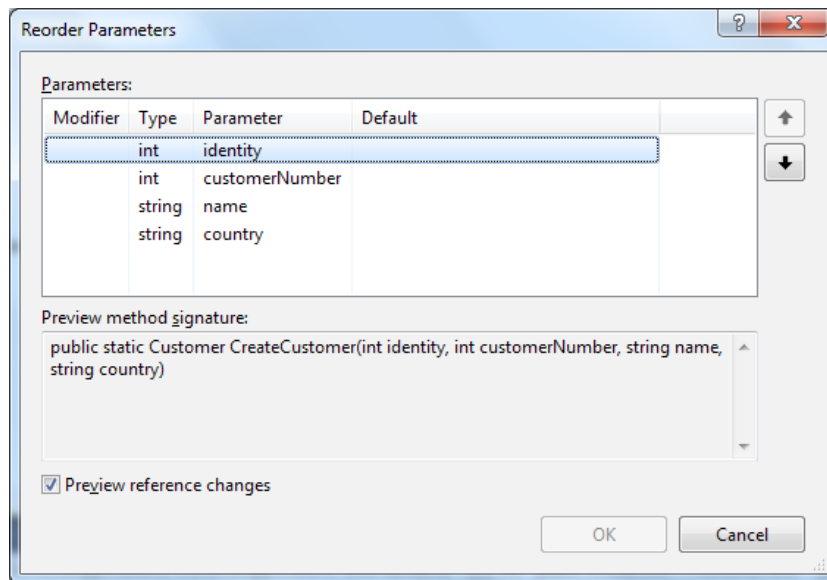
Designing an object-model is hard. First and foremost an object-model needs to fulfill its functional requirements. If we can't do what we need to do with it, it doesn't offer any value. Second, it needs to be robust and reliable. If it does do what we need to do with it but can't rely on it or need to spend extraneous amounts of work to work around problems, its value can be limited.

Even if we create a functional object model that is robust and reliable, it may be difficult to use or easy to use it in incorrect ways making the code that uses it not robust and unreliable.

One way this can happen is the choice of parameters and their order. It's easy to create a method that does something useful but requires many parameters. A method with many parameters is easy to call incorrectly while still being syntactically correct. If some adjacent parameters are the same type, it's easy to transpose arguments and end up calling the method incorrectly without any compilation error.

Simply changing the order of parameters in such cases can make use of the method less error prone and promote the appearance of better reliability and robustness.

Luckily Visual Studio® 2010 has a built-in automated refactoring to reorder parameters. To reorder parameters of a method, simply right-click the method name in the editor and choose **Refactor/Reorder Parameters...** You will be presented with **the Reorder Parameters** form, such as the following:



In our example, we'd like to make our `CreateCustomer` less likely to be used incorrectly, so we'd like to change the signature so that `identity` and `customerNumber` are not next to each other, and `name` and `country` are not next to each other.

To do this, click on `customerNumber` and click the down arrow. This changes the signature so that the order of the parameters are `identity`, `name`, `customerNumber`, and `country`. Now press **OK**. The **Preview Changes** form displays each of the reference changes that will be made. Pressing **Apply** will apply the changes and reorder the arguments passed to all calls to the `CreateCustomer` method. This is much easier and more reliable than manually changing each reference.

Now that we've changed the parameter, if we accidentally transposed two arguments we'd end up with a compile error. Previously if we transposed two arguments like this:

```
Customer.CreateCustomer(1, 100, "Canada", "Peter Ritchie");
```

then we wouldn't be able to detect the problem until the code was executed and tested. Now that we've changed the order of the parameters, if we transposed two arguments like this:

```
Customer.CreateCustomer(1, "Peter Ritchie", "Canada", 100);
```

a compiler error would be output when we try to compile it. This makes it much harder for a bug to be introduced.

Contrived Complexity

Erich Gamma et al introduced many software designers to a concept called patterns with the book *Design Patterns*. Patterns have been around in other disciplines for quite some time. A pattern is basically a recipe; it systematically describes a problem, its solution, when to apply the solution, and the consequences.

Design patterns took off with many designers. After reading *Design Patterns*, many could see many places to implement patterns like Strategy, Observer, Adapter, Bridge, Visitor, and so on. Sometimes use of these patterns was applicable, but sometimes it was not. Designers forced various patterns into places where they didn't really apply, or in places that didn't really add value. Their zeal to use patterns wherever they could increased the complexity of their code base with little added value.

This is the Contrived Complexity code smell – where extra complexity is introduced for the sake of a design pattern or even for the sake of complexity. When extraneous complexity is introduced, it makes the code base harder to understand and harder to maintain. With design patterns, if they're forced into situations where they aren't really meant to be in, it defeats the purpose of the design pattern.

A design pattern is a means to communicate a concept. Within code, it's a form of documentation. When the code implements an Adapter, for example, the reader of the code can understand what it is doing and what it is intending to do. But, when the pattern is forced into places it doesn't belong, it makes it harder to understand what is going on. Implementing the Strategy pattern, for example, where there's only one possible policy that can be used adds complexity where it doesn't need to be.

The Strategy pattern is a useful pattern that separates policy from the code that uses it to introduce the flexibility of using different policies depending on run-time conditions. But, it's fairly complex to implement. Generally, implementing the Strategy pattern means introducing an interface that abstracts the policies from the code that uses it – a means by which to inject a policy into the code that needs it and the classes that implement the differing policies. When you're learning the Strategy pattern, it's easy to get caught up in the pattern and in your zeal to get experience see any logically independent logic as a candidate for Strategy.

Once Strategy is introduced, there needs to be a means to manage the strategy implementations and delivering them to the code that needs them.

Let's have a quick look at an example of overzealous use of Strategy. If we revisit the Invoice class, a designer eager to get experience with Strategy might see an opportunity to use Strategy with the calculation of an invoice's subtotal.

```
public interface IInvoiceGrandTotalStrategy
{
```

```
float CalculateGrandTotal(float invoiceSubtotal,
    float invoiceTotalTax);
}

public class InvoiceGrandTotalStrategy :
    IInvoiceGrandTotalStrategy
{
    public float CalculateGrandTotal(float invoiceSubTotal,
        float invoiceTotalTax)
    {
        float invoiceGrandTotal = invoiceTotalTax +
            invoiceSubTotal;
        return invoiceGrandTotal;
    }
}

public class Invoice
{
    private IInvoiceGrandTotalStrategy
        invoiceGrandTotalStrategy;
    public Invoice(
        IEnumerable<InvoiceLineItem> invoiceLineItems,
        IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy)
    {
        InvoiceLineItems =
            new List<InvoiceLineItem>(invoiceLineItems);
        this.invoiceGrandTotalStrategy =
            invoiceGrandTotalStrategy;
    }
    //...
}
```

Then, in order to calculate the invoice grand total, there would be code in the Invoice class like this:

```
float invoiceGrandTotal =
    invoiceGrandTotalStrategy.CalculateGrandTotal(
        invoiceSubTotal,
        invoiceTotalTax);
```

In order to implement the strategy pattern we've introduced `IInvoiceGrandTotalStrategy`, an implementation of `IInvoiceGrandTotalStrategy` `InvoiceGrandTotalStrategy`, an added field in `Invoice` `invoiceGrandTotalStrategy`, and a changed constructor to inject the strategy into the invoice object (since it's required in order to properly calculate the invoice). This is approximately 15 extra lines of code, drastically increasing the complexity and maintainability of the code. Now, someone not only needs to understand that Strategy is being used here; but also understand why. The *why* isn't apparent, and the reader will likely spend extra time trying to figure out the *why* – wasting time.

This particular Contrived Complexity is fairly easy to refactor – the method in `InvoiceGrandTotalStrategy` can be moved to the `Invoice` class, the `invoiceGrandTotalStrategy` field and its assignment removed, and the `invoiceGrandTotalStrategy` removed (via Remove Parameters automated refactoring).

Strategy is a fairly easy to understand pattern (which may explain its frequent misuse). Its misuse is fairly easy to detect. Strategy is meant to introduce the ability for different policies to be used by code, depending on some condition – usually at runtime. If this implementation only implements one policy (`InvoiceGrandTotalStrategy` in our example), we know that Strategy isn't being used properly.

Some readers may think that the fact that that a particular implementation only implements one policy introduces extensibility. It does introduce extensibility, and if you need that extensibility then this implementation is a good thing. But, if this extensibility isn't immediately usable then it's not a good thing. If you're not immediately using it, you don't know if how it was designed or implemented is correct and if you don't have a requirement to extend it, you don't really have a way to verify the design or implementation. This is an example of YAGNI.

So, we've seen some examples of refactoring object-oriented myopia, but what about refactoring procedural code to be more object-oriented. Object oriented design is effectively about modularity through designing objects and implementing their data and behaviour through the use of abstraction, encapsulation, and polymorphism. Polymorphism can be realized in many ways: subtype polymorphism (inheritance), parametric polymorphism (Generics), and ad-hoc polymorphism (member overloading). Through correct use of design of a domain model, we realize encapsulation and abstraction.

Not fully utilizing polymorphism can sometimes lead to implementing procedural code, code that isn't truly object-oriented. Polymorphism gives us the ability to create an object that performs logic for the situations to which it applies. This means the choice of which type to instantiate should be the only decision that needs to be made. Use of conditional statements is generally indicative of a procedural design. The greater the number of conditionals in a particular block of code means an increased number of potential paths through the code. These increased paths increase the surface area for code coverage and testing of those paths within the methods they are contained within. More conditionals mean more tests that need to be written and performed in order to thoroughly test the code – making it harder to maintain. The choices being made by the conditionals should be implemented through polymorphism. The ultimate misuses of this are type-based conditionals – comparisons based on the type of an object.

Shapes are a canonical example of procedure code where it should be object-oriented. For example, let's say we have a service that draws shapes onto some sort of canvas like `Graphics`. If it were procedural, it may look something like this:

```
public class ShapeDrawingService
{
    public void Draw(Graphics graphics, Shape shape,
        Point location)
    {
        Circle circle = shape as Circle;
        if(circle != null)
        {
            RectangleF rectangle =
                new RectangleF(location,
                    new SizeF(circle.Radius * 2,
                        circle.Radius * 2));
            graphics.DrawEllipse(Pens.Black, rectangle);
            return;
        }
        Square square = shape as Square;
        if(square != null)
        {
            graphics.DrawRectangle(Pens.Black, location.X,
                location.Y, square.Width, square.Width);
            return;
        }
        throw new ArgumentException("Unsupported Shape",
            "shape");
    }
}
```

This code works, but it's more complex than it needs to be. Dealing with all shapes is done in one method, and multiple comparisons are performed to find out what shape it is dealing with. Since some sort of decision was made in order to create the shape, and these comparisons are extraneous. As with most refactorings, this is fairly easy to refactor. We simply need to create two overloads, one that takes a `Circle` and one that takes a `Square` instead of `Shape`. We could do this entirely manually by creating a new overload, changing the signature of the existing method, moving code to the new overload, and then cleaning the existing method. But, we can use some automated refactorings in this particular circumstance. We can use the extract method with the body of one of the if statements to create a new method and when we give it the same name as the original method it will create an overload using the `Shape` subclass. If applicable, we may need to change the order of the parameters of the new method to match the original method. We can do that with the Reorder Parameters refactoring. We then need to remove the code in the original method that dealt with the `Shape` type that is now handled by the new overload (if we extracted a method from the `Circle` code, for example, that would have been replaced by a call to `Draw`; we simply need to delete everything that deals with `Circle`). We then need to change the type and name of the parameter of the original method to be a `Shape` subclass (`Square`, if we extracted the `Circle` code into a new method). Since the original method now takes a specific `Shape` subclass, we can now remove all code except the call to `Graphics` method. We'd then end up with code that looks like the following:

```
public class ShapeDrawingService
{
    public void Draw(Graphics graphics, Shape square,
        Point location)
    {
        graphics.DrawRectangle(Pens.Black, location.X,
            location.Y, square.Width, square.Width);
    }

    public void Draw(Graphics graphics, Circle circle, Point location )
    {
        RectangleF rectangle =
            new RectangleF(location,
                new.SizeF(circle.Radius * 2, circle.Radius * 2));
        graphics.DrawEllipse(Pens.Black, rectangle);
        return;
    }
}
```

What we have now is more object-oriented. We're making use of polymorphism implemented by overloads to avoid extraneous type conditionals. The added benefit of this implementation is that we now have compile error if we try to use a subclass of `Shape` that isn't `Circle` or `Square`. Previously, this was a run-time error and could only be detected if that scenario was actually executed (for example, with a specific unit-test).

Doing this particular refactoring can often orphan the super class. This may have been the only use of `Shape` (as a parameter in the original `ShapeDrawingService.Draw` method). After doing this type of refactoring, it's a good idea to re-evaluate whether you still need the super type.

Detecting maintainability issues

Visual Studio® 2010 includes a feature that calculates metrics based upon the code and the structure of the code. These metrics include:

- Maintainability Index
- Cyclomatic Complexity
- Depth of Inheritance
- Class Coupling
- Lines of Code

All of these metrics can be used, in some fashion, to gauge the maintainability of projects, namespaces, classes, and methods.

The Maintainability Index attempts to systematically rate the maintainability of a piece of code through weighting a modified Halstead volume, Cyclomatic Complexity, and Lines of Code. The Code Metrics features analyses compiled code (the Intermediate Language contained in the assembly) which doesn't include comments, so the Halstead volume used in the Maintainability Index doesn't take into account comments. The introduction of comments does not have a specific effect on the maintainability of the code for which it is commented, so this isn't necessarily a bad thing. A rating of 100 is the highest index code can receive. The index doesn't necessarily mean the code maintainability can be improved or not. For example, an index of 100 doesn't mean the code should not be evaluated for maintainability improvements. For example, our original `ShapeDrawingService.Draw` method had a respectable index of 61 and a green rating, yet it clearly needed improvement. The refactoring increased the index to green ratings of 80 and 88. This is an improvement obviously, but 61 by itself is clearly not enough to necessarily detect a maintainability issue.

Maintainability index values of between 10 and 19 have a yellow rating, and index values between 0 and 9 have a red rating. Red and yellow ratings are an indication of code that is in dire need of evaluation for maintainability. But, what if all your code is green? You can focus maintainability refactoring efforts not necessarily on the value of a particular index; but on the difference of two or more values.

When looking at two values with a mandate to improve maintainability, the lower index should be chosen first. With maintainability index, the higher the index the better. Both values may be green, but the lower one may be a better candidate or may pay better maintainability dividends when refactoring for maintainability. The index should be taken with a pinch of salt though, as depending on how you refactor the code, the original index may now be meaningless. For example, if you end up removing a class during refactoring, that class's original index is now moot. You can't measure the improvement because you don't have anything to compare it with. When evaluating metric values, it's not useful to mandate a goal of reaching specific values, despite being rated green, yellow, and red. For example, if they're all green that doesn't mean there's no room for improvement.

There are many ways to refactor code to improve Maintainability Index. As we've seen with our `ShapeDrawingService`, we can improve Maintainability Index by refactoring from procedural to more object-oriented. Lines of code are weighted in the index calculation, so, simply performing an Extract Method refactoring can improve the Maintainability Index. Refactoring specifically to improve Cyclomatic Complexity will also improve the Maintainability Index, which we'll look at next.

The Maintainability Index includes the **Cyclomatic Complexity (CC)**; so, improvements to Maintainability should also include improvements to CC. CC is essentially a measurement of the number of branches in code. CC is based on the number of conditionals within the code. We saw an improvement in our `ShapeDrawingService.Draw` overloads because we reorganized the code to avoid conditionals. The higher the CC value the higher the number of conditionals. A method with no conditionals will have a CC metric of 1. Clearly we can't have all our methods having a CC metric of 1, but it's better for the metric to be lower than higher. A higher CC could be due to procedural code that could be more object-oriented. At the very least, a large CC value means the code is too complex and should be dealt with. Dealing with the code could be as simple as extracting some methods, or could be as complex as introducing more classes. There's generally a direct relationship between CC and the return on investment from refactoring the code; but as with Maintainability Index, focusing on the difference in values rather than specific values leads to better gains. As with Maintainability Index, this applies to evaluating the specific values. Although you can't get a better value than 1, just because a value is close to 1 doesn't mean there's no room for improvement.

The Cyclomatic Complexity can only be improved by reducing conditionals. Using polymorphism isn't the only way to reduce conditionals. Extract Method refactoring can move some conditionals from one method to another; so you can get CC improvements within a particular method with Extract Method. But be careful; you are only moving the conditionals around within a class so the CC of the class won't improve.

The Class Coupling metric measures a particular class's coupling to other classes. A class that is coupled to many classes is harder to maintain because it's harder to move around without those other classes and thus harder to use in other contexts. High class coupling can be improved by consolidating multiple classes together, if that makes sense. Some classes will always have higher coupling than other classes simply because of their nature. Classes that are part of a framework should tend toward lower class coupling values, but the value is ultimately subjective. A particular class may be coupled to many other private classes in a particular framework. In this case, it's unlikely that the class is any less reusable than it could be if the other classes were consolidated within it. Don't ignore high class coupling values, but take the value with a pinch of salt and evaluate each candidate within its unique circumstances.

The only way to improve the Class Coupling metric is to use less unique classes. For the most part we do want unique and specific classes in our code base, so make sure your drive to improve Class Coupling isn't actually making your code less maintainable. You could implement your application in one huge method in one class and get a really low Class Coupling, but your other metrics would be really high. So, when you're refactoring to improve class coupling, keep an eye on Maintainability Index and Cyclomatic Complexity to make sure you're not making things worse.

The last metric is Lines of Code (sometimes referred to as LOC). I don't recommend refactoring specifically to lower Lines of Code. I recommend using it in conjunction with the other metrics. All other metrics being equal, your efforts should start with the code with significantly more Lines of Code. Two pieces of code with a Maintainability Index of 50, one with 100 lines of code, and the other with 10,000 lines of code; the code with 10,000 lines will have that many more opportunities for improvement.

Obviously, there's specific refactorings that reduce lines of code; but focus refactoring efforts on improving the other metrics instead and use Lines of Code to focus on which code to refactor, not how.

Summary

We've seen how we can improve code maintainability with Visual Studio®. Visual Studio® can be used to detect maintainability issues and refactor them through refactorings like Pull Down to make code more maintainable. We've also seen how we can make an API less prone to being used incorrectly by using the Reorders Parameters Refactoring. Automated unit testing supports the refactoring effort and maintains certain quality expectations by validating code before and after being refactored.

In the next chapter, we'll continue our maintainability focus and detail how we can use Visual Studio® to improve code navigation with object-oriented code through naming, structure with refactorings like Extract Class, Pull Up Method, and Move Method. We will also detail navigation with Visual Studio®.

4

Improving Code Navigation

Improving how code can be changed and how fast depends partly on the navigability of the code. Code navigability is the ability of code to be understood and maneuvered simply through reading it. If the code is hard to absorb and hard to navigate, it's going to reduce the speed at which changes can be made to it.

There are many ways code can be made more navigable. We've discussed some ways in previous chapters, such as restructuring in response to code smells like Large Method or Large Class. Code is more navigable if large chunks of it aren't in one method or class. By improving code structure, we can increase its understandability through the introduction of new classes and methods that make concepts explicit (and able to be checked), thus making code easier to absorb and maneuver through. But, there are other ways of making code more navigable without specifically addressing code smells. We'll discuss these in this chapter, including refactoring the following to improve navigability:

- Object-oriented code
- Naming
- Structure
- Accounting for navigators (Visual Studio®)

Navigating object-oriented code

Object-oriented code is not procedural code.

The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines, whereas in object-oriented programming it is to break down a programming task into objects with each "object" encapsulating its own data and methods...

– Wikipedia (http://en.wikipedia.org/wiki/Procedural_programming)

Procedural code is inherently linear, has a beginning and an end, specifies one step after another, and uses variables for application state. There's generally an entry point, a collection of un-encapsulated data, and a series of steps that modify the data before completion. Each step is generally dependant on the previous and it's easy to follow from one step to the next. Object-oriented code is similar, but rather than modifying global state and invoking many functions, object-oriented code decouples and modularizes the logic into classes that each manage their own data or state. Unlike procedural code, each class is essentially autonomous. Procedural code implements structure through functions, but functions are just reusable containers of code which translates input to an output or modifies global state.

Introduce inheritance or interface-driven design and the connections between classes become even more disjointed. It becomes increasingly harder to follow object-oriented code simply by reading it because of these disconnections. It's unclear that `ClassA`, for example, makes use of `ClassB` because it only uses `IInterfaceC` – which `ClassB` implements. This is on purpose, mind you, because we want to be as loosely coupled as possible so that we may swap out one class for another without having to redesign the class or classes that use the original. This, of course, comes at the cost of readability.

It's reasonably easy to read object-oriented code that uses a specific method of decoupling when you know what the method of decoupling is. A veteran of a project is less likely to have a harder time following code because she knows the decoupling policies. A new member to the project will have a much harder time following loosely coupled object-oriented code if they're unfamiliar with the decoupling policies or implementations.

It's fairly easy to understand linear code – just follow it from beginning to end. Object-oriented code often has a beginning but following it from step to step can be much more difficult. The creation of an object is often not performed in line with the steps being performed and it's much harder to know exactly where implementation of the next step is. This is of course if the object-oriented code isn't simply procedural code, hidden in an object-oriented language.

Code written in an object-oriented language, where each class is directly coupled to the classes it uses, is not much more object-oriented than the same code written in a procedural language. Sure, the additional meta-detail of classes (their name, what they encapsulate, and so on) increases the information within the code. But, the same information can be implemented in most procedural languages with concepts like modules, subfolders, namespaces, and so on.

In upcoming chapters, we'll explore in more detail ways in which object-oriented code can be refactored to improve various aspects of the code base, such as maintainability and quality.

Convention over Configuration

The **Convention over Configuration** (CoC) pattern—or design standard—details that the design of code should follow particular conventions instead of requiring *configuration*. This can be applied in many places. Certain frameworks have the ability to be configured for their use in your code base—which is where this standard emanated from. But, *configuration* can apply to many more things and at many more levels. Documentation, for example, is a form of configuration. It configures the reader. In the case of source code, if the source code requires the developer to read and understand documentation about the source code, the developer is required to absorb that configuration before the source code can be absorbed most efficiently.

Convention over Configuration has far reaching effects in the navigability of a particular code base. To a certain extent, the convention ends up being configuration as the developer must essentially be configured to use the convention. Convention over Configuration adds value, especially when the convention is an industry-standard convention. Developers learning the convention end up being junior so that some level of learning (or *configuration*) would need to be realized regardless; learning industry-standard conventions are a win-win.

Consistency

The principle of consistency goes a long way in making any of the navigability changes useful. If a particular convention isn't implemented consistently, despite the ability to improve navigability, the change may actually reduce navigability. Naming standards are a good example. If there is a convention to prefix interface types with the letter *I* and some interfaces don't follow this convention (good or bad) and leave it off or suffix with *Interface* then the code is difficult to navigate because it forces the reader to think and interpret before they can absorb the meaning or usage of the code.

Interfaces are a fairly benign example; the amount of thought involved in recognizing `DeviceInterface` as an interface type may be extremely small, but the rate at which reading and navigating code annoys the developer, I believe, is directly proportional to its navigability.

When dealing with certain patterns, it becomes increasingly important to define and be consistent with a particular convention. The Model-View-Controller pattern, for example, essentially mandates that two classes be created, one to implement the view and one to implement the controller. Each effectively implements support of a particular user-interface concept. To not suffix each with *View* or *Controller* can disconnect the class from the actual concept, introducing confusion and making the code that much harder to navigate.

The issue of consistency is often a consideration for the *I*-prefix of interface types convention. While most agree that polish notation is largely out-dated and unnecessary, the convention of prefixing interfaces with *I* persists. This continues to be a convention with many development teams for consistency with the .NET Framework. Since interfaces in the .NET Framework are consistently prefixed with *I*, it becomes confusing in non-.NET Framework code when interfaces are not prefixed with *I*. I believe this last vestige of Polish Notation lives on simply because it is used in the .NET Framework.

Conventions

We've touched on one convention (to prefix, or not prefix interface types with *I*). Outside the specific Convention over Configuration abilities of specific frameworks or libraries, there are certain conventions that can improve code navigability over and above consistency.

Naming

Naming conventions are probably the most vital part of the navigability of a code base. Names that are meaningless convey no information to the reader. Names that require specific knowledge, or – worse yet – a crib sheet, require the user to stop and think. This needlessly slows the reader down.

I won't get into too much detail on avoiding meaningless names. I'll leave it as an exercise to the user why `xyz` instead of `customerCount` reduces navigability. I will detail exceptions to this where names at face value are *meaningless* but acceptable because they are convention.

Mathematically, names like *x*, *y*, and *z* have meaning, but generally single-letter names have little value to the reader. An exception is iterated integer indexing variables – which are generally acceptable as a single character like *i*.

```
InvoiceLineItem[] invoiceLineItems =
    new InvoiceLineItem[invoiceLineItemCount];
for (int i = 0; i < invoiceLineItems.Length; ++i)
{
    invoiceLineItems[i] = new InvoiceLineItem();
    //...
}
```

This would be no more readable or understandable had it been written as the following:

```
InvoiceLineItem[] invoiceLineItems =
    new InvoiceLineItem[invoiceLineItemCount];
for (int index = 0; index < invoiceLineItems.Length; ++index)
{
    invoiceLineItems[index] = new InvoiceLineItem();
    //...
}
```

This is because this is a common convention. Right or wrong, this convention aids in the developers' speed at which they can read, absorb, and understand code because it is common and the developer does not need to think about it. Naming variables *i* where they are not iterators would cause confusion because the common convention suggests they should be iterators.

Depending on the project, there may be project-specific naming conventions that may need to be put into place, naming conventions that are specific to the project or the organization.

There are, fortunately, common naming standards to help your project have naming conventions that match the rest of the industry. This allows new members of your project to hit the ground running, with respect to naming. The *Framework Design Guidelines* and the *.NET Framework Developers Guide, Guidelines for Names* detail various naming conventions that will allow the naming in your code base to be consistent with the rest of the industry, with regard to the .NET community.



Names of Classes, Structs, and Interfaces: <http://msdn.microsoft.com/en-us/library/ms229040.aspx>

Names of Type Members: <http://msdn.microsoft.com/en-us/library/ms229012.aspx>

A benefit of adopting these naming conventions is that they can be checked or enforced by tools like Code Analysis or FxCop to analyze the code and check for naming violations. For example, the CA1710 warning will warn against various incorrect suffixes.

Microsoft Visual Studio® Code Analysis and FxCop are *static code analysis* tools. This class of tools analyses software without physically executing it (unlike dynamic analysis which analyses executing software. Visual Studio® Performance Analysis is an example of dynamic analysis). Generally, static code analysis tools analyze source code. Both Code Analysis and FxCop are slightly different than the classical static code analysis tools in that they analyze compiled code, but the analysis is the Intermediate Language that the compiler generates rather than the machine code that is physically executed on a given processor (which is generated by the runtime's just-in-time (JIT) compiler after an assembly is loaded for execution).

Keep in mind that naming conventions defined in these texts are fairly .NET generalized, so don't expect these tools to check all your other chosen suffixing conventions out-of-the-box.

Fortunately, changing a code base to consistently follow or implement a naming convention is fairly easy. The Rename refactoring is all you need to rename an errant name to a satisfactory name.

Scoping types with namespaces

Scoping types with namespaces allow you to logically organize code. Namespaces could be viewed as code subfolders. The specific scope created by putting types within a namespace allows for organizing of the code into logical units. The units are first-class citizens in C#, so they are checked and validated by the compiler.

Similar organization can be attained through comments, but comments are effectively ignored by the compiler and thus can be entered incorrectly or they may be completely missing and the compiler will not warn you.

As with many refactorings, not refactoring to increase the depth of the namespace hierarchy of your project doesn't mean your code isn't perfectly functional. What we are attempting to address in this chapter is refactoring the structure of code to make it easier to navigate. Our refactoring effort to improve readability also improves navigability to a certain extent. For example, we can navigate to all references to a class simply by searching for it by name—if we've performed Rename refactorings to make our identifier declaration naming consistent and free of spelling errors. Searching is more accurate, thus quicker to navigate with.

Adding organization through increasing the namespace hierarchy isn't directly associated with readability. Moving a class to a sub-namespace, for example, doesn't help readability. Use of using directives generally hides namespaces from code and thus doesn't come into focus while reading. There are cases where performing a Move To Namespace refactoring along with Rename refactoring can improve the readability of some code.

In our `Sales` and `PartnerInvoice` example, we can change the following code:

```
SalesInvoice salesInvoice = new SalesInvoice(null, null);
PartnerInvoice partnerInvoice =
    new PartnerInvoice();
```

Now, `SalesInvoice` and `PartnerInvoice` are moved to their own namespaces (`Invoicing.Domain.Sales` and `Invoicing.Domain.Partner` respectively) and both are renamed to "Invoice", as exemplified in their usage:

```
Invoice salesInvoice = new Invoice(null, null);
Invoicing.Domain.Partner.Invoice partnerInvoice =
    new Domain.Partner.Invoice();
```

This isn't necessarily easier to read. But, let's with the following:

```
SalesInvoice invoice = new SalesInvoice(null, null);
invoice.FooterText = "All sales final.";
//...
```

Performing the same Move to Namespace refactorings along with Rename refactorings will end up with the following:

```
Invoice invoice = new Invoice(null, null);
invoice.FooterText = "All sales final.";
//...
```

This is, in my opinion, easier to read on its own.

IDE navigation

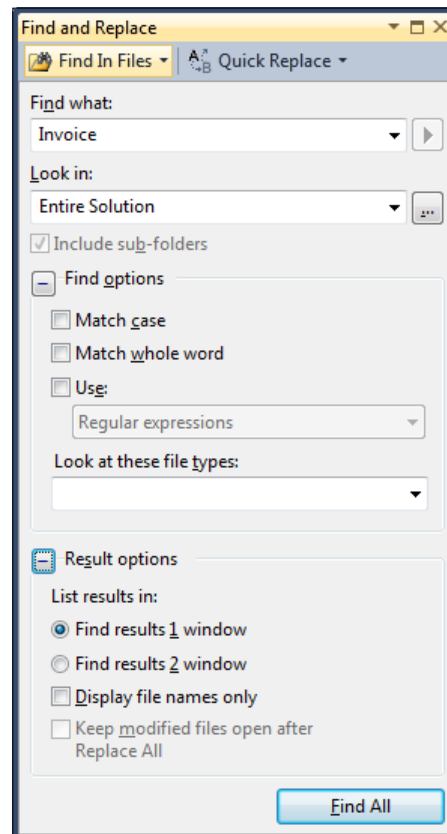
Now that we've seen design techniques that aid in navigability, let's see some of the functionality in Visual Studio® where this improved navigability comes into play.

The Visual Studio® 2010 IDE offers many ways of navigating your code base over and above simply opening a file and scrolling around the text with the mouse or the arrow keys. We'll detail many of these features and discuss how we can refactor our code to accommodate each of these features to make the code base easier to navigate and thus easier to maintain.

Search

One of the easiest ways of navigating code in Visual Studio® 2010 is to search for it with **Find In Files**. This searches for code based on the text you provide for the **Find what** text.

The following is the **Find and Replace** form in Visual Studio® 2010:



The **Find and Replace** form (in the **Find in Files** mode) consists of a textbox to enter the **Find What** text, find options, and result options. The find options allow us to specify case sensitivity, whether only whole words should match, and whether to use some form of wildcard matching. The result options configure where the results are displayed and what results are displayed.

When searching for particular identifiers in code, using the **Match case** and the **Match whole word** options can be very useful. If we wanted to find `Invoice` and not `SalesInvoice`, we could enter "Invoice" for the **Find what** text, and check the **Match case** and **Match whole word** options.

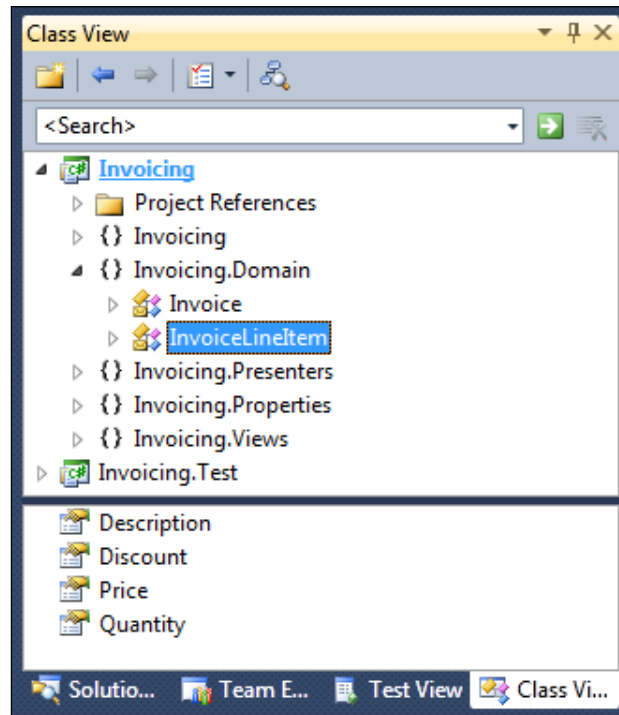
Due to the fact that this is a simple textual search, it's important to have consistent naming. If a developer was working with the `AddCustomerForm` form and they wanted to find out where the `Add Customer Controller` was used, they might decide to search for `AddCustomerController`. If a consistent naming convention wasn't in place or the other developer had not suffixed the `Add Customer controller` class with "Controller", the developer would not find any results. The developer would first have to find out the exact name of the controller class (possibly without using `Find in Files`).

Now, this is a somewhat contrived example to illustrate a point. Depending on the circumstances, there are much better ways to find all the references to a particular type. *Find All References*, for example, finds all references to a particular type or member based on Visual Studio®'s internal hierarchy of the code (based on a form of compilation of the code).

To fix an inconsistent naming that would make the code base hard to navigate, use the `Rename` refactoring.

Class View

Another form of navigating code in Visual Studio® 2010 is the **Class View**. As seen in the following diagram, the **Class View** is a hierarchical display of the classes, their namespaces, and their members:



The **Class View** organizes classes by their project and their namespaces.

There are two panes in the Class View, the upper Objects pane and the lower Members pane. The Objects pane mirrors the structure you've given to your solution (projects, namespaces, types, and so on) and allows you to explore that structure in much the same way as you'd explore the file system. It allows you to view your code base at a higher level and drill-down to the actual code by double-clicking a particular type or member. Without a consistent namespace convention, the usefulness of the Class View to navigate your solution is reduced. The hierarchy may be flattened and may lump all classes within the same level.

To make the Class View more efficient, classes may be grouped and moved to specific namespaces with the **Move To Namespace** refactoring. Unfortunately, there isn't a built-in refactoring to perform this; the simplest way to move a class to another namespace is to change the namespace declaration for a particular class.

```
namespace Invoicing
{
    public class Invoice
    {
        private IInvoiceGrandTotalStrategy
            invoiceGrandTotalStrategy;
        public Invoice(IEnumerable<InvoiceLineItem>
            invoiceLineItems,
            IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy)
        {
            InvoiceLineItems = new
                List<InvoiceLineItem>(invoiceLineItems);
            this.invoiceGrandTotalStrategy =
                invoiceGrandTotalStrategy;
        }
        //...
    }
}
```

This can be another nested namespace declaration as follows:

```
namespace Invoicing
{
    namespace Domain
    {
        public class Invoice
        {
            private IInvoiceGrandTotalStrategy
                invoiceGrandTotalStrategy;
            public Invoice(IEnumerable<InvoiceLineItem>
                invoiceLineItems,
                IInvoiceGrandTotalStrategy
                invoiceGrandTotalStrategy)
            {
                InvoiceLineItems = new
                    List<InvoiceLineItem>(invoiceLineItems);
                this.invoiceGrandTotalStrategy =
                    invoiceGrandTotalStrategy;
            }
            //...
        }
    }
}
```

Or it can be nested with the dotted namespace syntax, as follows:

```
namespace Invoicing.Domain
{
    public class Invoice
    {
```

```
private IInvoiceGrandTotalStrategy
    invoiceGrandTotalStrategy;
public Invoice(IEnumerable<InvoiceLineItem>
    invoiceLineItems,
    IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy)
{
    InvoiceLineItems = new
        List<InvoiceLineItem>(invoiceLineItems);
    this.invoiceGrandTotalStrategy =
        invoiceGrandTotalStrategy;
}
//...
```

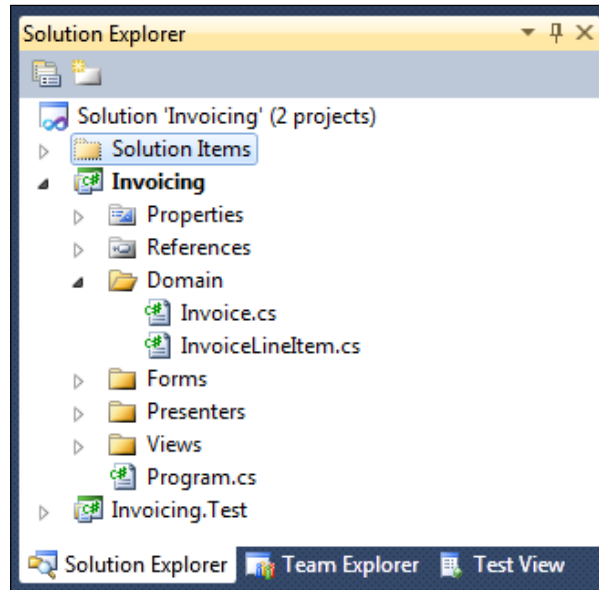
Once the class is moved to the other namespace, the using directive needs to be changed in all the other classes that use this one. I find the easiest way to get this done is to simply start a re-compile. The errors resulting from the class name that can't be found make a nice work list. First, simply double-click each error in the **Error List** to navigate to the file with the error. Then, right-click the class usage with the error in the code, and choose **Resolve\using...**

If your project has other errors (you're not refactoring and adding features, are you?) then using Find All References before moving to the new namespace may be more useful. Simply right-click the class you're going to move to a new namespace, choose **Find All References**. Then, change the namespace for the class that you want to move to a different namespace. The entries left in the Find All References list are your work list. First, simply double-click an item in **Find Symbols Results** to navigate to the file that uses the class whose namespace has just changed. Then, right-click the class usage that now has an error and choose **Resolve\using...**

Depending on your project's guidelines, it may be necessary for classes to be within a sub-folder with the same name as the namespace it is contained within. This is a very common guideline because, out of the box, Visual Studio® automatically does this when adding class files to folders. So, to be consistent with what Visual Studio® does, many projects make the inverse to be a guideline: files in namespaces must be contained in a folder structure that matches that of the namespace. In this case, part of the refactoring will also include moving the file to the new or existing sub-folder. This refactoring is easily accomplished by clicking the class in the Solution Explorer and dragging it to another sub-folder. If the sub-folder does not exist yet, right click on the project or the folder where you would like to create the folder and choose **Add\New Folder**.

Solution Explorer

Visual Studio® 2010 has the tried-and-true Solution Explorer, which is depicted in the following screenshot:



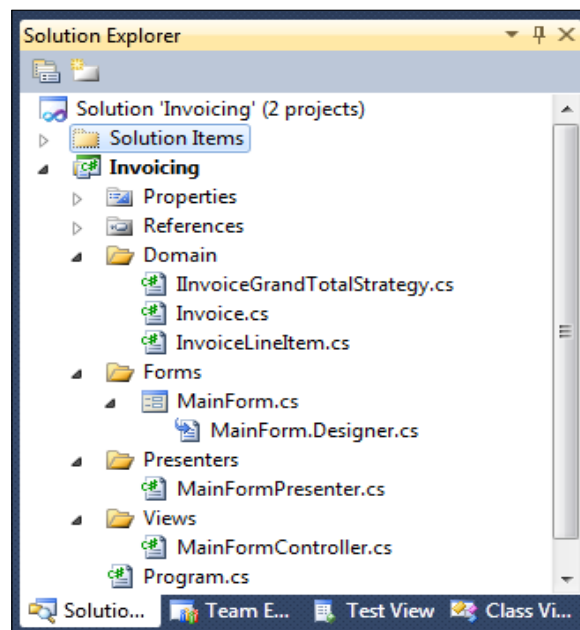
The Solution Explorer organizes the projects, sub-folders, solution folders, and files in a hierarchical display. You are able to act on each of the nodes in the hierarchy in different ways. Depending on what node you right-click on, you can perform actions like **Add Item**, **Rename**, **Display Properties**, and so on.

The **Solution Explorer** is much like hierarchical views like the Class View, except it is the main dashboard for the solution. Despite similar interfaces in other places, the Solution Explorer allows you to perform almost everything you can possibly do to solution items either directly or indirectly. Because the Solution Explorer is so powerful and it has an inherent ability to access all items in a solution, it's the primary means of navigating projects and solutions. Optimizing the code structure to take advantage of the Solution Explorer means increased navigability in our projects.

The Solution Explorer organizes solutions by project, folders, and files. The lowest level of granularity is by file. If your classes aren't organized by files, you're not getting as much out of the Solution Explorer as you could be. For example, if you have more than one class in a file, there are certain things you can't do with the Solution Explorer. Solution Explorer allows you to rename a file that, in turn, allows you to rename the class and all references to it (effectively performing the Rename refactoring). If your class is not contained within its own file (whose name matches that of the class) you can't use this short-cut to Rename refactoring. You can still rename it directly in the code or via Class View.

There's no built-in refactoring to move a class to its own file, but it's easy to manually perform this refactoring. Simply right-click on the location Solution Explorer you would like the class to exist (either a project itself, or a folder within a project) to have the class created in a sub-namespace. Then choose **Add\Class**, enter the name of the existing class and click **Add**. A new file is created with a class, with the same name as an existing class. Simply cut and paste the previous class into the new file, update the using directives, and you're done. You can perform the Add Class portion of this refactoring in many other places, like the Class View, Class Diagram, Sequence Diagram, and so on. But, due to the sheer number of tasks that can be performed from the Solution Explorer, it is most likely visible and awaiting your command.

If your project has a mandate to physically keep classes in different namespaces in physical sub-folders, you'll have a project structure like the following in the Solution Explorer:



Moving classes from the root to a sub-folder is accomplished by dragging it from its location and dropping it on a sub-folder. Unfortunately, the Solution Explorer does not act like the Windows 7 Explorer and does not expand sub-folders when you hover a dragged file over a sub-folder. So, you must have the folder in which the sub-folder is contained expanded in the Solution Explorer.

When projects start out, it may not be necessary to have many, if any, added namespaces. In fact, if the project starts out small enough, adding namespaces may hinder navigation. It's fairly rare that something starting out that small would end up being so large as to benefit from namespaces, but it's not unheard of. As the project evolves and grows in size, it's very likely that namespaces in which classes live need to be changed. Depending on your design methodology, there may simply be no need for any sort of namespace hierarchy (remember You Ain't Going To Need It?). So, refactoring the namespace hierarchy is a very common refactoring.

Refactoring the namespace hierarchy is an excellent way of improving the navigability of the code base in light of contemporary Integrated Development Environments that provide high-level views of the source code. Namespaces help categorize and organize code, which makes the code easier to navigate with such tools.

In more complex systems, an effective namespace hierarchy is vital in the ability to name classes correctly. A good example of this can be found in many places in the .NET Framework. The `Timer` classes are a good example. There are currently three `Timer` classes in the .NET Framework. Each works best in different scenarios (and each is in its own namespace). If they were not in different namespaces the designers would have to result to unique names like `SystemTimer`, `FormTimer`, and `ThreadingTimer`—none of which would add any value to developers. "Timer" makes sense in each scenario, regardless of where each should be used.

When deciding on reusing a name already in use in another namespace, it's important to make sure there's value to it. Having duplicate names in different namespaces for the sake of having duplicate names, or for the sake of namespaces, isn't a valid reason. For example, if the two classes are expected to be used in the same places, the same name will be a hindrance to usability. The developer will be forced to add a `using` alias or to use a fully-qualified name. For example:

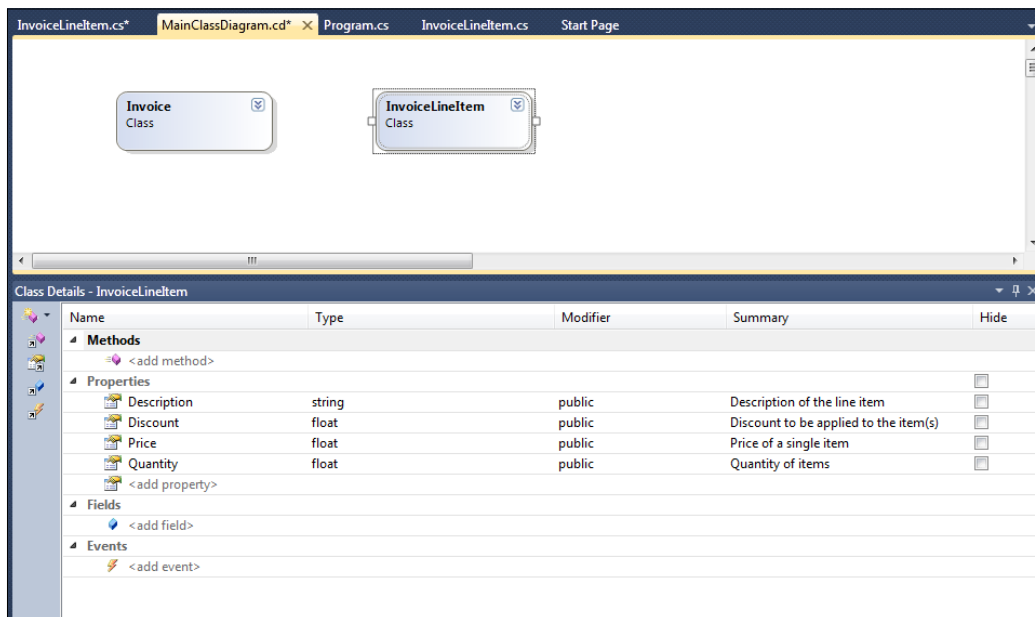
```
Invoice salesInvoice = new Invoice(null, null);
Invoicing.Domain.Partner.Invoice partnerInvoice =
    new Domain.Partner.Invoice();
```


In this example, the designer has created two `Invoice` classes. One in the `Sales` namespace (and the above code resolves this simply as `Invoice` because we have a `using Invoicing.Domain` directive at the top of the file). There's clearly some need for two unique `Invoice` classes: one to deal with general sales and one to deal with partners. It becomes difficult to use the two classes together if they don't have a unique name. This makes it harder to write code and harder to understand the code. Depending on the circumstances, it may have simply been better to have a `PartnerInvoice` and do away with the `Sales` and `Partner` sub-namespaces.

In any case, it's easy to refactor this to be more navigable. Simply perform the `Move To Namespace` refactoring and `Rename` refactoring and change the fully-qualified declarations.

Class Diagram

There exists a document type in Visual Studio® 2010 that depicts several classes in diagrammatic form. This is the **Class Diagram**. It is depicted in the following screenshot:



The Class Diagram contains two panes: The top design surface pane and the bottom class details pane. The design surface pane diagrammatically shows the classes it contains and their relationship.

The Class Diagram is very similar in its navigational abilities to the Class View. The design surface pane is similar to class nodes in the Class View, and the **Class Details** pane is similar to the lower Members pane in the Class View.

The Class Diagram allows you to create a subset of classes within a project to which you can navigate and edit. You can double-click on a class on the diagram surface in the same way as you can double-click a class in the Class View and the Solution Explorer, to navigate to a specific class. You can also double-click members of a class in the Class Details pane, in the same way as you can double-click members in the lower Members pane of the Class View, to navigate to a specific member of a class.

The Class Diagram allows you to edit much of the same attributes of classes as does the Class View. You can Rename refactor (through the *F2* keyboard shortcut in addition to the Context Menu) the class and its members, add members, and edit members. Editing members includes everything except the body (if applicable) of the member including type, modifier, XML docs, and so on. Unfortunately, the Class Diagram doesn't allow you to multi-select members so you can't modify members (like modifier) en-masse.

With large, complex projects, using a Class Diagram as a means of navigation is useful because you can create a sub-view of your project.

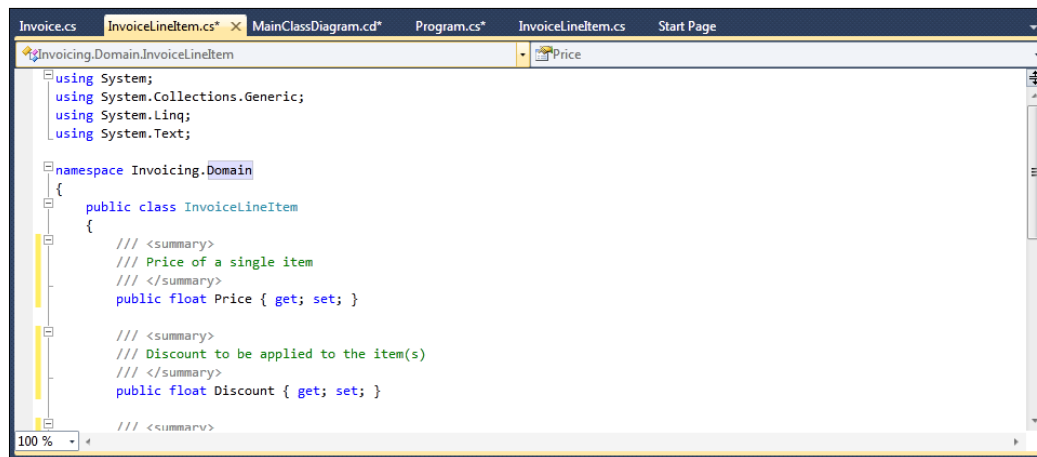
With navigational tools such as Solution Explorer and Class View, you have no way of filtering out or focusing what subset of the project/solution you want to navigate. You're limited to seeing everything and filtering out information by collapsing namespaces. This isn't too much of an issue if how you want to focus your navigation is on parity with how the code is structured with respect to namespaces. If what subset you want to navigate isn't on parity with its namespace structure, a Class Diagram can be useful.

To create a class diagram, simply right-click where you want the file to exist (project or project folder) in a project in the Solution Explorer, and choose **Add Item....** Select **Class Diagram** as the template type, enter a name for the class diagram in **Name** and click **Add**. You can now drag and drop what classes you want to be shown in the diagram by dragging them from the Solution Explorer or the Class View onto the Class Diagram design surface. You can place all the classes from a particular namespace into a Class Diagram by dragging a project folder from the Solution Explorer onto the Class Diagram design surface, or by dragging a namespace from the Class View onto the Class Diagram design surface. Class may be removed from the Class Diagram (without removing them from the project) by selecting the class and pressing the *Del* key.

What the Class Diagram displays depends on what the developer adds to it; so, there's less refactoring to make use of the Class Diagram optimal. To make use of the Class Diagram optimal, design of the classes contained within it should be logical. Refactoring for readability is enough to make use of the Class Diagram optimal. Refactoring for readability may involve Rename refactoring, Move To Namespace Refactoring, and so on.

Code Editor

The last navigational tool in Visual Studio® 2010 that we'll cover is the Code Editor. "The Code Editor, how is that a navigational tool?" you ask. The Code Editor is just like other views of code, such as the Class Diagram. It just happens to view the code in raw form. The Code Editor is depicted in the following screenshot:



The Code Editor consists of text editor area, the ability to expand and contract regions of code (to the left of the text editor area), the **Class Name/Types** combo box above the text editor area, and the **Method Name/Members** combo box to the right of the **Class Name/Types** combo box.

The Code Editor knows quite a bit about the code that it edits. You may not think about it but you may often navigate around in code from within the Code Editor. From the Code Editor you can do many things that other views of code also allow, other than simply typing text. You can perform actions like Find all References and Go to Definition.

Find All References is useful when you're viewing a particular class and you want to see where it is used. Find All References results in a list of locations where the selected class is referenced in the solution. Double-clicking on a particular list item navigates to the file and row and column where that item is referenced. Find All References also works with variables and members to display a list of where a particular variable or member is referenced within the entire solution. Find All References has the shortcut *Ctrl+,* (Ctrl+comma).

Go to Definition is useful when you're editing code and you need to look at the source for type, member, or variable that is referenced in other code. Using Go to Definition takes you directly to the definition of the type, member, or variable. In the case of a type (like a class or an interface) Go To Definition takes you to the top of the type definition in whatever class it is in. Go To Definition has the shortcut *F12*.

Navigating Backward and Forward

When using navigational tools like Find All References and Go to Definition, it's quite common to want to return to where you navigate from. This can be done with the **Navigate Backward** command button. The Navigate Backward command returns you to where you were when you navigated to another piece of code with actions like Go to Definition or double-clicking an entry in the Find All References results list. The Navigate Backward is also enabled when you navigate more than 10 lines away from the current cursor position with the mouse and when opening a new file (Navigate Backward navigates back to the previous file and the cursor position).



Once you navigate backward you can then navigate (or repeat the previous navigation) forward with the **Navigate Forward**.

The **Navigate Forward** and **Navigate Backward** commands are available on the Standard toolbar, next to the Undo and Redo buttons, and are depicted in the following image:



With the standard C# Visual Studio® keyboard layout, you can also navigate backward with *Ctrl+-* (hold *Ctrl* down while pressing the *minus* key) and navigate forwards with *Ctrl+Shift+-*.

For the most part, nothing in particular needs to be done to promote navigability in the text editor. You can go directly to declarations based on the text in the file or navigate directly to a member based on the content of the **Method Name/Members** combo box. The granularity of the code affects the range in which you can navigate to. For example, if you have a large method, the navigational techniques available in the text editor limit the number of places you can navigate to. To improve navigability in the text editor, refactor any large method code smells. Breaking a large method into multiple methods increase the number of places you can navigate with the text editor's **Method Names/Members** combo box. The ability to understand the code will hinder navigation, so naming is also an important aspect of code navigability within the text editor. Use Rename refactor to ensure meaningful and consistent names.

Part of what makes code more navigable is the reader's ease with which they can navigate to a particular location in the code base. For the most part, Visual Studio® 2010 offers many different techniques to perform this navigation, with a variety of criteria. But, some of that criterion depends upon the reader's knowledge of the code base. For example, searching for a particular type requires that the reader knows almost exactly the name of the type. Naming standards and conventions and consistent use of them go a long way in making code more navigable.

Navigation with design patterns and principles

The purpose of many design patterns in object-oriented design is to acknowledge that quality is tied to how decoupled a design is by introducing and/or increasing decoupling. Some of the more notable patterns and principles are the Model View Controller/Presenter and the Dependency Inversion principle. These patterns and this principle increases decoupling.

The **Model View Controller/Model View Presenter** patterns (**MVC/MVP**) provide a pattern of decoupling presentation logic from business logic. The logic that performs business decisions like enforcing business rules is decoupled from the presentation and its logic (like how to add items to user interface controls).

The Dependency Inversion principle designs designers into keeping classes decoupled in the general case. This promotes independence and allows for greater reusing and increases extensibility.

As more classes become physically decoupled from one another and become coupled to abstractions, it becomes much harder to navigate the code, even with navigation abilities of contemporary Integrated Development Environments.

With the MVC/MVP pattern, the Model is abstracted away from the user interface aspects by way of a Controller or Presenter. The Controller and Presenter are then responsible for direct coupling to particular classes in the Model (or Domain). It isn't necessary for the Controller or the Presenter to be directly coupled to particular classes within the Model, but they would end up being used only by the Controller or Presenter. The Controller and the Presenter are responsible for getting data from/to the Model to/from the View (or the user-interface component).

So, when viewing the user-interface code (the View) it's difficult to trace back to the actual source of the data being used by the View.

Dependency inversion almost always involves creating a dependency on an abstraction rather than a concrete type (this is the opposite of creating a dependency on a concrete type, thus it's *inverted*). This is usually done through some form of interface-driven design, where the dependency is upon an interface whose implementation is injected into the dependant. This injection can occur through the constructor, through property setters, through setter methods, or even through a proxy like a Factory or some other creational service.

Decoupling classes from one another can involve use of specific decoupling method, methods like interface-driven design, or use of a framework like an **Inversion of Control** Container (**IoC** container). We'll discuss refactoring to support these types of decoupling, but when these decoupling methods are used it can make the code harder to navigate. The code is easier to navigate when the particular method of decoupling is known, but it's important for code to be structured in specific ways and pre-defined conventions used in order to minimize the effect on navigability.

So how can we refactor our code to accommodate this? As we detailed earlier in the chapter, one of the most important ways is to perform Rename refactoring in order to use consistent naming conventions to make types and members easier to discover.

We'll get more into refactoring code to support patterns like Model View Controller and Model View Presenter in future chapters. Future chapters will also cover refactoring to principles like Dependency Inversion.

Summary

So, we've seen that the ability of a developer to navigate through the code quickly, easily, and without valueless thought is a key attribute of a code base's ability to be maintained and changed. There are many aspects that can affect the navigability of a code base, including things like identifier naming, source code organization, and how the Integrated Development Environment supports navigating the code base. We've detailed the refactorings that can be performed in order to improve navigability, such as Rename refactor, Move To Namespace refactoring, Move Class To File refactoring, and so on.

In the following chapter, we'll begin to look at how we can refactor code to help improve quality. We'll delve deeper into some object-oriented principles and how to refactor code that doesn't follow these principles. We'll also look at some design methodologies, their pros and cons, and how to refactor towards better design methodologies.

5

Improving Design Correctness

Design Correctness is a pretty strong term. For many, this is a very subjective term. For the most part, if you ask two developers to look at the same design and have them evaluate it, they'll give you a different list of changes they would make to the design to make it *correct*. The focus of this chapter is understanding generally-accepted design principles and methodologies, examples of violating them, and how to refactor code to alleviate the maintenance issues resulting from these violations.

I'm not going to try to convince you that your design needs to change to accommodate everything in this book. You need to evaluate any paradigm, technology, or pattern for yourself and decide if it works in your circumstances. You need to decide whether it fits your scenario, whether it needs improvement, or whether there's a completely different technique that works better. When this book was written, there were generally accepted industry principles and patterns that this book uses, in part, to look at how to focus refactoring efforts.

This chapter focuses on some primarily design-focused principles and methodologies that are generally accepted for improving a software system's design. Most object-oriented principles, techniques, patterns, and methodologies attempt to improve overall design by addressing the quantity and scope of dependencies and coupling. It's generally accepted that reducing dependencies and unnecessary coupling to those dependencies improves the ability for a code base to be maintained by changing and evolving it over time. This, in turn, has an observed tendency to help improve quality.

The principles and methodologies we'll detail in this chapter are as follows:

- Liskov substitution principle
- Composition over inheritance

- Object-oriented design and object behavior
- Move initialization to declaration refactoring
- Value types

Liskov substitution principle

In 1988, Barbara Liskov posited about what it means for a class to be a subtype of another. She detailed *What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.* Robert Martin coined the name **Liskov Substitution Principle (LSP)** and describes it in his *Engineering Notebook* column *The Liskov Substitution Principle* as *functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.* Essentially, what this means is that a derived class should act exactly like a super class would when used as an object reference of the super class.



Barbara Liskov, *Data Abstraction and Hierarchy*, SIGPLAN Notices, 23,5 (May, 1988).

Detecting or avoiding violating the Liskov Substitution Principle can be difficult at times. It's a principle that can be overlooked by even the most seasoned of software development designers. Most of the time the violation is very subtle, despite the design appearing logical and sometimes matching the real world object.

The quintessential example is a Square class that derives from a Rectangle:

```
public class Rectangle
{
    public virtual Color Color { get; set; }
    public virtual float CalculateArea()
    {
        return Height * Width;
    }
    public virtual float Width { get; set; }
    public virtual float Height { get; set; }
}
public class Square : Rectangle
{
    public override float Height
```

```
{
  get
  {
    return base.Height;
  }
  set
  {
    // Set the width to be
    // the same as the height
    // to enforce a square.
    base.Height = value;
    base.Width = value;
  }
}
public override float Width
{
  get
  {
    return base.Width;
  }
  set
  {
    // Set the height to be
    // the same as the width
    // to enforce a square.
    base.Width = value;
    base.Height = value;
  }
}
}
```

In this example, we have a simple `Rectangle` class that is an abstraction for information about a `Rectangle`. For our domain, we simply need to know that a rectangle has a width and height that can be changed independently. We have another class, `Square`, that derives from `Rectangle`, but it differs in that both the width and the height are synchronized.

Violating the Liskov Substitution Principle often stems from a misguided drive to model *is-a* relationships from the real world in a virtual world. The `Rectangle/Square` scenario is a perfect example of this. While *is-a* relationships can be modeled in object-oriented software, issues arise when the subclass does not behave the same as the superclass. In the `Rectangle/Square` scenario, we don't view rectangle or square as having any sort of behavior. Even when modeling rectangles and squares in software, we don't generally view a `Rectangle` or `Square` as having behavior due to the way we design them. In the example implementation, a rectangle is simply a container of two floats: the width and the height. They overlook the implicit behavior that the rectangle has.

One way of seeing the problem that arises is when we try to write a unit test for the `Rectangle` and `Square` classes. There's really not much to test with a rectangle class, so one test is to verify that there is no interaction between width and setting the height (and vice versa). This could be done with something like the following:

```
/// <summary>
///This is a test class for RectangleTest and is intended
///to contain all RectangleTest Unit Tests
///</summary>
[TestClass()]
public class RectangleTest
{
    private Rectangle CreateRectangle()
    {
        return new Rectangle();
    }
    /// <summary>
    ///A test for Rectangle Constructor
    ///</summary>
    [TestMethod()]
    public void RectangleConstructorTest()
    {
        Exception exception = null;
        try
        {
            Rectangle target = CreateRectangle();
        }
        catch (Exception e)
        {
            exception = e;
        }
        Assert.IsNull(exception,
            "Rectangle constructor threw exception.");
    }
    /// <summary>
    ///A test for Height
    ///</summary>
```

```
[TestMethod()]
public void HeightTest()
{
    Rectangle target = CreateRectangle();
    float expected = 42F;
    float actual;
    target.Height = expected;
    actual = target.Height;
    Assert.AreEqual(expected, actual);
}
/// <summary>
///A test for Width
///</summary>
[TestMethod()]
public void WidthTest()
{
    Rectangle target = CreateRectangle();
    float expected = 24F;
    float actual;
    target.Width = expected;
    actual = target.Width;
    Assert.AreEqual(expected, actual);
}
/// <summary>
///A test for potential interactions
///</summary>
[TestMethod()]
public void InteractionTest()
{
    Rectangle target = CreateRectangle();
    float expectedWidth = 24F;
    float expectedHeight = 42F;
    float actualHeight;
    float actualWidth;
    target.Width = expectedWidth;
    target.Height = expectedHeight;
    actualHeight = target.Height;
    actualWidth = target.Width;
    Assert.AreEqual(expectedWidth, actualWidth);
    Assert.AreEqual(expectedHeight, actualHeight);
}
}
```

All of these tests pass fine when tested solely with a `Rectangle` object. But, when `Square` is substituted for its base class, we see that `InteractionTest` fails with a message *Assert.AreEqual failed. Expected:<24>. Actual:<42>*.

This points out that `Square`, when used as a `Rectangle`, does not behave intuitively. A non-intuitive API (which is the combination of `Square` and `Rectangle`, and is a unique API) increases the likelihood that someone will use the API incorrectly – in this case, incorrectly in the expectation that a `Square` behaves the same as a `Rectangle`.

The `square/rectangle` scenario is a fairly easy scenario to refactor. In the real world, we know that a square is a type of rectangle (as static shapes) but, does the code base really need to view a square specifically as a type of a rectangle? Many times we tend to transfer our real-world understanding of the objects we're modeling into our domain without any specific need to. We've seen examples of this in previous chapters, but it's been more explicit. Deriving `Square` directly from `Rectangle` introduces implicit behavior and side-effects.

Fortunately, this is easy to refactor in Visual Studio® 2010. The easiest is to perform a **Convert to Sibling** refactoring.



Convert to Sibling refactoring is changing the base class of particular class to be the same as another class.

Convert to Sibling refactoring

Performing the Convert to Sibling class is not an automatic refactoring supported by Visual Studio® 2010. So, it must be done manually; but, it's fairly straight forward. The best case is simply that the class being changed needs to change its base class. This case is fairly rare because if you're deriving from another class, you're most likely overriding something. The next best case is that some functionality from the superclass needs to be copied to the subclass and the class's base class is changed or removed. The worse case is that the class being modified becomes nonsensical when the logic from the base is copied to it. This is the case with the `rectangle/square` scenario. Performing a refactoring that merely copies logic from the base and removes the inheritance, we'd have something like this:

```
public class Square
{
    public float height;
    public float width;
    public float Height
    {
        get
        {
            return height;
        }
    }
}
```

```
    }
    set
    {
        height = value;
        width = value;
    }
}
public float Width
{
    get
    {
        return width;
    }
    set
    {
        width = value;
        height = value;
    }
}
}
```

By removing the inheritance from `Rectangle`, `Square` becomes a very unclear class. We now have a class with two properties to essentially contain the same value. You'd want to get rid of one of the properties and focus on just one, like `width`. Doing that, our refactoring ends up like this:

```
public class Square
{
    public float Width { get; set; }
}
```

This, of course, requires that everything that was given a `Square` object and expected a `Rectangle` object be changed to accept a `Square` object. This isn't always possible. So, this refactoring may need to introduce a common base for both `Square` and `Rectangle` to replace places where `Rectangle` is expected but `Square` can be given. This is heavily dependent on the context in which the two classes are used. In one case, this could be as simple as creating a new base class `Shape`, and moving all common behavior and attributes (`CalculateArea()` and `Color`) to `Shape` and deriving `Rectangle` and `Square` from `Shape`. We would complete this refactoring by selectively replacing references to `Rectangle` with `Shape` — where appropriate.

Refactoring to single class

So far, we have focused on refactorings that deal with retaining the subclass. Sometimes retaining this class simply isn't feasible. We'll look at another, less pedantic example of a subclass that isn't truly substitutable for its base:

```
/// <summary>
/// Abstraction of customer
/// attributes and behavior
/// </summary>
public class Customer
{
    private List<Contact> contacts = new List<Contact>();

    public Customer(DateTime creationDate,
        String accountNumber)
    {
        AccountNumber = accountNumber;
        CreationDate = creationDate;
        Contacts = contacts;
    }

    /// <summary>
    /// Textual representation of
    /// customer account number
    /// </summary>
    public String AccountNumber { get; private set; }

    /// <summary>
    /// Contact information for the customer
    /// </summary>
    public IEnumerable<Contact> Contacts { get; private set; }

    /// <param name="description"></param>
    /// <summary>
    /// Comments about the customer
    /// </summary>
    public virtual void AddContact(Contact contact)
    {
        contacts.Add(contact);
    }

    /// <summary>
    /// Remove <paramref name="contact"/> from
    /// this customer
    /// </summary>
    /// <param name="contact"></param>
    public virtual void RemoveContact(Contact contact)
```

```
{
    contacts.Remove(contact);
}

/// <summary>
/// Date the customer was created
/// </summary>
public DateTime CreationDate { get; private set; }

/// <summary>
/// Textual description of the customer
/// </summary>
public String Description { get; private set; }

/// <summary>
/// Add description for the customer
/// </summary>
public virtual void DescribeCustomer(String description)
{
    Description = description;
}

/// <param name="description"></param>
/// <summary>
/// Comments about the customer
/// </summary>
public String Comments { get; private set; }

/// <summary>
/// Add comments to the customer
/// </summary>
/// <param name="comment"></param>
public virtual void CommentOnCustomer(String comment)
{
    Comments = comment;
}
}

/// <summary>
/// Abstraction of an inactive customer's
/// attributes and behavior
/// </summary>
public class InactiveCustomer : Customer
{
    public InactiveCustomer(DateTime creationDate,
        String accountNumber)
        : base(creationDate, accountNumber)
    {
```



```
    }
    /// <summary>
    /// Add description for the customer
    /// </summary>
    public override void DescribeCustomer(string description)
    {
        throw new InvalidOperationException("Cannot change "
            + " description of an Inactive customer.");
    }
    /// <summary>
    /// Add contact information to the contact
    /// </summary>
    /// <param name="contact"></param>
    public override void AddContact(Contact contact)
    {
        throw new InvalidOperationException("Cannot "
            + " add contacts to an Inactive customer.");
    }
    /// <summary>
    /// Remove <paramref name="contact"/> from
    /// this customer
    /// </summary>
    /// <param name="contact"></param>
    public override void RemoveContact(Contact contact)
    {
        throw new InvalidOperationException("Cannot remove"
            + " contacts from an Inactive customer.");
    }
    /// <summary>
    /// Add comments to the customer
    /// </summary>
    /// <param name="comment"></param>
    public override void CommentOnCustomer(string comment)
    {
        throw new InvalidOperationException("Cannot add"
            + " comment top an Inactive customer.");
    }
}
```

The designer has designed a `Customer` class that is an abstraction for customer information and behavior. It contains information about a customer that is required to perform business operations. This information contains things like contact information, description, comments, account number, the date when the customer first became a customer, and so on.

The designer also designed an `InactiveCustomer`. This is a customer that, for whatever reason, is no longer active in the system. The designer chose to implement `InactiveCustomer` as a specialization of `Customer`.

As you can see in the implementation of `InactiveCustomer`, there is certain behavior that an inactive customer does not support. In this domain, that includes changing the contacts, description, and comments. In order to implement this, the programmer has chosen to simply throw an `InvalidOperationException`.

The designer intended that an `InactiveCustomer` would be instantiated to represent a customer that is no longer active. Without putting too much thought into it, it's easy to think that inactive customer has an *is-a* relationship with customer. There are many good arguments why this isn't the case, but we're going to concentrate on the problems that arise when substituting `InactiveCustomer` for its base, `Customer`.

It's logical to think that an `InactiveCustomer` can't have behaviors such as changing contacts, but this means that `InactiveCustomer` can't be used anywhere a `Customer` object is expected. For example, if we pass an `InactiveCustomer` object to a method that accepts a `Customer` object and that method makes a call to `AddContact`, an exception will occur. In a situation like this, there are two non-refactoring options. One is that the exception is caught and something different is done instead. This is difficult due to how general the exception is — `InvalidOperationException` doesn't offer much detail about why the exception is being thrown, which doesn't offer much criteria to decide what to do in light of the exception. Another option is to check to see what type of `Customer` object we're dealing with and write different code when it's an `InactiveCustomer` object. This, of course, circumvents polymorphism and negates the benefits of inheritance.

So, how do we refactor in this case? For the most part, this may be domain-specific. So, depending on your circumstances you may need to do something different than what I'm going to propose.

The flaw in the design of these two classes isn't necessarily that an inactive customer is a type of customer, it's that an inactive customer behaves the same as an active customer. `Customer` becomes the model for an active customer due to `InactiveCustomer` and the way it's designed. As you might be able to tell, an inactive customer can be modeled in a better way.

The way of refactoring `Customer/InactiveCustomer` involves recognizing that the behavior that needs to be modeled is that of an attribute of a customer, not unique behaviors. Once recognizing this, refactoring is straight forward: a flag is added to the `Customer` class and logic is moved from `InactiveCustomer`, use of `InactiveCustomer` is removed and replaced with `Customer` and use of the new flag, and the `InactiveCustomer` class is removed. This results in a single `Customer` class similar to the following class:

```
public class Customer
{
    private List<Contact> contacts = new List<Contact>();

    public Customer(DateTime creationDate,
        String accountNumber)
    {
        AccountNumber = accountNumber;
        CreationDate = creationDate;
        Contacts = contacts;
    }

    // previously discuss members removed for clarity...
    public virtual void AddContact(Contact contact)
    {
        if (Inactive)
        {
            throw new InvalidOperationException("Cannot add"
                + " contacts from an Inactive customer.");
        }
        contacts.Add(contact);
    }

    public virtual void RemoveContact(Contact contact)
    {
        if (Inactive)
        {
            throw new InvalidOperationException("Cannot remove"
                + " contacts from an Inactive customer.");
        }
        contacts.Remove(contact);
    }

    public virtual void AddContact(Contact contact)
    {
        if (Inactive)
        {
            throw new InvalidOperationException("Cannot add"
                + " contacts from an Inactive customer.");
        }
    }
}
```

```
    }
    contacts.Add(contact);
}

public virtual void RemoveContact(Contact contact)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot remove"
            + " contacts from an Inactive customer.");
    }
    contacts.Remove(contact);
}

public virtual void DescribeCustomer(String description)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot change"
            + " description of an Inactive customer.");
    }
    Description = description;
}

public virtual void CommentOnCustomer(String comment)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot add"
            + " comment top an Inactive customer.");
    }
    Comments = comment;
}

public void Deactivate()
{
    Inactive = true;
}

public void Reactivate()
{
    Inactive = false;
}
}
```

Where `InactiveCustomer` was being created, this could be refactored to the following:

```
Customer inactiveCustomer =
    new Customer(creationDate, customerAccountNumber);
inactiveCustomer.Deactivate();
```

Composition over inheritance

After seeing some of the issues that arise from not correctly using inheritance, the next logical step is to look at a principle that avoids inheritance altogether. This principle is the **Prefer Composition over Inheritance principle**.

Composition over Inheritance means that classes should be composed of instances of other classes to implement behavior or attributes instead of inheriting behavior or attributes from a base class. This means that a class would contain one or more fields that are references to other classes, and the current class delegates work to them.

Let's look at a slightly different design for `Invoice`. An invoice could be considered simply as a collection of `InvoiceLineItems`. As such, we could implement it by being a specialization of a collection type, `List<T>` for sake of argument. In this case, we'd have a class similar to the following:

```
public class Invoice : List<InvoiceLineItem>
{
    public Invoice(IEnumerable<InvoiceLineItem>
        invoiceLineItems,
        IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy)
        : base(invoiceLineItems)
    {
        this.invoiceGrandTotalStrategy =
            invoiceGrandTotalStrategy;
    }

    public void GenerateReadableInvoice(Graphics graphics)
    {
        graphics.DrawString(HeaderText,
            HeaderFont,
            HeaderBrush,
            HeaderLocation);

        float invoiceSubTotal = 0;
        PointF currentItemLocation = LineItemLocation;
        foreach (InvoiceLineItem invoiceLineItem in this)
        {
```

```
float lineItemSubTotal =
    CalculateLineItemSubTotal (invoiceLineItem);
graphics.DrawString (invoiceLineItem.Description,
    InvoiceBodyFont,
    InvoiceBodyBrush,
    currentLineItemLocation);
currentLineItemLocation.Y +=
    InvoiceBodyFont.GetHeight (graphics);
invoiceSubTotal += lineItemSubTotal;
}

float invoiceTotalTax =
    CalculateInvoiceTotalTax (invoiceSubTotal);
float invoiceGrandTotal =
    invoiceGrandTotalStrategy.CalculateGrandTotal (
        invoiceSubTotal,
        invoiceTotalTax);
CalculateInvoiceGrandTotal (invoiceSubTotal,
    invoiceTotalTax);

graphics.DrawString (String.Format (
    "Invoice SubTotal: {0}",
    invoiceGrandTotal - invoiceTotalTax),
    InvoiceBodyFont, InvoiceBodyBrush,
    InvoiceSubTotalLocation);
graphics.DrawString (String.Format ("Total Tax: {0}",
    invoiceTotalTax), InvoiceBodyFont,
    InvoiceBodyBrush, InvoiceTaxLocation);
graphics.DrawString (String.Format (
    "Invoice Grand Total: {0}",
    invoiceGrandTotal), InvoiceBodyFont,
    InvoiceBodyBrush, InvoiceGrandTotalLocation);

graphics.DrawString (FooterText,
    FooterFont,
    FooterBrush,
    FooterLocation);
}

// previously discussed members removed for clarity...
public Font HeaderFont { get; set; }
public Brush HeaderBrush { get; set; }
public Font FooterFont { get; set; }
public Brush FooterBrush { get; set; }
public Font InvoiceBodyFont { get; set; }
```

```
public Brush InvoiceBodyBrush { get; set; }  
}
```

We have an `Invoice` class that derives from `List<T>`. The logic to store the `InvoiceLineItems` is inherited from `List<T>`. We initialize the base in the constructor by passing an `IEnumerable<InvoiceLineItem>` to the `List<T>` constructor (`: base (invoiceLineItems)`) and iterate the collection using this in the `foreach` loop.

This is fairly straightforward, and for our scenario it works fine. But, as we've discussed, we're not simply trying to write software that works; we're trying to write maintainable software that works.

In this example, we're limiting our `Invoice` class to always having an *is-a* relationship with a collection of `InvoiceLineItems`. While an invoice is always going to have a collection of `InvoiceLineItem` objects, that isn't going to be the only collection of things an invoice may contain. An invoice may also contain a collection of addresses (bill-to, ship-to, and so on); it may contain a collection of taxes (try designing an invoice to support multiple regions with no consistent tax structure without supporting an unknown number of taxes); or, it may contain a collection of payment methods (that is, the purchaser wants to split up the payment between cash, Mastercard, Visa, Amex, and so on).

.NET only supports inheritance from a single base class. To implement multiple-inheritance, you'll have to build up a complex hierarchy. For example, for `Invoice` to inherit from two classes, one of the base classes would have to inherit from the other. This, of course, really limits what we can inherit from and really limits our ability to subsequently inherit at all in the future. If we inherit from `List<T>`, we're always a type of `List<T>`. Consumers of our class will become coupled to `List<T>` because we're republishing its interface.

Another problem with inheritance is that it really makes our design details public. A class must be concrete, at least partially, in order to be inherited from. We're forcing our consumers to depend on concretions they have no control over. This also forces our class to continue depending on that implementation detail. Our consumers are now coupled to this implementation detail and we can't change it. This reduces the flexibility in which to evolve our code, either in response to design/requirements changes or to bug fixes.

Although lack of multiple-inheritance may seem like a limitation, it's a good thing. It forces designers to think about inheritance. It prevents misusing inheritance through multiple-inheritance. One of the side-effects of this is that inheritance is now a limited resource. Having one and only one inheritance slot means many designers try to keep this free, just in case – so, they avoid inheritance.

Inheritance, or subtype polymorphism, is really only useful when you know one class can really stand in for another. As we've seen, it's easy to misjudge *is-a* relationships between classes and end up paying the price for it. Inheritance introduces a coupling between two classes. Although the subclass obviously depends on the superclass, there is now a dependency from the superclass to the subclass. The superclass is now more restricted in how it can evolve itself.

Any member that a class publishes must be expected to be used; so, any changes to that class's existing interface must be thoroughly thought through. But, a superclass's behavior is inherited by the subclass. So, not only must the changes to member signatures (the interface) need to be well calculated, but now any change to the behavior of virtual members needs to be well calculated.

One easy example of this is our customer/inactive-customer scenario, where suddenly introducing an exception in the behavior of a superclass means anything that uses the subclass also needs to expect this new behavior.

What else does inheritance give us and how do we refactor? One thing that subtype polymorphism gives us is extensibility. I can create a superclass that defines a set of base functionality and extension points as virtual/abstract members. I can then create an interface to accept references to this type of object. A consumer of this interface can choose to create a subtype of this class and override the virtual members to extend existing functionality. For example:

```
public class Invoice : IDisposable
{
    private bool disposed = false;
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    protected virtual void OnDisposed()
    {
    }
    private void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // TODO: dispose of disposable members
            }
            disposed = true;
        }
    }
}
```



```
        OnDisposed();
    }
}
```

In this modified `Invoice` class, we've begun to add support for `IDisposable` members. We've derived from `IDisposable` and implemented the Dispose Pattern. In addition to that, we've added a virtual `OnDispose` method so a derived class can do something special when the object has been disposed.

Refactoring virtual methods to events

In our particular example, this allows derived classes to be informed of disposal, but this is very limited. In all likelihood more than just the base class may be interested in knowing when an object is disposed. This type of functionality is better implemented by means of events. We can easily refactor this class to implement an event instead of a virtual method by adding a public event, invoking the event where needed, and removing the virtual method. This results in a class similar to the following:

```
public class Invoice : IDisposable
{
    private bool disposed = false;
    public event EventHandler Disposed;
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    private void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                IDisposable disposable = HeaderBrush as
                    IDisposable;
                HeaderBrush = null;
                if (disposable != null)
                {
                    disposable.Dispose();
                }
                disposable = FooterBrush as IDisposable;
                FooterBrush = null;
                if (disposable != null)
```

```
        {
            disposable.Dispose();
        }
        disposable = FooterFont as IDisposable;
        FooterFont = null;
        if (disposable != null)
        {
            disposable.Dispose();
        }
        disposable = HeaderFont as IDisposable;
        HeaderFont = null;
        if (disposable != null)
        {
            disposable.Dispose();
        }
        disposable = InvoiceBodyBrush as IDisposable;
        InvoiceBodyBrush = null;
        if (disposable != null)
        {
            disposable.Dispose();
        }
        disposable = InvoiceBodyFont as IDisposable;
        InvoiceBodyFont = null;
        if (disposable != null)
        {
            disposable.Dispose();
        }
    }
    EventHandler disposedEventHandler = Disposed;
    if (disposedEventHandler != null)
    {
        disposedEventHandler(this, EventArgs.Empty);
    }
    disposed = true;
}
}
```

Once we have this class we need to modify any derived classes to replace the virtual method with an event handler. This can be done by changing the signature of the method and adding that to the event handler. For example:

```
protected void OnDisposed(object caller, EventArgs e)
{
    //...
}
```

To complete refactoring a derived class, assignment to the base's event should be added in the derived class's constructor – adding a constructor if none exists.

Instead of changing the signature of the virtual method, you can simply let Visual Studio® create a new method for you when you write the line of code to assign a value to the base's `Disposed` event handler, then move the body of the virtual method to the event handler and delete the virtual method.

The benefit of implementing events is that you can now have more than just the base class notified, as opposed to a single base class with virtual methods.

Exceptions to preferring composition

Prefer Composition over Inheritance is a principle for a reason; there are exceptions to this principle. It's not always possible to do what you want without using inheritance. There are many frameworks that rely on subtype polymorphism as the primary means of extension. The .NET Framework's WinForms is a good example of this. In order to implement a WinForm form, you must derive from `System.Windows.Forms.Form`.

Replace inheritance with delegation refactoring

Refactoring to composition instead of inheritance involves making use of an instance of another class instead of inheriting from another class. This refactoring is called **Replace Inheritance with Delegation**. The class is now delegating responsibility to another class instead of taking it on itself by inheriting it from another class, as was previously done when `Invoice` derived from `List<InvoiceLineItem>`:

```
public class Invoice
{
    private IInvoiceGrandTotalStrategy
        invoiceGrandTotalStrategy;
    private List<InvoiceLineItem> invoiceLineItems;
    public Invoice(IEnumerable<InvoiceLineItem>
```

```
        invoiceLineItems,
        IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy)
    {
        this.invoiceLineItems = new
            List<InvoiceLineItem>(invoiceLineItems);
        this.invoiceGrandTotalStrategy =
            invoiceGrandTotalStrategy;
    }
    //...
    public void GenerateReadableInvoice(Graphics graphics)
    {
        graphics.DrawString(HeaderText,
            HeaderFont,
            HeaderBrush,
            HeaderLocation);

        float invoiceSubTotal = 0;
        PointF currentLineItemLocation = LineItemLocation;
        foreach (InvoiceLineItem invoiceLineItem in
            invoiceLineItems)
        {
            float lineItemSubTotal =
                CalculateLineItemSubTotal(invoiceLineItem);
            graphics.DrawString(invoiceLineItem.Description,
                InvoiceBodyFont,
                InvoiceBodyBrush,
                currentLineItemLocation);
            currentLineItemLocation.Y +=
                InvoiceBodyFont.GetHeight(graphics);
            invoiceSubTotal += lineItemSubTotal;
        }

        float invoiceTotalTax =
            CalculateInvoiceTotalTax(invoiceSubTotal);
        float invoiceGrandTotal =
            invoiceGrandTotalStrategy.CalculateGrandTotal(
                invoiceSubTotal,
                invoiceTotalTax);
        CalculateInvoiceGrandTotal(invoiceSubTotal,
            invoiceTotalTax);

        graphics.DrawString(String.Format(
            "Invoice SubTotal: {0}",
            invoiceGrandTotal - invoiceTotalTax),
            InvoiceBodyFont, InvoiceBodyBrush,
            InvoiceSubTotalLocation);
        graphics.DrawString(String.Format("Total Tax: {0}",
            invoiceTotalTax), InvoiceBodyFont,
```

```
        InvoiceBodyBrush, InvoiceTaxLocation);
    graphics.DrawString(String.Format(
        "Invoice Grand Total: {0}",
        invoiceGrandTotal), InvoiceBodyFont,
        InvoiceBodyBrush, InvoiceGrandTotalLocation);
    graphics.DrawString(FooterText,
        FooterFont,
        FooterBrush,
        FooterLocation);
}
// previously discussed members removed for clarity...
}
```

Object-oriented design and object behavior

Data-driven design is a form of bottom-up design where the design of the system is based on the data that will be contained within the system. Of course, the data in the system can't be modeled without knowing the requirements. Focusing on just the data ignores requirements of how the system needs to work and focuses on implementation details.

Focusing on just the data in the system means making implementation decisions too early. Accurately modeling data in a system may require use of tools to manage that model. This generally means using a RDBMS or using a data-modeling tool that only supports an RDBMS.

In many organizations, investment in the relational model of the system is heavy and the tendency to reuse that investment is high. There is a tendency for the domain object model to be generated from the data model. There are actually commercial tools to generate code based on an RDBMS.

Data-driven design when designing your domain model is very problematic. When classes that contain state (data) don't also contain the behavior that operates on that state we lose much of what we consider object-orientation. When we don't take into account the behavior that an object is supposed to model, we end up modeling what a RDBMS allows. For example:

```
public class Customer
{
    private List<Contact> contacts = new List<Contact>();
    public String AccountNumber { get; set; }
```

```
public IEnumerable<Contact> Contacts { get; set; }
public DateTime CreationDate { get; set; }
public String Description { get; set; }
public String Comments { get; set; }
public bool Inactive { get; set; }
}
```

This customer class is just a container for data. It doesn't contain any business logic. The business rule where comments can't be updated when a customer is inactive would have to be executed within another class when `Comments` is assigned to it. This would have to be executed in all the places where `Comments` is assigned. Someone making an assignment would have to know and remember to also do the same. If this business rule is changed, all of this code would have to be changed.

A class designed this way technically implements data-hiding. The actual fields aren't visible to the outside world, but nothing is truly being hidden.

If object-orientation wasn't an issue, we'd still have a problem with this model as it pertains to the evolution of the data. If a DBA decides that the database was normalized enough and performed some normalization tasks, adding or removing tables, these classes would have to be updated and anything that used them would have to be updated. This tightly couples the domain to the data.

Data-driven design makes for a very rigid design. Developers push back on making changes to accommodate business logic and business rule changes because of the effort to keep everything synchronized.

I've seen a few systems where the *data layer* was simply a collection of data-driven classes like this, which were populated and saved to a database, and the *business logic layer* was a collection of classes that didn't need to be persisted to the database but didn't have anything directly to do with the UI. What in fact happens with architectures like this is that the business logic and rules are strewn across the UI and the business logic layer, and in some cases strewn across the UI, the business, and the data layer. And while pragmatically the system may work, it's very fragile and very hard to make changes to. Architectures like this defeat the purpose of layering. Their lack of focus and principles generally mean a fundamental tenant of layer is violated by introducing circular dependencies between the layers.

Designing data-driven classes like this are not a problem on their own. They become a problem when they're misused within the rest of a system. Designing data-driven classes like the previous example is fine if they're only used as a form of getting data from the database to our object-oriented code and are an implementation detail of the data layer. Domain objects – objects that model the state and behavior – would then be instantiated based on the data in these data-driven classes and used by the rest of the system. There are, fortunately, specific classes of tools or frameworks that create these data-driven classes called **Object/Relational Mappers (ORMs)**, which we'll discuss in a later chapter.

There is actually a design pattern around this particular use of data-driven designed classes called the **Data Transfer Object (DTO)**. A Data Transfer Object is a data-driven class whose purpose is to transfer data from one subsystem to another. This is common in some data access designs, where the data access layer would populate a DTO and that data would be used to populate or instantiate a domain object to be used by the rest of the system. Data Transfer Objects become very useful when you're dealing with systems whose subsystems are distributed and must communicate together over-the-wire. Design of a domain object for use in-memory can often be orthogonal to how data is transferred over the wire or to the layers responsible for communicating the data. Service-oriented architectures (SOA) often use web service APIs that have distinct restrictions on the type and format of data. Data Transfer Object helps to shield the implementation details of transferring data to and from subsystems without having to couple it to layers where it doesn't belong or cannot be implemented.

If you find that you're dealing with a code base where the domain layer (or the domain objects, if you have no layering) is mostly data-driven classes or that you have some particularly data-driven classes, refactoring can get very complex.

Refactoring to improved object-orientation

Refactoring data-driven classes generally follow a particular purpose, moving behavior (or logic) into a class and modifying use of the class by other classes. In our `Customer` example, we would remove some setters from some properties and introduce explicit behavior to the class.

```
/// <summary>
/// Abstraction of customer
/// attributes and behavior
/// </summary>
public class Customer
{
    private List<Contact> contacts = new List<Contact>();
```

```
public Customer(DateTime creationDate,
    String accountNumber)
{
    AccountNumber = accountNumber;
    CreationDate = creationDate;
    Contacts = contacts;
}
/// <summary>
/// Textual representation of
/// customer account number
/// </summary>
public String AccountNumber
{
    get;
    private set;
}
/// <summary>
/// Contact information for the customer
/// </summary>
public IEnumerable<Contact> Contacts
{
    get;
    private set;
}
/// <summary>
/// Add contact information to the contact
/// </summary>
/// <param name="contact"></param>
public virtual void AddContact(Contact contact)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot add"
            + " contacts from an Inactive customer.");
    }
    contacts.Add(contact);
}
/// <summary>
/// Remove <paramref name="contact"/> from
/// this customer
/// </summary>
/// <param name="contact"></param>
public virtual void RemoveContact(Contact contact)
```



```
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot remove"
            + " contacts from an Inactive customer.");
    }
    contacts.Remove(contact);
}

/// <summary>
/// Date the customer was created
/// </summary>
public DateTime CreationDate
{
    get;
    private set;
}

/// <summary>
/// Textual description of the customer
/// </summary>
public String Description
{
    get;
    private set;
}

/// <summary>
/// Add description for the customer
/// </summary>
/// <param name="description"></param>
public virtual void DescribeCustomer(String description)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot change"
            + " description of an Inactive customer.");
    }
    Description = description;
}

/// <summary>
/// Comments about or by the customer, their preferences,
///
/// </summary>
public String Comments
{
```

```
    get;
    private set;
}

/// <summary>
/// Add comments to the customer
/// </summary>
/// <param name="comment"></param>
public virtual void CommentOnCustomer(String comment)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot add"
            + " comment top an Inactive customer.");
    }
    Comments = comment;
}

/// <summary>
/// Whether the customer is active or not
/// true if inactive, false if active.
/// </summary>
public bool Inactive
{
    get;
    private set;
}

/// <summary>
/// Deactivate the customer
/// </summary>
public void Deactivate()
{
    Inactive = true;
}

/// <summary>
/// Reactivate the customer
/// </summary>
public void Reactivate()
{
    Inactive = false;
}
}
```

In this new class, the highlighted changes include changing the `Inactive`, `Comments`, `Description`, `AccountNumber`, and `Contacts` properties to get-only, and the behaviors `Deactivate`, `Reactivate`, `CommentOnCustomer`, `DescribeCustomer`, `AddContact` and `RemoveContact` were added to model the expected behavior (or how the object is used) of the class.


I find this refactoring to be the best when performing one behavior at a time. With `Customer`, for example, I would start with a behavior such as changing activation: reduce the visibility of the `Deactivated` property, introduce `Deactivate` and `Reactivate` methods, then update all the code that used the `Deactivated` property to use either the `Deactivate` method or the `Reactivate` method.

Move initialization to declaration

In C# there are various options to initialize variables. Variables include member and local variables. These variables can be simple types (one value) or they can be aggregate types (a composition of multiple values).

The first time a variable that is a simple type is assigned, it is considered initialized. Initialization of aggregate types is more complex. Aggregate types can be designed to be completely *initialized* via a particular constructor, or the aggregate type may only have a default constructor and *initialization* must occur through multiple property/field assignments. There's also a hybrid of the two where *initialization* occurs with both a constructor and field/property assignment.

I emphasize initialized and initialization in the above paragraph because for most aggregate types, initialization is subjective. If the type is **mutable** it can be changed throughout its lifespan and whether or not it's initialized depends on the code using it and not on the type itself.

 **Mutable Type:** A type that once instantiated can be modified.

The following is an example of a mutable type:

```
public class PersonName
{
    public string GivenName { get; set; }



    public string Surname { get; set; }
}
```

This is a straightforward aggregate type that aggregates the components of a simple person's name. It contains the given name of the person (or the *first* name, in many Western cultures) and the surname of the person (or *last* name, in many Western cultures). It only contains a default constructor (implicit, as we haven't defined any constructors).

A typical initialization of this type would look like this:

```
PersonName personName = new PersonName();
personName.GivenName = "Peter";
DoSomething();
personName.Surname = "Ritchie";
```

This initialization is not an **atomic operation**; it's done in multiple steps involving multiple instructions.

 **Atomic operation:** An operation that has only one observable side effect. 

Although this particular snippet of code normally can't have its side effects observed by any other thread than the current thread, and use of the `personName` object is done after all the properties have been assigned, it has three observable side effects. The first is a `personName` object where `GivenName` and `Surname` both have the value `String.Empty`. The second is that `GivenName` has the value `Peter` and the `Surname` has the value `String.Empty`. The third is that `GivenName` has the value `Peter` and `Surname` has the value `Ritchie`.

A `PersonName` instance is free to be changed at any time in its lifespan. Its value may change to something else for whatever purpose. For example, we may have code like the following:

```
PersonName personName = new PersonName();
personName.GivenName = "Peter";
DoSomething();
personName.Surname = "Ritchie";
Use(personName);
personName.GivenName = "Dennis";
Use(personName);
```

In this example, the code is *initializing* `personName` to *Peter Ritchie* for the first call to `Use()`, then *initializing* `personName` to *Dennis Ritchie* for the second call to `Use()`.

Obviously this code works as it is, but it has some maintainability issues. As written, *initialization* is implicit. It's implied that `personName` is or should be initialized to *Peter Ritchie* before the first call to `Use()` and that it should be initialized to *Dennis Ritchie* prior to calling `Use()` the second time. These initializations are implicit because we're not atomically initializing `personName`. As it is written, another person can edit the code to be something like the following:

```
PersonName personName = new PersonName();
personName.GivenName = "Peter";
DoSomething();
var customer = LookupCustomer(personName);
personName.Surname = "Ritchie";
Use(personName);
personName.GivenName = "Dennis";
Use(personName);
```

This introduces a new method call to `LookupCustomer` that makes use of `personName`.

Value types

Value types don't represent an entity, or something with an identity. A value type generally models something like an attribute or a characteristic. A value is something that acts as a whole. `System.DateTime` is an example of a value type. November 5, 1970 is always November 5, 1970. Changing the day of the month of November 5, 1970 to 11 doesn't change November 5, 1970; it makes another date, November 11, 1970. A value type isn't the same as a .NET value type – or a type modeled with a `struct`.



Value types are types that don't have an identity and don't represent something else, they represent a value.

Refactoring to value type

Refactoring to a value type is simple. It basically entails making the type immutable and providing the ability to construct the value in all possible ways. With our `PersonName`, this could be done by refactoring the class as follows:

```
public class PersonName
{
    public PersonName(string givenName, string surname)
    {
        GivenName = givenName;
        Surname = surname;
    }
}
```

```
    }  
    public PersonName(string givenName)  
        : this(givenName, String.Empty)  
    {  
    }  
    public string GivenName { get; private set; }  
    public string Surname { get; private set; }  
}
```

Then, modify all the code that assigns values to the `GivenName` or `Surname` properties and change the call to the constructor. Depending on how the `PersonName` instances are used, this may also require introducing a new constructor call. For example, after refactoring the `PersonName` class and changing the existing constructor calls, one of our previous examples is left with an orphaned `GivenName` assignment:

```
PersonName personName = new PersonName("Peter", "Ritchie");  
DoSomething();  
var customer = LookupCustomer(personName);  
Use(personName);  
personName.GivenName = "Dennis"; // orphaned  
Use(personName);
```

In order to resolve this, we need to introduce a new instance. This final refactoring results in something similar to the following:

```
PersonName personName = new PersonName("Peter", "Ritchie");  
DoSomething();  
var customer = LookupCustomer(personName);  
Use(personName);  
personName = new PersonName("Dennis", "Ritchie");  
Use(personName);
```

In the context of our example, this is just dandy. But, be careful when performing this refactoring in other contexts. We started with a single instance of a `PersonName` object and re-used it; in other contexts there may be some implicit side-effects that result from this re-use that you'd lose by creating another instance. If this happens, I would suggest you've got a design issue or two to address; it's something you should watch out for.

Making a value type immutable becomes important due to the way they are used and their lifespan. Value types are intended to be used as values (hence the name), and they're meant to be modeling domain-specific values that represent primitive types. Value types are generally transient; they don't live on their own beyond the invocation of the application in which they are used, that is, they're not persisted to some sort of repository unless they're aggregated in another type that does have an identity.

Immutable types are not unique to Value Types. As parallelization and concurrency becomes more mainstream taking greater advantage of multi-core and multi-processor computers, immutability becomes more important. Sending a message from one thread to another that is mutable, requires synchronization of changes to the object. Changing an object in Thread A must not be interrupted by modifications of Thread B, for example, because this will end up corrupting the data contained within the object. Rather than having to deal with thread synchronization, making types are used amongst threads immutable it eliminates the ability to modify data being used by multiple threads and eliminates the need for expensive and error-prone synchronization.

Modeling business rules appropriately

What we've shown here is a class that doesn't completely model the business rules of the domain or the organization. In all likelihood, there exist rules around what a valid *person name* is. We've modeled a class that only contains a given name and a surname; so, we have explicitly modeled a business rule that details that a person name only contains the given name and the surname. This is perfectly acceptable for many domains. But, implicitly what is modeled is that both the given name and surname are optional. This is clearly incorrect, I know of no domain where a person can be nameless.

What the programmer is forced to do is implement the business rules associated with a person name throughout the entire code base (or more specifically, where `PersonName` is used). What this does, of course, is make it hard to enforce the business rule as well as hard to maintain it. If the rule changes, we have that many places to change how a person name is used correctly.

I'm not suggesting that the `PersonName` class should take on the responsibility of validating itself directly. Validation of business rules should generally be explicit and autonomous. But, whether or not a class is instantiated correctly often depends on the design of that class's constructor. In our `PersonName` example, we haven't limited how someone using the `PersonName` class can create an instance. So, they're able to create any permutation of properties.

If our business rules stipulate that a person name is only valid if they have a given name and a surname, then refactoring `PersonName` is as simple as making `PersonName` immutable and providing a single constructor that accepts the given name and the surname. For example:

```
public class PersonName
{
    public PersonName(string givenName, string surname)
    {
        GivenName = givenName;
        Surname = surname;
    }

    public string GivenName { get; private set; }

    public string Surname { get; private set; }
}
```

These changes remove the default (implicit) constructor, make the `GivenName` and `Surname` properties read-only outside of the `PersonName` class, and introduce a single constructor to initialize `GivenName` and `Surname` at instantiation.

Summary

In this chapter, we've seen specific ways we can improve our design to reduce coupling and improve maintainability. We've seen how to refactor to principles like Composition over Inheritance, refactor violations of the Liskov Substitution Principle, and refactor to practices like Value Types and initializing at declaration. We've also seen how we can refactor our classes to be more object-oriented and focus on the behavior as well as the state contained within a class.

In the next chapter, we'll expand on design-based refactoring. We'll look at specific ways to refactor code to improve the potential for quality. We will examine improving cohesion and decreasing coupling and principles that relate to cohesiveness and decoupling.

6

Improving Class Quality

Larry Constantine is attributed with the creation of systematic measurement of software quality. In the mid-to-late seventies, Larry Constantine (and Ed Yourdon) attributed several things to the quality of software code. Under the umbrella of structured design, among those attributes of quality software code were cohesion and coupling. At the time they associated quality with generality, flexibility, and reliability. We're going to concentrate on generality and flexibility and how cohesion and coupling can be applied to increase code quality.



Larry Constantine: http://en.wikipedia.org/wiki/Larry_Constantine.

Cohesion

Cohesion applies to many different disciplines. Cohesion in physics relates to the force that keeps molecules integrated and united and that makes the molecule what it is. A highly cohesive molecule is one that tends to remain autonomous and not adhere to or blend with other molecules. In geology, cohesion is the strength of substances to resist being broken apart. In linguistics, it's the degree to which text is related. Text whose content relates to the same subject is cohesive.

Cohesion in software is very similar to cohesion elsewhere. A cohesive block of code is a block of code that relates well together and has the ability to remain together as a unit. Object-oriented programming brings distinct cohesive abilities. All programming languages have certain cohesive abilities, such as their ability to group code in modules, source files, functions, and so on. A programming language's ability to define physical boundaries enables cohesiveness. A module, for example, defines a specific boundary for which some content is retained and other content is repelled. Code from one module can use code from another module, but only in specific and defined ways – usually independent of language syntax. All code within a module has innate cohesion: their relation amongst themselves as being contained within the module.

Any rule, principle, guideline, or practice needs to be implemented thoughtfully. This text isn't a manual on how you must perform your refactoring; it's a description of several types of refactorings and their impetus. By the same token, this text doesn't set out to prove the benefits of any particular rule, principle, guideline, or practice. "Mileage may vary" and incorrect usage will often negate most, if not all, benefits. I'll leave it as an exercise for the reader to find the research that "proves" the benefits of any particular rule, principle, guideline, or practice. This text assumes the generally accepted benefits of various principles and practices, including cohesion, as an indicator of quality. If you decide that the benefits aren't outweighing the costs, it's up to you to decide not to implement that principle.

Class cohesion

Object-orientation brings extra cohesive abilities to the programmer. The programmer has the ability to relate code together within a *class*. Other code can use the code within a class, but only through its defined boundaries (the class's methods and properties).

In object-oriented design, cohesion is generally much more than simply code contained within a class. Object-oriented cohesiveness goes beyond the physical relation of code within a class and deals with the relation of meaning of the code within a class.

Object-oriented language syntax allows the programmer to freely relate code to other code through a class definition, but this doesn't mean that code is cohesive. For example, let's revisit our *Invoice* class so far.

```
/// <summary>
/// Invoice class to encapsulate invoice line items
/// and drawing
/// </summary>
public class Invoice
```

```
{
    private IInvoiceGrandTotalStrategy
        invoiceGrandTotalStrategy;
    public Invoice(IEnumerable<InvoiceLineItem>
        invoiceLineItems,
        IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy)
    {
        InvoiceLineItems = new
            List<InvoiceLineItem>(invoiceLineItems);
        this.invoiceGrandTotalStrategy =
            invoiceGrandTotalStrategy;
    }
    private List<InvoiceLineItem> InvoiceLineItems
    {
        get;
        set;
    }
    public void GenerateReadableInvoice(Graphics graphics)
    {
        graphics.DrawString(HeaderText,
            HeaderFont,
            HeaderBrush,
            HeaderLocation);

        float invoiceSubTotal = 0;
        PointF currentLineItemLocation = LineItemLocation;
        foreach (InvoiceLineItem invoiceLineItem in
            InvoiceLineItems)
        {
            float lineItemSubTotal =
                CalculateLineItemSubTotal(invoiceLineItem);
            graphics.DrawString(invoiceLineItem.Description,
                InvoiceBodyFont,
                InvoiceBodyBrush,
                currentLineItemLocation);
            currentLineItemLocation.Y +=
                InvoiceBodyFont.GetHeight(graphics);
            invoiceSubTotal += lineItemSubTotal;
        }
        float invoiceTotalTax =
            CalculateInvoiceTotalTax(invoiceSubTotal);
        float invoiceGrandTotal =
            invoiceGrandTotalStrategy.CalculateGrandTotal(
```

```
        invoiceSubTotal,
        invoiceTotalTax);
CalculateInvoiceGrandTotal (invoiceSubTotal,
        invoiceTotalTax);
graphics.DrawString (String.Format (
    "Invoice SubTotal: {0}",
    invoiceGrandTotal - invoiceTotalTax),
    InvoiceBodyFont, InvoiceBodyBrush,
    InvoiceSubTotalLocation);
graphics.DrawString (String.Format ("Total Tax: {0}",
    invoiceTotalTax), InvoiceBodyFont,
    InvoiceBodyBrush, InvoiceTaxLocation);
graphics.DrawString (String.Format (
    "Invoice Grand Total: {0}",
    invoiceGrandTotal), InvoiceBodyFont,
    InvoiceBodyBrush, InvoiceGrandTotalLocation);
graphics.DrawString (FooterText,
    FooterFont,
    FooterBrush,
    FooterLocation);
}

public static float CalculateInvoiceGrandTotal(
    float invoiceSubTotal, float invoiceTotalTax)
{
    float invoiceGrandTotal = invoiceTotalTax +
        invoiceSubTotal;
    return invoiceGrandTotal;
}

public float CalculateInvoiceTotalTax(
    float invoiceSubTotal)
{
    float invoiceTotalTax =
        (float) ((Decimal) invoiceSubTotal *
            (Decimal) TaxRate);
    return invoiceTotalTax;
}

public static float
CalculateLineItemSubTotal(
    InvoiceLineItem invoiceLineItem)
{
    float lineItemSubTotal =
        (float) ((decimal) (invoiceLineItem.Price
            - invoiceLineItem.Discount)
```

```
        * (decimal)invoiceLineItem.Quantity);
    return lineItemSubTotal;
}

public string HeaderText { get; set; }
public Font HeaderFont { get; set; }
public Brush HeaderBrush { get; set; }
public RectangleF HeaderLocation { get; set; }
public string FooterText { get; set; }
public Font FooterFont { get; set; }
public Brush FooterBrush { get; set; }
public RectangleF FooterLocation { get; set; }
public float TaxRate { get; set; }
public Font InvoiceBodyFont { get; set; }
public Brush InvoiceBodyBrush { get; set; }
public Point LineItemLocation { get; set; }
public RectangleF InvoiceSubTotalLocation { get; set; }
public RectangleF InvoiceTaxLocation { get; set; }
public RectangleF InvoiceGrandTotalLocation { get; set; }
}
```

We have an operational `Invoice` class. It does some things, and they work. But, our `Invoice` class isn't very cohesive. The `Invoice` class has distinct groups of fields. Some are for the state of an invoice and some are for generating a readable invoice. Methods that deal with the behavior and attributes of an invoice don't use the fields that deal with generating a readable invoice.

Our `Invoice` class is implementing two distinct tasks: managing invoice state and generating a readable invoice. The data required to generate a readable invoice (over and above the data shown on an invoice) isn't used by `Invoice` when not generating a readable invoice.

Our `Invoice` class can be said to have multiple responsibilities: the responsibility of managing state and the responsibility of generating a readable invoice. What makes an invoice an invoice may be fairly stable; we may occasionally need to add, remove, or change fields that store data contained in an invoice. But, the act of displaying a readable invoice may be less stable: it may change quite frequently. Worse still, the act of displaying a readable invoice may depend on the platform it is running on.

The Single Responsibility Principle

In terms of focusing refactoring efforts towards cohesiveness of a class, the **Single Responsibility Principle (SRP)** gives us guidance on what a particular class should or should not do. The Single Responsibility Principle states that "there should never be more than one reason for a class to change". In the case of our invoice class there's a couple of reasons why we'd need to change the class:

- The way an invoice is displayed needs to change.
- The data that is contained in an invoice needs to change.

The stability of each responsibility shouldn't need to depend on the other. That is, any change to the `Invoice` class affects stability of the whole class. If I often need to change the way an invoice renders a displayable invoice, all of `Invoice` is unstable – including its responsibility to the data an invoice contains.

The Single Responsibility Principle's focus is fairly narrow: class responsibility. A novice may simply accept that scope and use it only to focus cohesion efforts at the class level. The fact is that the Single Responsibility Principle is applicable at almost all levels of software design, including method, namespace, module/assembly, and process; that is, a method could implement too much responsibility, the types within an assembly or namespace could have unrelated responsibilities, and the responsibilities of a given process might not be focused effectively.

Simply saying "single responsibility" or detailing that something has too many responsibilities is simple. But the actual act of defining what a responsibility is can be very subjective and subtle. Refactoring towards Single Responsibility can take some time and some work to get right. Let's see how we can improve quality through better cohesion and the principle of single responsibility.

Refactoring classes with low-cohesion

Clearly single responsibility (and the heading of this section) suggests that we should refactor `Invoice` into multiple classes. But, how do we do that given our current scenario? The designer of this class has obviously thought that the functionality of `Invoice` included rendering a readable invoice, so how do we make a clean separation? Fortunately, the lack of cohesiveness gives us our separation points. What makes an invoice an invoice doesn't include those fields that deal solely with rendering a readable invoice. Fields like `HeaderFont`, `HeaderBrush`, `HeaderText`, `HeaderLocation`, `FooterFont`, `FooterBrush`, `FooterText`, `FooterLocation`, `InvoiceBodyFont`, `InvoiceBodyBrush`, `LineItemLocation`, `InvoiceBustTotalLocation`, `InvoiceTaxLocation`, and `InvoiceGrandTotalLocation` all deal with just the rendering responsibility.

The `Invoice` class is modeling a real-world invoice. When you hold an invoice in your hand or view it on the screen, it's already rendered. In the real-world we'd never think that a responsibility of rendering an invoice would be a responsibility of the invoice itself.

We know we want to retain our original `Invoice` class and we want to move the rendering responsibility to a new class. This new class will encapsulate the responsibility of rendering an invoice. Since this new class will take an `Invoice` object and help another class produce something useful, we can consider this an invoice rendering service.

In order to refactor our existing `Invoice` class to a new `Invoice` rendering service, we start with a new `InvoiceRenderingService` class and move the `HeaderFont`, `HeaderBrush`, `HeaderText`, `HeaderLocation`, `FooterFont`, `FooterBrush`, `FooterText`, `FooterLocation`, `InvoiceBodyFont`, `InvoiceBodyBrush`, `LineItemLocation`, `InvoiceBustTotalLocation`, `InvoiceTaxLocation`, and `InvoiceGrandTotalLocation` fields to the `InvoiceRenderingService`. Next, we move the `GenerateReadableInvoice` method to the `InvoiceRenderingService`. At this point, we basically have a functional class, but since the `InvoiceRenderingService` method was on the `Invoice` classes, the other properties that the `GenerateReadableInvoice` uses need an `Invoice` object reference—effectively changing it from "this" to a parameter to the `GenerateReadableInvoice` method. Since the original `Invoice` class was never expected to be used externally like this, we need to add a `CalculateGrandTotal` method that delegates to the `invoiceGrandTotalStrategy` object. The result is something like the following:

```
/// <summary>
/// Encapsulates a service to render an invoice
/// to a Graphics device.
/// </summary>
public class InvoiceRenderingService
{
    public void GenerateReadableInvoice(Invoice invoice,
        Graphics graphics)
    {
        graphics.DrawString(HeaderText,
            HeaderFont,
            HeaderBrush,
            HeaderLocation);

        float invoiceSubTotal = 0;
        PointF currentLineItemLocation = LineItemLocation;
        foreach (InvoiceLineItem invoiceLineItem in
            invoice.InvoiceLineItems)
        {
```



```
float lineItemSubTotal =
    Invoice.CalculateLineItemSubTotal(
        invoiceLineItem);

graphics.DrawString(invoiceLineItem.Description,
    InvoiceBodyFont,
    InvoiceBodyBrush,
    currentLineItemLocation);

currentLineItemLocation.Y +=
    InvoiceBodyFont.GetHeight(graphics);
invoiceSubTotal += lineItemSubTotal;
}

float invoiceTotalTax =
    invoice.CalculateInvoiceTotalTax(
        invoiceSubTotal);
float invoiceGrandTotal =
    invoice.CalculateGrandTotal(
        invoiceSubTotal,
        invoiceTotalTax);
Invoice.CalculateInvoiceGrandTotal(invoiceSubTotal,
    invoiceTotalTax);

graphics.DrawString(String.Format(
    "Invoice SubTotal: {0}",
    invoiceGrandTotal - invoiceTotalTax),
    InvoiceBodyFont, InvoiceBodyBrush,
    InvoiceSubTotalLocation);
graphics.DrawString(String.Format("Total Tax: {0}",
    invoiceTotalTax), InvoiceBodyFont,
    InvoiceBodyBrush, InvoiceTaxLocation);
graphics.DrawString(String.Format(
    "Invoice Grand Total: {0}",
    invoiceGrandTotal), InvoiceBodyFont,
    InvoiceBodyBrush, InvoiceGrandTotalLocation);

graphics.DrawString(FooterText,
    FooterFont,
    FooterBrush,
    FooterLocation);
}

public string HeaderText { get; set; }
public Font HeaderFont { get; set; }
public Brush HeaderBrush { get; set; }
public RectangleF HeaderLocation { get; set; }
```

```
public string FooterText { get; set; }
public Font FooterFont { get; set; }
public Brush FooterBrush { get; set; }
public RectangleF FooterLocation { get; set; }
public Font InvoiceBodyFont { get; set; }
public Brush InvoiceBodyBrush { get; set; }
public Point LineItemLocation { get; set; }
public RectangleF InvoiceSubTotalLocation { get; set; }
public RectangleF InvoiceTaxLocation { get; set; }
public RectangleF InvoiceGrandTotalLocation { get; set; }
}
```

Alternatively, the whole use of `invoiceGrandTotalStrategy` can be moved into the `InvoiceRenderingService` – which is a better design decision.

Detecting classes with low-cohesion

So, we've seen a fairly simple example of making a not-so-cohesive class into two more-cohesive classes; but, one of the tricky parts of refactoring away classes with low cohesion is finding them. How do we find classes with low-cohesion?

Fortunately, many people have put time and effort over the years into defining what it means for a class to be cohesive. There have been various metrics researched and created over the years to define cohesion in classes. The most popular metric is **Lack of Cohesion of Methods (LCOM)**. Lack of Cohesion of Methods measures the degree to which all methods use all fields. The more segregated field usage is amongst methods of a class, the higher the Lack of Cohesion of Methods metric of the class will be. Lack of Cohesion of Methods is a measure of the entire class, so it won't point out where the class is not cohesive or indicate where the responsibilities can be separated.

Lack of Cohesion of Methods is a measurement of the degree to which fields are used by all methods of a class. Perfection as defined by Lack of Cohesion of Methods is that every method uses every field in the class. Clearly not every class will do this (and arguably this will hardly ever happen); so, Lack of Cohesion of Methods is a metric, as most metrics are, that requires analysis and thought before attempting to act upon its value. LCOM is a value between 0 and 1, inclusive. A measure of 0 means every method uses every field. The higher the value, the less cohesive the class; the lower the value, the more cohesive the class. A typical acceptable range is 0 to 0.8. But, there's no hard-and-fast definition of specific value that represents *cohesive*; just because, for example, a class has an LCOM value of 0.9, that doesn't mean it can or should be broken up into multiple classes. Lack of Cohesion of Methods values should be used as a method of prioritizing cohesion refactoring work by focusing on classes with higher LCOM values before other classes (with lower LCOM values).

In the case of our Invoice class, it's apparent that its LCOM value does mean it can be split into multiple classes as we detailed in the previous section.

Method cohesion

Methods on their own can also suffer from low-cohesion. One symptom of a method with low-cohesion is size. Methods with low-cohesion are often large. A large method is generally doing more than it needs to. As we saw with the Large Method Code Smell in Chapter 2, refactoring is as simple as breaking the method up into multiple methods. Each method should take on a single responsibility.

Another symptom of a class that suffers from low-cohesion is one that has many parameters. A method that takes many parameters is probably doing too many things. Unfortunately, how to refactor depends on what is trying to be accomplished. One example of too many arguments is often constructors. For example:

```
/// <summary>
/// Example of a class with low-cohesion
/// </summary>
public class Invoice
{
    public string GivenName { get; private set; }
    public string SurName { get; private set; }
    public string Street { get; private set; }
    public string City { get; private set; }
    public string Province { get; private set; }
    public string Country { get; private set; }
    public string PostalCode { get; private set; }

    public Invoice(IEnumerable<InvoiceLineItem>
```

```

        invoiceLineItems, string givenName,
        string surName, string street,
        string city, string province,
        string country, string postalCode,
        Func<float, float, float> calculateGrandTotalCallback)
    {
        GivenName = givenName;
        SurName = surName;
        Street = street;
        City = city;
        Province = province;
        Country = country;
        PostalCode = postalCode;
    }
    //...
}

```

The `Invoice` class has taken on the extra responsibility of managing customer information (given name, surname, street, and so on) and thus one of its constructors has many parameters.

Refactoring methods with low-cohesion

`Invoice` could be refactored to make the constructor (and the class, for that matter) more cohesive by encapsulating customer information into a `Customer` class, encapsulating address information into an `Address` class and the `Invoice` class, and accepting a `Customer` parameter to the constructor that initializes a `Customer` property. The resulting refactoring would look like the following:

```

/// <summary>
/// Customer shape to encapsulate
/// name and address
/// </summary>
public class Customer
{
    public String FirstName { get; private set; }
    public String LastName { get; private set; }
    public Address Address { get; private set; }

    public Customer(string firstName, string lastName,
        Address address)
    {
        FirstName = firstName;
        LastName = lastName;
        Address = address;
    }
}

```

```
    }
  }
  /// <summary>
  /// Address shape to encapsulate
  /// western-style addresses
  /// </summary>
  public class Address
  {
    public string Street { get; private set; }
    public string City { get; private set; }
    public string Province { get; private set; }
    public string Country { get; private set; }
    public string PostalCode { get; private set; }

    public Address(string street, string city, string province,
        string country, string postalCode)
    {
      Street = street;
      City = city;
      Province = province;
      Country = country;
      PostalCode = postalCode;
    }
  }
  /// <summary>
  /// Invoice class that makes use
  /// of <seealso cref="Customer"/>
  /// </summary>
  public class Invoice
  {
    public Customer Customer { get; private set; }
    public Invoice(IEnumerable<InvoiceLineItem>
        invoiceLineItems, Customer customer,
        Func<float, float, float> calculateGrandTotalCallback)
    {
      Customer = customer;
      InvoiceLineItems = new
        List<InvoiceLineItem>(invoiceLineItems);
      this.calculateGrandTotalCallback =
        calculateGrandTotalCallback;
    }
    //...
  }
}
```

We now have an `Invoice` class that makes use of the `Customer` class, and the `Customer` class manages the customer (including address) responsibility.

This provides a much more cohesive constructor and encapsulates customer and address information so that code to initialize that type of information does not have to be repeated. We can use the code that used to be contained in `Invoice` anywhere else with `Customer` without having to repeat that code within the other classes that need it.

In other cases, a method with too many parameters is simply trying to do too many things.

```
/// <summary>
/// Invoice that uses callback for
/// grand total calculation
/// </summary>
public class Invoice
{
    private Func<float, float, float>
        calculateGrandTotalCallback;

    public Invoice(IEnumerable<InvoiceLineItem>
        invoiceLineItems,
        Func<float, float, float> calculateGrandTotalCallback)
    {
        InvoiceLineItems = new
            List<InvoiceLineItem>(invoiceLineItems);
        this.calculateGrandTotalCallback =
            calculateGrandTotalCallback;
    }

    public List<InvoiceLineItem> InvoiceLineItems { get; set; }
    public float TaxRate { get; set; }

    public bool CalculateTotals(out float invoiceSubTotal,
        out float invoiceTotalTax, out float invoiceGrandTotal)
    {
        invoiceSubTotal = 0;
        foreach (InvoiceLineItem invoiceLineItem in
            InvoiceLineItems)
        {
            invoiceSubTotal +=
                (float)((decimal)(invoiceLineItem.Price
                    - invoiceLineItem.Discount)
                    * (decimal)invoiceLineItem.Quantity);
        }
    }
}
```

```
        invoiceTotalTax = (float)((Decimal)invoiceSubTotal *
            (Decimal)TaxRate);
        invoiceGrandTotal =
            calculateGrandTotalCallback(invoiceTotalTax,
                invoiceTotalTax);
        return true;
    }
    //...
}
```

CalculateTotals is a straightforward example that effectively returns three floating-point values from a method: calculating three values and assigning them to output parameters. What is common with methods with "multiple return values" is to return a Boolean result. This Boolean result is always true – and an indication that there's a design issue. In this example, the designer thought it would be useful to be able to query all the total values at the same time. The problem with this is that if only one value is needed, then the caller has to create a couple of dummy variables to needlessly store the unwanted values.

In order to refactor this, we simply need to perform Extract Method refactorings to split up the method into three methods: CalculateInvoiceGrandTotal, CalculateInvoiceTotalTax, and CalculateLineItemSubtotal.

```
/// <summary>
/// Example of method with too many parameters
/// </summary>
public class Invoice
{
    private Func<float, float, float>
        calculateGrandTotalCallback;

    public Invoice(IEnumerable<InvoiceLineItem>
        invoiceLineItems,
        Func<float, float, float> calculateGrandTotalCallback)
    {
        InvoiceLineItems = new
            List<InvoiceLineItem>(invoiceLineItems);
        this.calculateGrandTotalCallback =
            calculateGrandTotalCallback;
    }

    public List<InvoiceLineItem> InvoiceLineItems { get; set; }
    public float TaxRate { get; set; }
    public bool CalculateTotals(out float invoiceSubTotal,
        out float invoiceTotalTax, out float invoiceGrandTotal)
```

```

    {
        invoiceSubTotal = 0;
        foreach (InvoiceLineItem invoiceLineItem in
            InvoiceLineItems)
        {
            invoiceSubTotal +=
                (float)((decimal)(invoiceLineItem.Price
                    - invoiceLineItem.Discount)
                    * (decimal)invoiceLineItem.Quantity);
        }

        invoiceTotalTax = (float)((Decimal)invoiceSubTotal *
            (Decimal)TaxRate);
        invoiceGrandTotal =
            calculateGrandTotalCallback(invoiceTotalTax,
                invoiceTotalTax);
        return true;
    }
    //...
}

```

We've now introduced three methods. `CalculateInvoiceGrandTotal` extracts the portion of code that calculates the invoice grand total and calculates the grand total based on an invoice subtotal and invoice total tax. `CalculateInvoiceTotalTax` extracts the portion of code that calculates the total tax based on an invoice subtotal. `CalculateInvoiceSubTotal` extracts the portion of code that sums the line item subtotals. Each of these methods takes on a single responsibility from the original `CalculateTotals` — which had taken on multiple responsibilities.

Another example is method, which is both a query and a modifier. Sometimes, this is easy to detect, as the word *And* is usually in the method name: `ChangeTaxAndCalculateGrandTotal`. Sometimes, it's harder to detect and requires a bit of analysis of the method body. For example:

```

    /// <summary>
    /// Example Command AND Query method
    /// </summary>
    /// <param name="taxRate"></param>
    /// <returns></returns>
    public float CalculateGrandTotal(float taxRate)
    {
        TaxRate = taxRate;

        float invoiceSubTotal = 0;

        foreach (InvoiceLineItem invoiceLineItem in
            InvoiceLineItems)

```

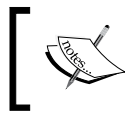


```
{
    invoiceSubTotal +=
        (float)((decimal)(invoiceLineItem.Price
            - invoiceLineItem.Discount)
            * (decimal)invoiceLineItem.Quantity);
}

float invoiceTotalTax = (float)((Decimal)invoiceSubTotal *
    (Decimal)TaxRate);

return calculateGrandTotalCallback(invoiceTotalTax,
    invoiceTotalTax);
}
```

This method first sets the `TaxRate` property, then proceeds to calculate the grand total based on the `TaxRate` property. In the case of this method, it's not clear from its signature (unlike `ChangeTaxAndCalculateGrandTotal`) that it is both a **Command** and a **Query**.



Command: A method that performs an action on an object; generally modifying state.

Query: A method or property that returns data to the caller.

In some circles, a **Command** is also known as a **Modifier**.

This can be refactored by performing the **Separate Query from Modifier** refactoring.

In our case, this is just a matter of removing the `taxRate` parameter from the method and removing the assignment to the `TaxRate` property, similar to the following code:

```
/// <summary>
/// Refactored to Query and not Command
/// </summary>
/// <returns></returns>
public float CalculateGrandTotal()
{
    float invoiceSubTotal = 0;

    foreach (InvoiceLineItem invoiceLineItem in
        InvoiceLineItems)
    {
        invoiceSubTotal +=
            (float)((decimal)(invoiceLineItem.Price
                - invoiceLineItem.Discount)
                * (decimal)invoiceLineItem.Quantity);
    }

    float invoiceTotalTax = (float)((Decimal)invoiceSubTotal *
```

```
(Decimal)TaxRate);  
return calculateGrandTotalCallback(invoiceTotalTax,  
    invoiceTotalTax);  
}
```

Code that needs to calculate the grand total with a different tax rate should simply use the `TaxRate` property before calling `CalculateGrandTotal`. For example, instead of this:

```
float grandTotal = invoice.CalculateGrandTotal(.12f);
```

We'd refactor to this:

```
TaxRate = .12f;  
float grandTotal = invoice.CalculateGrandTotal();
```

In a more complex **Separate Query From Modifier** refactoring, a new method or property would have to be created to separate the **Query**, and the method performing the modification would remove the code that modifies state and is renamed (for example, `ChangeTaxRateAndCalculateGrandTotal` to `CalculateGrandTotal`) to be clear that it isn't modifying state. Calls to the method would have to be changed to the new query method, and a call to the modifier added before the call to the query.

Namespace cohesion

As with any logical grouping of code, what is contained within the grouping may or may not be related. Syntactically, a namespace can contain any number of classes with any number of purposes related to any number of things. Grouping in a namespace is for the programmer; if it doesn't add any value, there's not much point to using it. Classes within a namespace should be related to one another in a particular way that adds value.

Refactoring namespaces with low-cohesion

Unfortunately, there isn't a built-in way to move a class from one namespace to another. You can rename a namespace, but if there is more than one class within the namespace, you "move" *all* the classes to a new or existing namespace.

Let's say we have a class in the `Invoicing` namespace, and we want to move it to the `Invoicing.Domain` namespace because this class represents a fundamental domain entity and locating it in the `Domain` namespace will mean it will be cohesive with the other members of the `Domain` namespace.

```
namespace Invoicing
{
    /// <summary>
    /// Address shape to encapsulate
    /// western-style addresses
    /// </summary>
    public class Address
    {
        public string Street { get; private set; }
        public string City { get; private set; }
        public string Province { get; private set; }
        public string Country { get; private set; }
        public string PostalCode { get; private set; }

        public Address(string street, string city,
            string province, string country, string postalCode)
        {
            Street = street;
            City = city;
            Province = province;
            Country = country;
            PostalCode = postalCode;
        }
    }
}
```

In order to perform a **Move to Another Namespace** refactoring, right-click the namespace name `Invoicing`, and select **Refactor\Rename...** then enter "`Invoicing.Domain`". This effectively "moves" the `Address` class to a new namespace, the `Invoicing.Domain` namespace. This results in the following:

```
namespace Invoicing.Domain
{
    /// <summary>
    /// Address shape to encapsulate
    /// western-style addresses
    /// </summary>
    public class Address
    {
        public string Street { get; private set; }
        public string City { get; private set; }
    }
}
```

```
public string Province { get; private set; }
public string Country { get; private set; }
public string PostalCode { get; private set; }

public Address(string street, string city,
    string province, string country, string postalCode)
{
    Street = street;
    City = city;
    Province = province;
    Country = country;
    PostalCode = postalCode;
}
}
```

The only "heavy lifting" at this point you'll have to do is move the file this class lives in from one directory to another (if you're synchronizing namespace names with directory names). This can be accomplished by dragging and dropping the file in the Solution Explorer.

1. If your namespace has many classes in it and you don't want all the classes to be moved, you'll have to manually perform the move:
2. Use *Find All References* to find all references to *Address*.
3. Change the namespace from *Invoicing* to *Invoicing.Domain* in *Address.cs*.
4. For each entry in the Find Symbol Results, double-click.
5. Add using directive for *Invoicing.Domain*.
6. Optionally move *Address.cs* to another folder with drag/drop in Solution Explorer.

Assembly cohesion

Assembly cohesion can be a bit of a red herring—it diverts attention away from assemblies' inherent features as a deployment strategy. Breaking up a solution into multiple assemblies simply to give related types a cohesive place to live is not the intention here. Assemblies are a deployment strategy; assemblies may need to exist for specific reasons unrelated to cohesion. For example, if two groups of different code need to be executed in separate processes or on separate computers, they need to live in different assemblies regardless of whether their packaging together would be more cohesive.

If you're restructuring a system at the assembly level, you've got some pretty specific needs for what is contained in what assembly. That's not to say you won't have multiple assemblies per deployment platform. You may want to have assemblies that are shared amongst multiple applications, and they may need to be highly cohesive. .NET is a good example of this. .NET 2.0 deployed a certain set of assemblies. .NET 3.0 added types and features, but they're primarily added to assemblies new to .NET 3.0 to avoid changing existing binaries. For example, types in the `System.Data.Linq` namespace could have been included in the `System.Data.dll` assembly and that assembly would have still been cohesive. But, because of the deployment issues (where `System.Data.dll` shouldn't be modified for anything other than show-stopper bugs) added, LINQ types and features were added to a new assembly: `System.Data.Linq.dll`.

Refactoring assemblies

When approaching refactoring assemblies by moving classes from one to another, the best starting point is to have all the projects associated with those assemblies in one Visual Studio® solution. Often, systems already have this type of organization. Larger projects may have to avoid this specific organization for performance and usability reasons within Visual Studio®. A temporary solution that contains these projects, in these cases, is the recommended place to start. When dealing with many project files, this may cause a bit of grief creating and loading the solution, but it will give a huge payoff if you want to move more than a couple of classes.

When approaching performing the Move Type to new Namespace from a single solution, performing the refactoring within Visual Studio® becomes very simple. Moving a class from one project to another becomes a simple process of selecting it in the Solution Explorer and dragging it to another project folder and dropping it while holding the *Shift* key down. Holding the *Shift* key down while dropping causes a move to occur instead of a copy. Once copied, the new file should be edited to change the original namespace to the destination namespace.

Coupling

Coupling is the degree to which two things are related. Coupling and cohesion go hand in hand. Something that is highly-cohesive generally has low coupling. Coupling is a form of dependency.

There are many different types of coupling when we're talking about software design and development. The effort here isn't to make a decoupled design, it's to change the coupling. At some level, one piece of code is coupled to every other piece of code in the system—there is rarely any change you can make to change that. Some code is more highly-coupled to other code and uses the other code directly. Some code is more loosely-coupled and uses other code indirectly. Efforts at refactoring towards a more loosely-coupled design are about the degree to which coupling has been made indirect.

Code can be coupled to other code by a shared data format (external coupling). Code can be coupled to other code by the fact that it results in the execution of other code (control coupling). Code can be coupled to other code by the fact that it results in executing other code by way of an abstract interface (message coupling). Code can be coupled to other code by the fact that they share data, usually in the form of parameters (data coupling). Code can be coupled to other code by the fact that it has a subclass relationship (subclass coupling). Code can also be coupled to other code by that fact that it directly references a type's public interface (content coupling). The direction and degree to which a type is coupled can also help focus our refactoring efforts. Afferent coupling is the degree to which a type is depended upon (inward coupling). Efferent coupling is the degree to which a type depends on other types (outward coupling). High afferent coupling can indicate that a type has too many responsibilities. It's trying to be everything to everyone and thus being used by everyone. High efferent coupling could indicate a type is very dependant. This becomes an issue when the types the class depends upon are in many different assemblies, suggesting a cohesion issue at the assembly layer.

Highly-coupled software is generally accepted to exhibit certain traits, and it can be hard to change. It's like pulling on the thread of a sweater; there are so many dependencies it's impossible to predict how much code will need to be modified in order to make a seemingly simple change. Highly-coupled code is also very rigid. It's hard to move or hard not to duplicate it outside its current context. It carries a lot of baggage (dependencies) with it that need to move with it as a unit. It's one thing to move the code you want to move, it's exponentially harder to move all its dependencies.

While good object-oriented design often promotes high cohesion, loosely coupled design and structure can easily fall to the wayside.

Refactoring subclass coupling

Refactoring subclass coupling involves removing the inheritance relationship between two classes. We discussed Composition over Inheritance in Chapter 5 and detailed how to refactor from inheritance to composition.

Refactoring content coupling

Content coupling is one class directly referencing another. There are several tactics for refactoring away this coupling. Depending on the situation, one tactic may be more applicable than another. One tactic is to use interface-based design and remove the coupling to the content of the class and replace it with coupling to an interface that the other class now implements. Another tactic is to replace method calls into the other class with delegate invocations. A final tactic is to use events instead of direct method calls.

For any particular refactoring, a combination of these tactics may be the best solution. You may find that despite a one-to-one coupling between two classes, it's more appropriate to use a combination of tactics to refactor away the content coupling.

Interface-based design

If you're already coupled to a particular class, replacing use of that class with an interface and having the other class implement that interface is the easiest way to change the coupling between two classes. This reduces coupling from content coupling to a more loosely coupled message coupling.

If the requirements of the other class are very complex or a series of members must come from a single source, using interfaces is often the best solution. Having to hook up several delegates or several events becomes tedious and error prone when a single reference to an object that implements a particular interface is so simple. Imagine if implementing a Windows Form wasn't as simple as deriving from Form and having to register a number of delegates or events.

If you find that implementers of the interface would find default or base implementation for them to be useful, implementing that interface may best be done with an abstract class.

Our `Invoice` class is a good example of something that can be more loosely coupled through interface-based design. It currently implements the calculation of grand totals through interface-based design and the strategy pattern (see Chapter 9). This could have easily been implemented through direct use of a particular class. For example:

```
/// <summary>
/// Service to encapsulate calculation of
/// grand totals.
/// </summary>
public class InvoiceGrandTotalService
{
```

```
    public float CalculateGrandTotal(float invoiceSubTotal,
        float invoiceTotalTax)
    {
        return invoiceSubTotal + invoiceTotalTax;
    }
}

/// <summary>
/// Invoice class that uses
/// <seealso cref="InvoiceGrandTotalService"/>
/// </summary>
public class Invoice
{
    InvoiceGrandTotalService invoiceGrandTotalService =
        new InvoiceGrandTotalService();

    public List<InvoiceLineItem> InvoiceLineItems { get; set; }

    public Invoice(IEnumerable<InvoiceLineItem>
        invoiceLineItems)
    {
        InvoiceLineItems = new
            List<InvoiceLineItem>(invoiceLineItems);
    }

    public float CalculateGrandTotal(float invoiceSubTotal,
        float invoiceTotalTax)
    {
        return invoiceGrandTotalService.CalculateGrandTotal(
            invoiceSubTotal, invoiceTotalTax);
    }
    //...
}
```

In this example, we've created the `InvoiceGrandTotalService` class that contains the `CalculateGrandTotal` method. We then instantiate this class in the `Invoice` class and make reference to it in the `CalculateGrandTotal` method.

We've given away the surprise with this refactoring. We're obviously going to replace direct use of the class with an interface. Since we essentially need a reference to an object right from the start, and to effectively loosen the coupling, we begin refactoring by accepting a reference to an `IInvoiceGrandTotalStrategy` object in the constructor. We then change our `InvoiceGrandTotalService` field to an `IInvoiceGrandTotalStrategy` field and initialize it in the constructor. We finish our refactoring by replacing references from `invoiceGrandTotalService` to `invoiceGrandTotalStrategy`. The resulting refactoring will look similar to the following:

```
/// <summary>
/// Invoice class that uses
/// <seealso cref="IInvoiceGrandTotalStrategy"/>
/// </summary>
public class Invoice
{
    private IInvoiceGrandTotalStrategy
        invoiceGrandTotalStrategy;

    public List<InvoiceLineItem> InvoiceLineItems { get; set; }

    public Invoice(IEnumerable<InvoiceLineItem>
        invoiceLineItems,
        IInvoiceGrandTotalStrategy invoiceGrandTotalStrategy)
    {
        InvoiceLineItems = new
            List<InvoiceLineItem>(invoiceLineItems);
        this.invoiceGrandTotalStrategy =
            invoiceGrandTotalStrategy;
    }

    public float CalculateGrandTotal(float invoiceSubTotal,
        float invoiceTotalTax)
    {
        return invoiceGrandTotalStrategy.CalculateGrandTotal(
            invoiceSubTotal, invoiceTotalTax);
    }
    //...
}
```

If you find that the relationship between the two classes is the invocation of one or two methods that return or update data, you may find that delegates are the best way of refactoring.

Delegates

Loosely coupling to delegates requires that another class (usually the other class) inform the class of what delegates to call. This is generally only useful if there are only one or two delegates. Use of another class that contains many methods that are used by the class becomes problematic to loosen coupling by using delegates simply due to the number of delegate initializations that need to occur.

Use of delegates is generally called callbacks. A callback is something that another class calls in order to obtain values or functionality from an external source. If you find that use of the other class requires values or functionality from the other class, use of callbacks may be an appropriate solution. This is particularly true if this is a comprehensive value or a single functionality.

In our `Invoice` class, we really only have one method that we need to inject into an `Invoice` object. This may be a perfect scenario for a callback.

Refactoring to a callback is much the same as refactoring to interface-based design. In our particular case, we begin by accepting a `Func<float, float, float>` parameter. Then, add a `Func<float, float, float>` field named `calculateGrandTotalCallback` to the `Invoice` class. Next we need to initialize the `calculateGrandTotalCallback` field in the constructor. Finally, we need to replace the call to `CalculateGrandTotal` to an invocation of the `calculateGrandTotalCallback` field. The refactoring should result in something similar to the following:

```
/// <summary>
/// Example of using callback
/// instead of interface
/// </summary>
public class Invoice
{
    private Func<float, float, float>
        calculateGrandTotalCallback;

    public List<InvoiceLineItem> InvoiceLineItems { get; set; }

    public Invoice(IEnumerable<InvoiceLineItem>
        invoiceLineItems,
        Func<float, float, float> calculateGrandTotalCallback)
    {
        InvoiceLineItems = new
            List<InvoiceLineItem>(invoiceLineItems);
        this.calculateGrandTotalCallback =
            calculateGrandTotalCallback;
    }
}
```

```
public float CalculateGrandTotal(float invoiceSubTotal,
    float invoiceTotalTax)
{
    return calculateGrandTotalCallback(
        invoiceSubTotal, invoiceTotalTax);
}
//...
}
```

If you find that you are only passing data to the other class and not receiving any data from it, then events are the best way of refactoring.

Events

Although events are effectively callbacks, they have first-class status in C# and .NET with their own syntax and follow a specific protocol. Events are an optional one-way communication between one class and many subscribers. If you're refactoring from a one-to-one coupling with another class and the use of that other class is to effectively notify it of particular values and not receive any information in return, events are a very apt refactoring.

Events are different from the average use of delegates in one important way: multicasting. This means the standard event interface automatically supports combining event listeners into a multicast delegate under the covers. You can certainly support multicast delegates in our delegate example, but we'd have to expand the interface to support "adding" and "removing" a callback. For example:

```
/// <summary>
/// Invoice class that supports
/// multicast delegates
/// </summary>
public class Invoice
{
    private Func<float, float, float>
        calculateGrandTotalCallback;
    //...

    public void AddCalculateGrandTotalCallback(
        Func<float, float, float> callback)
    {
        calculateGrandTotalCallback += callback;
    }

    public void RemoveCalculateGrandTotalCallback(
        Func<float, float, float> callback)
    {
```

```
        calculateGrandTotalCallback -= callback;
    }
}
```

But, of course, this isn't what callbacks are intended for and thus not applicable for our exemplified purpose. We only get one result from executing multiple delegates with a multicast delegate. For the same reason, multicast delegates aren't suitable for delegates that return values; events that return values are not appropriate. Let's return to a disposable *Invoice* for a moment. Let's say we wanted to inform a subscriber when we are disposed. One way of doing that is with delegates. For example:

```
/// <summary>
/// Invoice that implements IDisposable
/// </summary>
public class Invoice : IDisposable
{
    private bool disposed = false;
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    private void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                //...
            }
            Action callback = disposedCallback;
            if (callback != null)
            {
                callback();
            }
            disposed = true;
        }
    }

    private List<InvoiceLineItem> invoiceLineItems;
    private Action disposedCallback;
```

```
public Invoice(IEnumerable<InvoiceLineItem>
    invoiceLineItems,
    Action disposedCallback)
{
    this.invoiceLineItems = new
        List<InvoiceLineItem>(invoiceLineItems);

    this.disposedCallback =
        disposedCallback;
}
//...
}
```

This does what we want, but it limits us to just one subscriber – the subscriber that created the `Invoice` object. Clearly this isn't the best way to do this, plus it's uncommon and not intuitive. Using events is much more intuitive and supports multiple subscribers. Refactoring to events involves changing the delegate to a public event, removing the initialization of the delegate in the constructor, changing the invocation of the `Invoice` constructor, changing the method used for the delegate to include an object and an `EventArgs` parameter, using this new method to assign an event to the `Invoice` object, and changing the `Dispose` method to use an `EventHandler` object instead. This results in something like the following:

```
/// <summary>
/// Disposable Invoice and Disposed event
/// </summary>
public class Invoice : IDisposable
{
    private bool disposed = false;
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    private void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                //...
            }
            EventHandler handler = Disposed;
            if (handler != null)

```

```
        {
            handler(this, EventArgs.Empty);
        }
        disposed = true;
    }
}

private List<InvoiceLineItem> invoiceLineItems;
public EventHandler Disposed;

public Invoice(IEnumerable<InvoiceLineItem>
    invoiceLineItems)
{
    this.invoiceLineItems = new
        List<InvoiceLineItem>(invoiceLineItems);
}
//...
}
```

Refactoring external coupling

External coupling involves two pieces of code being coupled through the common use of a data format or protocol. Issues arise when coupling to a specific data format if the data format is less stable than the classes that use it. If the data format is less stable and tends to change quite often, you may want to refactor the code to not be directly coupled to the data format.

This coupling can be mitigating by performing the **Introduce Adapter** pattern. The Adapter Pattern converts the less stable data format to a more stable format. It's common that going from a persistent storage source to an in-memory representation of data that a specific data format is used for persistent storage then used to create objects (in-memory representations). Coupling the object-oriented class to this flat data format means the class takes on the responsibility of conversion. This can become out of control if there need to be many different data formats for this object to support many different persistence mechanisms. Data formats often need to change independently of the class(es) that use it due to size and performance requirements, which leads to the domain class having a dependence on instability.

For example, we may need to instantiate an `Invoice` object based on a flat data format such as the following:

```
/// <summary>
/// LineItemData shape for
/// serialization of line item data
/// </summary>
[StructLayout(LayoutKind.Sequential)]
```

```
struct LineItemData
{
    public float Price;
    public float Discount;
    public float Quantity;
    public String Description;
}
/// <summary>
/// InvoiceData shape for
/// serialization of invoice data
/// </summary>
[StructLayout(LayoutKind.Sequential)]
struct InvoiceData
{
    public LineItemData[] LineItemData;
}
public class Invoice
{
    public Invoice(InvoiceData invoiceData)
    {
        InvoiceLineItems = new
            List<InvoiceLineItem>(
                invoiceData.LineItemData.Length);
        foreach (LineItemData lineItemData in
            invoiceData.LineItemData)
        {
            InvoiceLineItems.Add(new InvoiceLineItem()
            {
                Price = lineItemData.Price,
                Discount = lineItemData.Discount,
                Description = lineItemData.Description,
                Quantity = lineItemData.Quantity
            });
        }
    }
    //...
}
```

We have two structures intended to be used for getting data in and out of some sort of flat storage (file system, over the wire, and so on): `LineItemData` and `InvoiceData`. The `Invoice` class populates itself when constructed with an `InvoiceData` instance.

This obviously couples the `Invoice` class directly with `InvoiceData` and indirectly with `LineItemData` and their instability. We can get around this problem by performing the **Introduce Adapter** refactoring.

This refactoring starts with abstracting the `InvoiceData` class by creating an adapter, like `InvoiceDataAdapter`. The `Invoice` class would then be changed to make use of `InvoiceDataAdapter` instead of directly using `InvoiceData`. This refactoring would result in something like the following:

```
/// <summary>
/// Provides a translation of InvoiceData
/// into more appropriate interface
/// </summary>
public class InvoiceDataAdapter
{
    List<InvoiceLineItem> invoiceLineItems;

    public InvoiceDataAdapter(InvoiceData invoiceData)
    {
        invoiceLineItems = new
            List<InvoiceLineItem>(
                invoiceData.LineItemData.Length);
        foreach (LineItemData lineItemData in
            invoiceData.LineItemData)
        {
            invoiceLineItems.Add(new InvoiceLineItem()
            {
                Price = lineItemData.Price,
                Discount = lineItemData.Discount,
                Description = lineItemData.Description,
                Quantity = lineItemData.Quantity
            });
        }
    }

    public IEnumerable<InvoiceLineItem> InvoiceLineItems
    {
        get { return invoiceLineItems; }
    }
}


/// <summary>
/// Invoice class that uses InvoiceDataAdapter
/// </summary>
public class Invoice
{
    public List<InvoiceLineItem> InvoiceLineItems { get; set; }
```



```
public Invoice(InvoiceDataAdapter adapter)
{
    InvoiceLineItems = new
        List<InvoiceLineItem>(adapter.InvoiceLineItems);
}
//...
}
```

Dependency cycles

We can't discuss coupling without discussing **Dependency cycles**. A dependency cycle (otherwise known as a **Circular dependency**) is when an item depends on something else and that something else, or its dependants, depend on that item. Where this presents a problem is at the deployment level. If something in assembly A depends on something in assembly B and something in assembly B depends on something in assembly A, the compiler won't be able to figure out which project to build first and generate an error. Assembly cycles are easy to detect, you've got an error to deal with. The **Acyclic Dependency Principle** details that the dependency structure of **packages** must have no cycles.

 **Package** is a grouping of elements. How the elements are grouped is somewhat subjective, but generally means a physical grouping (for example, assembly in .NET). However, it can be interpreted as any grouping (for example, namespace).

I tend to view packages as all groupings: from assembly to class. The real problem with dependency cycles rears its head at the assembly level, but the assembly structure is dependent on the deployment requirements of the system. The deployment requirements of the system are independent of the logical design of the system. Sometimes, the physical deployment requirements of the system are not known when initial design is begun, and often changes throughout the evolution of a system.

Dependency cycles that don't span assembly boundaries aren't as easy to detect. If class A uses class B and class B uses class A and they're all in the same assembly, there will be no errors. That may or may not be a bad thing. Obviously, it hinders your ability to maintain the code such that you cannot break those two classes out into their own assembly.

Proper dependency design

Dependencies between packages should always occur in one direction. There are various guidelines about what direction makes for the most maintainable code. One guideline deals with stability. The **Stable Dependencies Principle (DSP)** details that a package should only depend on other packages that are more stable than it.

One means of providing stability is abstractiveness. The more abstract something is, the more likely it can be stable. Interfaces, for example, since they only contain a contract and no code, have no possibility that a code change will cause instability with an interface. This assumes that thorough analysis went into the design of the interface. This means the best kind of dependency is a dependency upon an abstraction. The corollary to that is that abstractions should never depend on concrete implementation.

Depending upon abstractions is covered in more detail in Chapter 7.

Another proper dependency design attribute has to deal with layering. Layers can be physical and explicit or logical and implicit. A layer is some sort of abstraction grouping of elements with a common goal. Some common contemporary layers are the **Data Layer** and the **User Interface Layer**. Layers have levels. Some are lower than other layers. The Data Layer, for example, is a lower-level layer compared to the User Interface Layer, which is a higher-level layer. Proper dependency structure between layers is always such that lower-level layers never depend on higher-level layers. The abstraction between layers means a lower-level layer can be used by any number of higher-level layers. Take the User Interface Layer and Data Layer layers for example. I may have multiple User Interface Layers: a WinForm layer, a WPF layer, a Web layer, and so on. If my Data Layer depended on one of the User Interface layers, that User Interface layer would have to be deployed with all other user interface layers. It would be catastrophic if we had to deploy WinForm code with our Web User Interface Layer on a web server, for example.

We'll get more into refactoring as it relates to layers in Chapter 8.

Summary

We've reviewed what it is to be highly cohesive and loosely coupled. We've seen what it means to be non-cohesive and how to detect certain non-cohesiveness through metrics. With some simple refactoring we can make something cohesive where it was previously not.

By refactoring our code to be more cohesive and less coupled, we've improved the maintainability of our code base. Classes are easier to move or reuse and we're less likely to repeat ourselves. Changes are now easier to make because we've decreased the dependencies.

In future chapters, we'll see how we can expand our efforts at loosely coupling even further and discuss some of the benefits and features we can attain once there.

7

Refactoring to Loosely Coupled

In the previous chapter, we started to discuss some techniques for changing the coupling within code and to make its design coupled less to concretions and more to abstractions.

In this chapter, we'll detail aspects of refactoring to a loosely-coupled design, including the following:

- Dependency Inversion principle
- Dependency Injection
- IoC containers
- Factory patterns
- Decorator pattern

The end result of changing these couplings is often just deferring the coupling to another class. This may be good for one class, and bad for the other. There are different types of classes that are more apt at taking on this direct coupling, but many times just moving a direct coupling from one place to another isn't the best solution.

At this point, it's worth providing more detail about what we mean by loosely coupled.

What is loosely coupled?

Loosely coupled has to do with the degree to which one class is not directly coupled to another. To any extent, any two classes within a system will be coupled to some degree. The ideal design means every class is free to evolve without having any affect on any other class in the system. For the most part, this is a fallacy. Modification of anything within a system affects everything else in the system: the system needs to be rebuilt, it needs to be redeployed, and so on.

The degree we're interested in here is the degree to which maintaining each class affects compile-time dependencies on another class. This is important for maintainability.

At worst, compile-time dependencies mean that any change to a class also requires changing other classes – with the compile-time dependencies – to support the change. Compile-time dependencies don't mean *every* change to a class requires supporting changes to other classes with the compile-time changes to the changed class. Changing the body of a method, for example (that is, without changing the method's signature) will not require modifications of other classes in order to continue calling the method.

What are coupling and dependencies?

It's important to take a quick moment to reiterate what we mean when we say coupling and dependency. Dependency is fairly easy: the reliance on something else. Coupling is the degree to which something depends on something else. The key here is the degree. For more detail about detecting the degree to which items are coupled, see the "Detecting highly-coupled". Again, at some level everything in a system depends on everything else in the system, no matter the degree to which they depend on something. The degree of the coupling could be at the source-code level, or it could span process boundaries. The invoicing system depends on the class `Invoice` to function – no level of design magic will ever change that fact. This is a run-time dependency.

Our efforts at loosening the coupling isn't about removing run-time dependencies, it's about changing the types of compile-time dependencies in our system. Effectively, what we're striving for is **Modularity**.



Modularity is the design of a system so that distinct and logical parts are independent of one another until assembled into the system.

This means that independent from one another at design-time; but made dependent at compile-time or run-time.

Clearly, we can never entirely decouple anything from anything else used within a system; but we can design our classes, namespaces, assemblies, modules, and subsystems so they act as components – entirely independent from one-another until told how to work together.

Tightly-coupled

The inverse of loosely-coupled is **tightly-coupled**. Dependencies, as they are commonly referred to, are compile-time dependencies. If one class cannot compile without the existence of a reference to the assembly that contains another, it is dependent on that class. This is **tightly-coupled**.

An example of tightly coupled is code that directly uses a particular class. For example:

```
using (StreamWriter streamWriter = File.CreateText("log.txt"))
{
    streamWriter.Write("log.txt created");
}
```

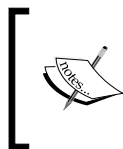
This code creates a `StreamWriter` object that wraps the text file `log.txt` and writes "log.txt created" to that file. It is tightly-coupled to the specific classes `File` and `StreamWriter`.

Before we get too much further, let's introduce and detail some of the principles and methodologies that can be used in our effort to refactor to loosely-coupled.

Dependency Inversion principle

The **Dependency Inversion principle** details that dependencies should be "inverted" from their traditional high-level dependency on low-level implementations. Of course, this doesn't mean that low-level implementations should then depend on high-level implementations – that causes grief in too many ways to mention in this chapter.

What the Dependency Inversion principle does detail is that both low-level and high-level implementations should depend only on abstractions.



High-level modules should not depend on low-level modules. Both should depend on abstractions.
Abstractions should not depend upon details. Details should depend on abstractions.

At face value, in terms of .NET software development and design, this means that one class no longer depends on another class, but depends on an interface or an abstract class.

Inversion of Control

Inversion of Control (IoC) is also known as the **Hollywood Principle**: *Don't call us, we'll call you*. There are many different ways of implementing inversion of control: callbacks, methods, constructor arguments, and so on. The key to understanding Inversion of Control is *don't call us, we'll call you* – inverting the *call us* with *call you* where the inversion comes from the fact that code doesn't call into other code, but other code calls into it.

There are many cases where Inversion of Control is necessary in order to implement the required functionality. In the example of callbacks, external code is *calling back* into your code to pass information or to request functionality. This is often done outside of your process flow (and isn't within your control and hence the inversion). For example:

```
private void InitailizeTimer()
{
    Timer timer = new Timer(OnTimerTick,
        null,
        6000, // start in 6 seconds
        1000); // tick every 1 second afterward
}

private void OnTimerTick(object stateInfo)
{
    // Do something on every timer tick
}
```

In this example, `InitializeTimer` creates a `System.Threading.Timer` object and passes it a delegate to the `OnTimerTick` method. The `Timer` object will then invoke the callback causing `OnTimerTick` to be executed. Due to the fact that `Timer` is controlling when things happen, we can't use procedural code to invoke `OnTimerTick`; we must relinquish control (inverting it) to `Timer` and let it invoke the callback as it sees fit.

For the most part, when we consider Inversion of Control we'll be considering how Inversion of Control facilitates loosely-coupled through **Dependency Injection**.

In terms of facilitating loosely-coupled, Inversion of Control is most easily implemented by means of **Inversion of Control Containers (IoC Containers)** – which we get into greater detail later in the chapter). These containers contain the knowledge (through configuration) of not only how to instantiate specific types, but also have the knowledge of where these instances can be used and how to inject them into other types.

In general, these containers are configured to associate an abstract type with a concrete type. When creating any concrete types that require any other types (abstract or concrete, either through constructor parameters or property setters) the container knows to create those types before attempting to create the other type.

Dependency Injection

Dependency Injection is a more specific type of Inversion of Control. Dependency Injection inverts traditional procedural control of dependencies from direct creational dependency (through invocation of constructors) to an indirect dependency. Instead of dependencies being created directly (*call us*), they are injected into the class that depends on them from an external source (*call you*). When this is coupled with Interface-based Design we can obtain the loosest coupling. This level of modularity allows both the containing class and its dependencies to evolve independently.

By shifting object creation in this way, the *how* and *what* that are instantiated are completely independent of a particular class and can vary completely independently. For example, a pedantic (for the sake of illustrative purposes) invoice rendering service could be implemented as follows:

```
/// <summary>
/// Service to encapsulate rendering of an invoice
/// </summary>
public class InvoiceRenderingService
{
    private Invoice invoice;

    public InvoiceRenderingService(IEnumerable<InvoiceLineItem>
        invoiceLineItems)
    {
        invoice = new Invoice(invoiceLineItems, new
            InvoiceGrandTotalStrategy());
    }

    public void GenerateReadableInvoice(Graphics graphics)
    {
        graphics.DrawString(HeaderText,
            HeaderFont, HeaderBrush, HeaderLocation);
    }
}
```



```
float invoiceSubTotal = 0;
PointF currentLineItemLocation = LineItemLocation;
foreach (InvoiceLineItem invoiceLineItem in
    invoice.InvoiceLineItems)
{
    float lineItemSubTotal =
        Invoice.CalculateLineItemSubTotal(invoiceLineItem);

    graphics.DrawString(invoiceLineItem.Description,
        InvoiceBodyFont, InvoiceBodyBrush,
        currentLineItemLocation);

    currentLineItemLocation.Y +=
        InvoiceBodyFont.GetHeight(graphics);
    invoiceSubTotal += lineItemSubTotal;
}

float invoiceTotalTax =
    invoice.CalculateInvoiceTotalTax(invoiceSubTotal);
float invoiceGrandTotal =
    invoice.CalculateGrandTotal(
        invoiceSubTotal, invoiceTotalTax);
invoice.CalculateInvoiceGrandTotal(invoiceSubTotal,
    invoiceTotalTax);

graphics.DrawString(
    String.Format("Invoice SubTotal: {0}",
        invoiceGrandTotal - invoiceTotalTax),
    InvoiceBodyFont, InvoiceBodyBrush,
    InvoiceSubTotalLocation);
graphics.DrawString(String.Format("Total Tax: {0}",
    invoiceTotalTax), InvoiceBodyFont,
    InvoiceBodyBrush, InvoiceTaxLocation);
graphics.DrawString(
    String.Format("Invoice Grand Total: {0}",
        invoiceGrandTotal), InvoiceBodyFont,
    InvoiceBodyBrush, InvoiceGrandTotalLocation);
graphics.DrawString(FooterText,
    FooterFont, FooterBrush, FooterLocation);
}
//...
}
```

The `InvoiceRenderingService` class creates an `Invoice` object directly based on supplied collection of `InvoiceLineItem` objects during creation. This of course requires that the `InvoiceRenderingService` constructor take on the responsibility of creating an `Invoice` object and directly coupling to the `Invoice` type's constructor.

A more loosely-coupled approach would be that the `InvoiceRenderingService` class constructor accepts an `Invoice` object as a parameter. More loosely coupled still is if the `InvoiceRenderingService` constructor accepted an abstract invoice interface, `IInvoice` as a reference to the invoice object.

This refactoring could be performed as follows:

1. Change the type of the `invoice` field to `IInvoice`.
2. Remove the initialization of the `invoice` field in the constructor.
3. Add an `IInvoice` parameter to the constructor.
4. Initialize the `invoice` field with the value of the `IInvoice` parameter.
5. Change code that invokes the `InvoiceRenderingService` constructor to create an `IInvoice` reference and pass it to the constructor.

This results in an `InvoiceRenderingService` similar to:

```
public class InvoiceRenderingService
{
    private IInvoice invoice;
    public InvoiceRenderingService(IInvoice invoice)
    {
        this.invoice = invoice;
    }
    //...
}
```

The `InvoiceRenderingService` is now loosely-coupled to `Invoice` through sole use of the `IInvoice` interface.

This is an example of Dependency Injection through a constructor parameter. Dependency Injection can also be implemented in any way that allows external code to pass a dependency into another class via a property setter, or method argument as well. A field could also be used, but, public fields are generally considered deprecated.

The general convention is that required dependencies be passed in as constructor arguments and optional dependencies either through property setters or constructor overrides that don't accept those particular dependencies. A required dependency is one where the code requires a valid reference to that dependency in order to function properly. In order to use your class, you know that other code must instantiate your class—or call its constructor. In order to be sure that a valid reference to this dependency exists in an instance of your class with dependency injection, it would be accepted as a parameter in the constructor—checking for null. Optional dependencies are those that don't require a valid reference in order for the class to function properly. The optional dependencies can be accepted as constructor parameters—but null values should be accepted.

What we've attained by this refactoring is essentially moving the code to perform the construction to a different place, the creator of the `InvoiceRenderingService` object. If the use of the `InvoiceRenderingService` object wasn't also inverted, we've essentially just moved the problem somewhere else. This may be entirely acceptable because the creator of the `InvoiceRenderingService` may not be in a package as stable as the package that `InvoiceRenderingService` lives (for example, `InvoiceRenderingService` is a domain object; and something that uses it may be a UI object—something less stable depending on something more stable) which may be a better dependency but possibly not ideal.

Unfortunately, the terminology we're dealing with is somewhat academic. The terms have come from several people that have evolved them over time. While the terms we've discussed so far in this chapter sound very similar, what they mean differs in subtle ways. It's easy to get the three terms Dependency Inversion, Inversion of Control, and Dependency Injection confused or view them much in the same way. To further complicate matters, Dependency Injection is often referred to as DI, which has the same initials as Dependency Inversion.

It helps to remember that Dependency Inversion is a principle and Dependency Injection is a design or implementation detail. That is, you use Dependency Injection as a way of following the Dependency Inversion Principle.

Working with IoC containers

There are a great number of IoC containers for .NET: StructureMap, Autofac, Castle Windsor, Spring.NET, Ninject, Microsoft Unity, and so on. IoC containers are third-party libraries that can be used within your code to better support Dependency Injection. Some have more features than others, and some differ in their API. Comparing and contrasting these containers and detailing their usage could fill its own text; so, we won't get into that level of detail. But, we'll give a short example of using an IoC container with Dependency Injection and loosely-coupled design.

In sticking with the general theme of this book, we'll stick with something from the Microsoft stack: Microsoft Unity. This by no means is to suggest it's better than any of the other existing IoC containers. Suitability is subjective when it comes to IoC containers, what you consider important criteria may, in fact, mean Unity is not the best choice for you.

Here are some criteria to consider when choosing an IoC container:

- Does it support `app.config`-only configuration?
- Is it open-source?
- What are the support options?

Working with an IoC container like Unity is fairly simple. In order to use Unity, specifically, you must first install it. Once installed, add a reference to `Microsoft.Practices.Unity`. There are two basic steps to integrating the container into your code: the first is configuring the container to associate types and the second is to resolve instances of types from the container. In either step, you need to create an instance of the container that you want, to register types and resolve instances. This can be done as follows:

```
UnityContainer unityContainer = new UnityContainer();
```

You can configure the Unity container either in code or in your `app.config` file. We'll only look at various options for configuration in code; but a simple association of a type to an interface is done as follows:

```
unityContainer.RegisterType<Stream, MemoryStream>();
```

This tells Unity that the type `MemoryStream` is associated with a request for `Stream` objects. When Unity is asked for an instance of a `Stream` object, it will create an instance of a `MemoryStream` object. Unfortunately, this isn't enough information for Unity to instantiate a `MemoryStream` object because by default, Unity tries to use the constructor with the largest parameter list. In our case, it has no way to know what to pass for those arguments. We need to tell it which constructor to use. This can be done as follows:

```
unityContainer.RegisterType<Stream, MemoryStream>().  
    Configure<InjectedMembers>().  
    ConfigureInjectionFor<MemoryStream>(  
        new InjectionConstructor());
```

This tells Unity to configure the injected members for `MemoryStream`, so that it should use the constructor with no arguments. Passing an `InjectionConstructor` object to the `ConfigureInjectionFor` method, this tells Unity about the construction injection requirements of this particular registration. By instantiating an `InjectionConstructor` object without passing any arguments, this tells Unity there are no arguments to use for constructor injection so it will use the default constructor for `MemoryStream`.

The next step in integrating the container is to request instances from it. This can be done by invoking the `Resolve` method, as follows:

```
Stream stream = unityContainer.Resolve<Stream>();
```

This asks Unity to resolve an instance of a `Stream` object. What we've configured so far means that Unity will create an instance of `MemoryStream` and return it from `Resolve`.

This is an extremely simple example that doesn't involve any extra dependencies. When dealing with extra dependencies, we need to tell Unity how to resolve those dependencies. For example, consider if we had a loosely-coupled `Invoice` class that was dependant on an `IInvoiceGrandTotalStrategy` interface as follows:

```
public class Invoice : IInvoice
{
    private IInvoiceGrandTotalStrategy
        invoiceGrandTotalStrategy;

    public Invoice(IInvoiceGrandTotalStrategy
        invoiceGrandTotalStrategy)
    {
        this.invoiceGrandTotalStrategy =
            invoiceGrandTotalStrategy;
        //...
    }
    //...
}
```

In order for Unity to construct this `Invoice` object, it would need a type registered for `IInvoiceGrandTotalStrategy`. To do that, we'd do what we did before, plus add another registration:

```
unityContainer.RegisterType<IInvoice, Invoice>().
    RegisterType<IInvoiceGrandTotalStrategy,
        InvoiceGrandTotalStrategy>().
    Configure<InjectedMembers>().
    ConfigureInjectionFor<Invoice>(new
    InjectionConstructor(typeof(IInvoiceGrandTotalStrategy)));
```

This is similar to our previous registration example: we're registering `Invoice` for `IInvoice` resolutions and `InvoiceGrandTotalStrategy` for `IInvoiceGrandTotalStrategy` resolutions and we're telling Unity how to inject dependencies in the constructor. Instead of no constructor arguments, we're telling Unity to use the constructor with the single `IInvoiceGrandTotalStrategy` parameter. Now, when we ask Unity to resolve an `IInvoice` object, it will first instantiate an `InvoiceGrandTotalStrategy` object, then instantiate an `Invoice` object by invoking the constructor with the `InvoiceGrandTotalStrategy` parameter.

The purpose of a loosely-coupled design is not to replace all calls to operator `new` with a request to the container. One common anti-pattern I see is just that; instead of redesigning to use Dependency Inversion, all uses of operator `new` are replaced with calls to `Resolve<T>()` – effectively manually resolving dependencies. This may satisfy Stable Dependencies Principle in that Unity (or whatever container you've chosen) may be more stable than the types Unity is creating, but it's still a needless dependency, as we're still coupling to `Resolve<T>()`. Dependency Inversion relinquishes this dependency and allows the container to automatically resolve dependencies – as we've seen with the `IInvoiceGrandTotalStrategy` example.

Tightly coupled to creation

IoC containers are great, but what about scenarios where code actually does need to create objects? Let's look at another example of some tightly-coupled code:

```
public partial class CreateInvoiceForm : Form
{
    private void okButton_Click(object sender, EventArgs e)
    {
        Invoice invoice =
            new Invoice(invoiceLineItems,
                new InvoiceGrandTotalStrategy());
        StoreInvoice(invoice);
        //...
    }

    private void StoreInvoice(Invoice invoice)
    {
        //...
    }
    //...
}
```

In this example, the `CreateInvoiceForm` class has taken on the responsibility of creating an `Invoice` by directly calling its constructor – tightly-coupling it to the `Invoice` class both by reference and creation.

It's important that we make the distinction that `okButton_Click` both references and creates `Invoice` objects because we can't refactor towards loosely-coupled without addressing both issues. For example, we could refactor towards interface-based design in and derive `Invoice` from `IInvoice` and use `IInvoice` references instead of `Invoice` references.

This would result in `CreateInvoiceForm` refactoring to something like this:

```
public partial class CreateInvoiceForm : Form
{
    private void okButton_Click(object sender, EventArgs e)
    {
        IInvoice invoice =
            new Invoice(invoiceLineItems,
                new InvoiceGrandTotalStrategy());
        StoreInvoice(invoice);
        //...
    }
    //...
}
```

We've replaced references to `Invoice` with `IInvoice` here. This reduces our coupling on `Invoice`, but doesn't reduce it enough to make it useful. We're still coupled to the `Invoice` constructor, so we really haven't accomplished anything useful in terms of making `CreateInvoiceForm` independent from `Invoice`.

Clearly the `CreateInvoiceForm` class needs to be able to create `IInvoice` objects. But how do we do that without directly using a concrete class's constructor? We can do this through the use of an **Abstract Factory**.

Factory Pattern

The **Factory Pattern** is a creational design pattern that takes the creation of an object of a particular type and encapsulates it. This allows other types to relinquish some of the responsibilities of how to create an object. This allows the details of how to create an object to change independently of the code that creates it. There are several types of implementations of the Factory Pattern. The Factory Pattern can be utilized either as a Factory Class or a Factory Method. To be truthful, the Factory Class is just a container for the Factory Method; the differentiation comes in the use of the factory. The use of a factory could be via a delegate (Factory Method) or through an object (Factory Class). The most flexible use of a factory is via factory method (implemented through use of a delegate in .NET).

The simplest implementation is a concrete factory. A concrete factory is a factory that creates a specific type. For example:

```
public class InvoiceFactory
{
    public Invoice CreateInvoice(
        IEnumerable<InvoiceLineItem> invoiceLineItems)
    {
        return new Invoice(invoiceLineItems,
            new InvoiceGrandTotalStrategy());
    }
}
```

This class contains a single method, `CreateInvoice` that returns a reference to a new `Invoice` object. This type of factory encapsulates some of the details of the creation of a particular object, but really just creates an additional dependency (on the factory class) with very little added value. Use of `InvoiceFactory` would still result in a coupling directly to `Invoice`. Added value comes when providing a concrete type through an abstract interface: the **Abstract Factory**.

Abstract Factory

The **Abstract Factory** pattern is a means of creating concretions based on a particular abstraction. These factories can be used as a class, or they can be used as a method. A design that hasn't fully reached being loosely coupled will have to deal directly with the class before reaching a loosely coupled level where they can depend only on a method (delegate). The power of the **Abstract Factory** comes in that it can supply the caller with any number of different implementations of the abstract type – dependent on some optional external criteria. External criteria are implemented as parameters on the method that instantiates the concrete type. The method uses those parameters to decide how to instantiate the concrete type.

The next step to refactoring our `CreateInvoiceForm.okButton_Click` method involves breaking the dependency on the `Invoice` constructor through a factory. We could start out by creating a factory class such as:

```
public class InvoiceFactory
{
    public static IInvoice CreateInvoice(
        IEnumerable<InvoiceLineItem> invoiceLineItems)
    {
        return new Invoice(invoiceLineItems,
            new InvoiceGrandTotalStrategy());
    }
}
```


Then making use of `InvoiceFactory` in `okButton_Click`, such as:

```
public partial class CreateInvoiceForm : Form
{
    private void okButton_Click(object sender, EventArgs e)
    {
        IInvoice invoice =
            InvoiceFactory.CreateInvoice(invoiceLineItems);
        StoreInvoice(invoice);
        //...
    }
    //...
}
```

Now, `CreateInvoiceForm` is componentized in its relation to `Invoice`; they are now both independent from one another.

`CreateInvoiceForm` is now loosely-coupled with regard to `Invoice`; but now it's tightly-coupled to `InvoiceFactory`. We've mitigated the coupling somewhat by using a static `InvoiceFactory` class; clients of `InvoiceFactory` don't carry the burden of creation. Again, this is a slightly better dependency because the `InvoiceFactory` interface is going to be more stable than both `Invoice` and `CreateInvoiceForm`. But, even this coupling could be reduced.

We could continue much in the same vane as `Invoice` to `Invoice` and creating a factory; but pretty soon we'll have factories for factories and we won't be able to stop. No, this is not a better situation.

A solution to this is to use the **Factory Method** variant of the Factory Pattern. The `CreateInvoiceForm` could be given a delegate to a method that creates `IInvoice` objects and the `CreateInvoiceForm` becomes coupled only to some very abstract delegate that could point to a static method or an instance method.

To refactor this, the following should be performed:

- Create a `Factory<T, TResult>` delegate
- Add a `Factory<IEnumerable<InvoiceLineItem>, IInvoice>` field to the `CreateInvoiceForm` class
- Add a constructor to the `CreateInvoiceForm` class that accepts a `Factory<IEnumerable<InvoiceLineItem>, IInvoice>` delegate.
- Add initialization of the factory field
- Replace the calls to `InvoiceFactory.CreateInvoice` with a call to the factory delegate

This would result in something similar to the following:

```
/// <summary>
/// Sample form for creating an invoice
/// </summary>
public partial class CreateInvoiceForm : Form
{
    CreateInvoiceForm(
        Factory<IEnumerable<InvoiceLineItem>, IInvoice>
            invoiceFactory)
    {
        this.invoiceFactory = invoiceFactory;
    }
    private Factory<IEnumerable<InvoiceLineItem>, IInvoice>
        invoiceFactory;
    private void okButton_Click(object sender, EventArgs e)
    {
        IInvoice invoice = invoiceFactory(invoiceLineItems);
        StoreInvoice(invoice);
        //...
    }
    //...
}
```

Decorator pattern

The decorator pattern is another implementation of inversion of control. Rather than the traditional procedural technique of specifying all the steps that need to be performed within a block of code, the Decorator Pattern implements a particular interface or abstract class, wrapping another implementation of that interface and then "decorates" it with additional behavior.

Now that we've discussed Interface-based Design and Dependency Injection, the true power of patterns like Decorator starts to become apparent. The Decorator pattern is quite a hurdle to succumb in a highly-coupled code base. It requires many code changes that could introduce regression bugs, with very little value in comparison. Decorators are generally transient changes to a code base – something done temporarily to accomplish a specific goal.

For example:

```
/// <summary>
/// Logging Invoice Factory Decorator
/// </summary>
```

```
public class LoggingInvoiceFactory : IInvoiceFactory
{
    private IInvoiceFactory invoiceFactory;
    public LoggingInvoiceFactory(IInvoiceFactory
        invoiceFactory)
    {
        this.invoiceFactory = invoiceFactory;
    }
    public Domain.IInvoice CreateInvoice(
        System.Collections.Generic.IEnumerable<
            Domain.InvoiceLineItem>
            invoiceLineItems)
    {
        Trace.WriteLine("Creating invoice");
        return invoiceFactory.CreateInvoice(invoiceLineItems);
    }
}
```

This `LoggingInvoiceFactory` implements `IInvoiceFactory` to "wrap" another `IInvoiceFactory` implementer. It essentially logs the calls to it by invoking `Trace.WriteLine`. This class can be substituted anywhere a `IInvoiceFactory` object is required, adding functionality and decorating it with logging abilities.

At some point, the equivalent of the following code would need to be invoked in order to use the `LoggingInvoiceFactory`:

```
IInvoiceFactory invoiceFactory =
    new LoggingInvoiceFactory(new InvoiceFactory());
```

This code creates an `InvoiceFactory` object and then creates a `LoggingInvoiceFactory` object passing the `InvoiceFactory` object to the constructor so that `LoggingInvoiceFactory` may decorate it with logging functionality.

And a quick review of `okButton_Click` shows how the use of `LoggingInvoiceFactory` is completely transparent:

```
private void okButton_Click(object sender, EventArgs e)
{
    IInvoice invoice =
        invoiceFactory(invoiceLineItems);
    StoreInvoice(invoice);
    //...
}
```

Decorators are very powerful in a loosely coupled design. Because of being loosely-coupled, decorators can be chained and still be used by code that uses the abstract interface. For example, we may decide that we not only want to log calls to the factory, but should also keep statistics about how many objects are created. Rather than include this functionality in `LoggingInvoiceFactory` and risk taking on too much responsibility, we could create yet another decorator as follows:

```
public class StatisticalInvoiceFactory : IInvoiceFactory
{
    private IInvoiceFactory invoiceFactory;
    public uint CreatedObjectCount { get; private set; }

    public StatisticalInvoiceFactory(IInvoiceFactory
        invoiceFactory)
    {
        this.invoiceFactory = invoiceFactory;
    }

    public Domain.IInvoice CreateInvoice(
        System.Collections.Generic.IEnumerable<
            Domain.InvoiceLineItem>
            invoiceLineItems)
    {
        CreatedObjectCount += 1;
        return invoiceFactory.CreateInvoice(invoiceLineItems);
    }
}
```

This second decorator wraps another `IInvoiceFactory` object and counts how many times `CreateInvoice` is called before delegating it on to the contained `IInvoiceFactory` object. If this were to be used with the `LoggingInvoiceFactory`, the following code would have to be invoked before passing `IInvoiceFactory` or its `CreateInvoice` method on to another class/method to use it:

```
IInvoiceFactory invoiceFactory =
    new LoggingInvoiceFactory(new StatisticalInvoiceFactory(
        new InvoiceFactory()));
```

This code creates an `InvoiceFactory` object which is used as a constructor argument to a new `StatisticalInvoiceFactory` object, which is in turn used as a constructor argument to a new `LoggingInvoiceFactory`.

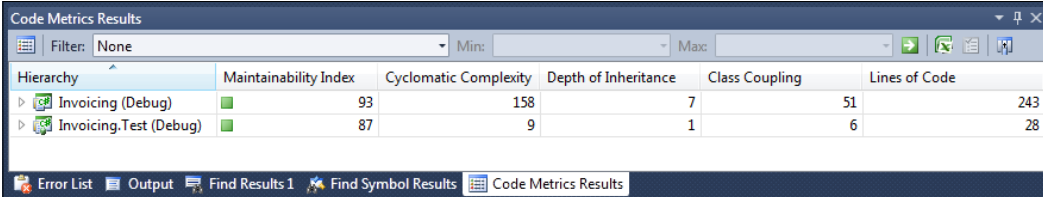
Use of the resulting `IInvoiceFactory` object would result in the `LoggingInvoiceFactory.CreateInvoice` being called, which calls `StatisticalInvoiceFactory.CreateInvoice`, which would then call `InvoiceFactory.CreateInvoice` to create an `IInvoice` object.

As we can see from this example, decorators are an excellent means of implementing single responsibility and separation of concerns, but without a loosely-coupled implementation implementing them would also break separation of concerns and single responsibility.

Detecting highly-coupled

Visual Studio® 2010 Premium and Ultimate include the Code Metrics feature, which includes the Class Coupling metric. Code Metrics can be calculated by right-clicking a project in the Solution Explorer and selecting **Calculate Code Metrics**.

An example output of the Code Metrics can be seen as follows:



Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
▶ Invoicing (Debug)	93	158	7	51	243
▶ Invoicing.Test (Debug)	87	9	1	6	28

The Class Coupling metric measures the number of unique classes referenced by a particular project (assembly), namespace, class, or method. Focus on metric delta, not a line in the sand. Creating a fixed metric goal and striving at all costs to attain that goal rarely adds value. Developers will focus on very small improvements over very big improvements. Setting a Class Coupling goal for classes of 2, for example, across the board means developers are forced to strive for that goal. This would mean any class should reference no more than two other classes. A value of 2 is too arbitrary to be of any use. Improving from 3 to 2, for example, is an improvement of only a single point. It's very difficult to define the benefit attained by a single point decrease in Class Coupling for a particular class. It's also hard to justify the cost of this single point decrease. Focus your loosely-coupled refactoring efforts with classes with the highest Class Coupling. If large changes in Class Coupling are not obvious or too much work; focus the refactoring effort on other aspects that offer better gains.

Other metrics included in Code Metrics are Maintainability Index: a representation of the relative ease of maintaining the code, *Cyclomatic Complexity*: a measure of the structural complexity of the code, *Depth of Inheritance*: a measure of the number of class definitions that extend to the root of the class hierarchy, and *Lines of Code*: a measure of the IL lines of code. The built-in Code Metrics of Visual Studio® offers a very high-level view of your code base. The metrics included are a very small subset of the possible metrics that can be gathered for any particular code base. The built-in metrics represents (arguably subjective) some of the more important metrics. As your code base grows or as the management of that code base becomes more intense, you may find that you will need to gather more in-depth metrics about your code base.

But, as with focusing on the minutia of the Visual Studio® Class Coupling metrics, additional metrics offer the ability to be over-focused on metrics. The project can come to a virtual stand-still while the minutia of these metrics are analyzed and what to do about the metrics and how to put them in action becomes too much of a focus for the team. Beware of this analysis paralysis and ensure you're balancing the value attained by these analysis efforts.

With that in mind, there are third-party tools and applications that can adjunct Visual Studio®'s Code Metrics features. Many of these third party software metrics tools provide much more detail in the coupling metrics they provide. One such tool is NDepend. NDepend was originally a tool to analyze and report on the dependencies of a particular code base; NDepend has branched out into one of the most comprehensive metrics gathering tools available to Visual Studio® users.

Types of coupling

As you get more into reducing coupling, you may find that you may want to focus your refactoring efforts to the type of coupling. As with almost all metrics, they have to be taken with a grain of salt and prioritized. Most details about the couplings within your code base can only help in the prioritization of your refactoring efforts based on coupling.

Afferent coupling

The number of dependencies something has on other types within the package is afferent coupling. This inward coupling metric is a gauge of the responsibility taken on by the types within an assembly, namespace, type, or method. A high afferent coupling maybe a good thing for a particular package and suggest very good cohesion—in that most of the types in the package share the same responsibility.

Efferent coupling

The number of dependencies to other types is efferent coupling. This outward coupling is a gauge of how independent the assembly, namespace, type, or method is. Something that has a high number of external dependencies suggests it is not independent. Something with a high number of external dependencies is said to have the Feature Envy Code Smell and should most likely be subsumed into the package that it depends upon most.

Interface segregation principle

With the power of Interface-based design, Dependency Injection, and Inversion of Control containers; there's a propensity for the Extract Interface refactoring to be invoked on many classes in a code base and those classes used solely through the new extracted interface. This takes a giant leap towards loosely coupling, but can create new, unnecessary couplings.

Each class has its own interface that is used (hopefully) by one or more other classes. Each class may use that class in a very specific way, that is, it may not use its entire interface. By simply extracting an interface from a class and forcing all users of the class to be coupled to that one interface, you're forcing all those classes to be coupled to all of those usages. By providing one interface for all these client usages, it's suggesting one implementation (class) for the one interface. At the very least, each implementation must certainly implement the entire interface – regardless of whether a particular client usage of that interface uses the entire interface.

Enter the **Interface Segregation Principle**.



Interface Segregation Principle: Clients should not be forced to depend on interfaces they do not use.

What the Interface Segregation Principle suggests is that these single interfaces should be broken up into multiple interfaces. The perfect level of granularity would be on interface per client, but that's an ideal that can become hard to maintain. There's a sweet spot between a single interface and one interface per client that will apply to your context or project.

For example, let's posit that the Person class in our system was created to subsume Customer and support users of our system. This class would look like the following:

```
/// <summary>
/// Person class that can be used as
/// a customer or a user
```

```
/// </summary>
public class Person : IUser, ICustomer
{
    private List<Contact> contacts = new List<Contact>();
    public Person(DateTime creationDate,
        String accountNumber)
    {
        AccountNumber = accountNumber;
        CreationDate = creationDate;
        Contacts = contacts;
    }
    public String AccountNumber { get; private set; }
    public IEnumerable<Contact> Contacts { get; private set; }
    public virtual void AddContact(Contact contact)
    {
        if (Inactive)
        {
            throw new InvalidOperationException("Cannot add"
                + " contacts from an Inactive customer.");
        }
        contacts.Add(contact);
    }
    public virtual void RemoveContact(Contact contact)
    {
        if (Inactive)
        {
            throw new InvalidOperationException("Cannot remove"
                + " contacts from an Inactive customer.");
        }
        contacts.Remove(contact);
    }
    public DateTime CreationDate { get; private set; }
    public String Description { get; private set; }
    public virtual void DescribeCustomer(String description)
    {
        if (Inactive)
        {
            throw new InvalidOperationException("Cannot change"
                + " description of an Inactive customer.");
        }
        Description = description;
    }
}
```



```
public String Comments { get; private set; }
public virtual void CommentOnCustomer(String comment)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot add"
            + " comment top an Inactive customer.");
    }
    Comments = comment;
}
public bool Inactive { get; private set; }
public void Deactivate()
{
    Inactive = true;
}
public void Reactivate()
{
    Inactive = false;
}
public string UserName { get; private set; }
public string Name { get; set; }
public string Password {get; set;}
public void UpdateUser()
{
    // write user to physical storage...
}
}
```

Where members `AccountNumber`, `UserName`, `Name`, `CreationDate`, `Password`, and `UpdateUser` are used for users and `AccountNumber`, `Name`, `AddContact`, `CommentOnCustomer`, `Comments`, `Contacts`, `CreationDate`, `Deactivate`, `DescribeCustomer`, `Description`, `Inactive`, `Reactivate`, `RemoveContact` are used for customers. (Note that `AccountNumber` and `CreationDate` are shared between the two).

Clients that use `Person` in the user context shouldn't care about the other customer members, and vice versa for clients that use `Person` in the customer context. The fact is these clients shouldn't care about the `Person` implementation detail. They should be free to concentrate simply on either their user or customer concern, allowing them and the implementation details of a user and a customer to evolve freely and independently.

Interface segregation would suggest that user clients of `Person` should reference it through one interface, and customer clients of `Person` should reference it through a different interface. Should the implementation details of `Person` change—for whatever reason—the clients would be none the wiser and simply continue as if nothing happened. To refactor to this interface segregation, we can perform these actions:

- Create an `IUser` interface
- Create an `ICustomer` interface
- Derive `Person` from both `IUser` and `ICustomer`
- Replace referent of `Person` with either `IUser` or `ICustomer`

This would result in code similar to the following:

```
public interface IUser
{
    string AccountNumber { get; }
    string UserName { get; }
    string Name { get; set; }
    DateTime CreationDate { get; }
    string Password { get; set; }
    void UpdateUser();
}

public interface ICustomer
{
    string AccountNumber { get; }
    string Name { get; set; }
    void AddContact(Contact contact);
    void CommentOnCustomer(string comment);
    string Comments { get; }
    System.Collections.Generic.IEnumerable<Contact> Contacts
        { get; }
    DateTime CreationDate { get; }
    void Deactivate();
    void DescribeCustomer(string description);
    string Description { get; }
    bool Inactive { get; }
    void Reactivate();
    void RemoveContact(Contact contact);
}

public class Person : IUser, ICustomer
{
    //...
}
```

We now have an `IUser` interface that contains the members `AccountNumber`, `UserName`, `Name`, `CreationDate`, `Password`, and `UpdateUser` and an `ICustomer` interface that contains members `AccountNumber`, `Name`, `AddContact`, `CommentOnCustomer`, `Comments`, `Contacts`, `CreationDate`, `Deactivate`, `DescribeCustomer`, `Description`, `Inactive`, `Reactivate`, `RemoveContact` and the `Person` class now implements both `IUser` and `ICustomer`.

If at some point in the future, the design called for the `Person` class to be implemented in two classes instead of one, user clients and customer clients would not have to be refactored. For example:

```
public class User : IUser
{
    public User(DateTime creationDate,
                String accountNumber)
    {
        AccountNumber = accountNumber;
        CreationDate = creationDate;
    }
    public String AccountNumber { get; private set; }
    public DateTime CreationDate { get; private set; }
    public string UserName { get; private set; }
    public string Name { get; set; }
    public string Password {get; set;}
    public void UpdateUser()
    {
        // write user to physical storage...
    }
}

public class Customer : ICustomer
{
    private List<Contact> contacts = new List<Contact>();
    public Customer(DateTime creationDate,
                    String accountNumber)
    {
        AccountNumber = accountNumber;
        CreationDate = creationDate;
        Contacts = contacts;
    }
    public String AccountNumber { get; private set; }
    public IEnumerable<Contact> Contacts { get; private set; } }
```

```
public virtual void AddContact(Contact contact)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot add"
            + " contacts from an Inactive customer.");
    }
    contacts.Add(contact);
}

public virtual void RemoveContact(Contact contact)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot remove"
            + " contacts from an Inactive customer.");
    }
    contacts.Remove(contact);
}

public DateTime CreationDate { get; private set; }
public String Description { get; private set; }
public virtual void DescribeCustomer(String description)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot change"
            + " description of an Inactive customer.");
    }
    Description = description;
}

public String Comments { get; private set; }
public virtual void CommentOnCustomer(String comment)
{
    if (Inactive)
    {
        throw new InvalidOperationException("Cannot add"
            + " comment top an Inactive customer.");
    }
    Comments = comment;
}

public bool Inactive { get; private set; }
public void Deactivate()
```

```
{
    Inactive = true;
}

public void Reactivate()
{
    Inactive = false;
}

public string Name { get; set; }
public string Password { get; set; }
public void UpdateUser()
{
    // write user to physical storage...
}
}
```

We now have a `Customer` class and a `User` class that implement `ICustomer` and `IUser` respectively. All the same functionality that `Person` implemented is still there, and clients of `IUser` and `ICustomer` would not need to change.

Drawbacks of loosely-coupled

Loosely-coupled offers some ease of maintainability through the mere fact that we can reduce our dependencies and change dependencies from instable concretions to more stable abstractions. But, this design does have some drawbacks.

Despite the maintainability gains we get from the more stable dependencies and the flexibility we get from the lack of being tightly-coupled, it can be difficult to understand what components depend on what. For example, since `InvoiceRenderingService` doesn't directly use `Invoice` at all, but uses `IInvoice`, readers of the code must assume that `InvoiceRenderingService` actually uses `Invoice` and thus depends upon it (at least with production code).

Other methods of loose-coupling

In this chapter, we've focused on loosely-coupling code within a single compile unit—that is code that has access to any other code in the code base. The drawback of this is that despite defining abstract interfaces for code to interact with other code, there's nothing stopping a programmer from accessing that code directly and becoming tightly-coupled once again.

Other methods of implementing loose-coupled go beyond the compile-unit boundary and involve process boundaries. A designer might think that separation at the assembly level could help enforce loose-coupling, but the classes that should be internal (and thus hidden) still need to be created by the other assembly and segregation becomes very difficult. It doesn't stop the developer from simply making the class in question public instead of internal and tight-coupling returns.

Web services

Publishing an interface via a web service limits the client of that interface to just what is published. Unlike an assembly reference, there's no means to get at the implementation details of the web service. The web service only makes available what it wants or needs clients to have access to.

Web services use network communications to communicate, as such drastically limiting how the two sides can communicate and what they communicate. Implementing a web service implements a loosely-coupled design because it is coupled only to the interface defined by the web service, so there's no way for the client to become tightly coupled to any of the code that implements the web service. This is very similar to using Interface-based design but we don't directly instantiate any implementation details – everything is dealt with through the web service interface only.

Implementing a web service is a response to very specific requirements and using web services as a means of implementing and enforcing loosely-coupled design is not recommended because of the drawbacks. Obviously communications is not in-memory and is via network communications, so performance is much slower. Lack of in-memory communications means data needs to be marshalled from one side to the other. Generated proxies do most of the marshalling heavy-lifting; but this process is often the conversion from binary data to text data and back again – having a further affect on performance.

Communication hosts

Web services are generally something that is deployed on a completely different computer – usually a web server. There are other techniques of maintaining process-level boundaries between components like communication hosts. Communication hosts are separate processes that can be communicated with through specific technologies. These technologies include sockets or pipes. Sockets are also network communications methods but they are easy to deploy and easy to create and support communication within the same computer. Pipes are similar to sockets but can be scoped to within the same computer and offer the same ease of deployment as sockets.

These non-compile-unit designs with loosely-coupled side-effects are outside the scope of this book, and refactoring to such designs is not included.

Summary

In this chapter, we discussed what side-effects arise from being tightly coupled and how it affects maintainability. With some refactorings specific to attaining a loosely-coupled design, we saw how we can obtain an optimal level of flexibility – a flexibility that increases maintainability by reaching componentization.

In the next chapter, we'll begin the section on refactoring architecture. We'll discuss refactoring to layers, what benefits we'll get from layers, what that means for testability, and detail the Model, View, Presenter pattern.

8

Refactoring to Layers

In the previous chapter, we detailed refactoring to a loosely-coupled design by having classes communicate between each other by means of an abstract interface. We want to refactor to loosely-coupled in order to componentize regions of our software system and be able to modify or enhance a component and minimize the effects on other components.

In this chapter, we take the concept of decoupling to the next level and delve into layers. This journey will include the following destinations:

- What are layers?
- Model View Presenter pattern
- Business Logic and Domain Layers
- Data layers
- Plain Old CLR Objects (POCOs)
- Repository pattern

Layers

So far we've concentrated on abstractions at low-level physical boundaries. We've seen how to refactor classes for increased abstraction to decouple classes from one another. This increases independence and componentization and introduces a flexibility and robustness without a change in external behavior of the software system.

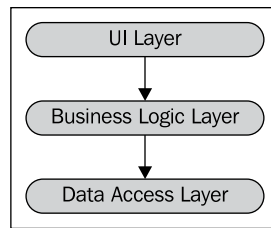
Within a solution in Visual Studio, there are various physical groupings like class (by default, a file), class library (DLL assembly), and application (EXE assembly). Our class-level abstractions deal with this physical boundary. With software design, we can increase the value that we get from the code base by also introducing conceptual or logical boundaries. Layers are a design technique to logically group similar functionality.

Grouping related functionalities into layers has some specific attributes over and above simply conceptually thinking of a group of classes as one particular "layer". A real-world layer is something that is laid atop another layer. Physically one layer can, at most, contact two other layers (the layer above and the layer below); it cannot contact any other layers. Logical layers in software design are similar.

When we create layers and implement them, we organize them from lower-level concepts to higher-level concepts. Logical layers differ slightly from real-world layers in that they "contact" (or take dependencies) only in one direction: downward. In other words, lower-level layers never know about higher-level layers, even though logically they're adjacent. Just as with real-world layers, logical layers only know about adjacent layers (specifically adjacent lower layers).

To recap: a layer is a logical grouping of types with similar external assembly dependencies. Dependencies between layers occur only in one direction: from higher-level layers to adjacent lower-level layers.

Let's look at some typical examples of layering. One common layering design is to have three layers: User Interface Layer, Business Logic Layer, and Data Access Layer. With this common three-layer design, the User Interface Layer is the highest-level layer and interfaces directly with the Business Logic Layer (and not the Data Access Layer) and the Business Logic Layer interfaces directly with the Data Access Layer, which is the lowest level layer. The layers are organized from high-level (more conceptual) to low-level (more specific). The User Interface Layer deals mostly with conceptual details that help the user use the system in a more intuitive way and that are independent of how the system is physically implemented. The Data Access Layer deals specifically with one implementation detail: data persistence. Typically, applications make use of a relational database management system (RDBMS) to store application data. The Data Access Layer encapsulates this implementation detail from the rest of the system. In turn, the Business Logic Layer would encapsulate the Domain details, which would include how those details get translated into requests to the Data Access Layer. The Business Logic Layer would free the User Interface Layer from knowing about and having to deal with the Data Access Layer, and the Data Access Layer would free the Business Logic Layer from having to know about and deal with whatever particular persistence mechanism was chosen. The following diagram shows the associations between layers:



This diagram details that the UI Layer makes use of the Business Logic Layer and the Business Logic Layer makes use of the Data Access Layer. There are some explicit rules of layering that are implied here: the lower-level layers are not dependent on higher-level layers, dependency between layers is in one direction, and layer interdependence is to adjacent layers.

In reality, this 3-layer architecture isn't fine-grained enough. Implementing a 3-layer architecture like this means there are concerns that don't necessarily belong in any of the three layers and have no place to live – often getting shoe-horned into one of the three layers. This often leads to violations of the abstraction-between-layers and unidirectional-dependency rules because of the various concerns in software systems that are used by multiple layers.

Choosing to incorporate layers into a software system can be driven by several motivations. As with our class-level abstractions, we gain componentization and independence. One motivation for refactoring to a layered design may be one of robustness and quality. As we further increase the componentization of a system, we increase its ability to be changed and decrease its fragility. The fragility of a system is often a motivation to refactor to layers. Refactoring to layers can be a precursor to redesigning to tiers – separating functionality in layers to physical boundaries in tiers.

Business logic and domain layers

There are many terms for the same thing in our industry and business logic is no exception. There are a few patterns that refer to data and types specific to the business as the "Model". Other specialties like *Domain-Driven Design* call it the "Domain". Traditionally, in the Microsoft community it has been called "Business Logic". For the sake of consistency when we refer to the Domain, we're effectively also referring to Business Logic or Model.

The Domain contains what is specific to and important to the business. Objects in the business are modeled in our domain so that we may act upon them in a logical and sensible way within our software system. We, the software experts, choose to echo the terminology from our domain experts within our Domain, because we are not the Domain Experts; we do not fully understand the Domain as they do. We use their terminology to keep a one-to-one mapping with their domain to focus on how we model it.

The Domain objects contain logic that has been defined by our Domain Experts with respect to how the object acts. This may include behavior and rules.

At this point, we have a pretty good understanding of our domain, so it may be worthwhile to start our refactoring effort with the Domain Layer.

I find it useful from a development standpoint for each layer to be contained within its own assembly. The User Interface Layer would be a Graphical User Interface application, a website, or something else that provides a user interface. Other layers are implemented as class libraries – an assembly that contains the objects we wish to model.

To begin refactoring, create a new class library. I prefer to have a naming scheme similar to `ProductName.Namespace` (if the class library is to be shared, you may also want to include a company identifier: `Company.ProductName.Namespace`). In our case, we would name the domain class library as `"Invoicing.Domain"`. This sets up the namespace hierarchy nicely right off the bat after project creation.

What we started with is a single application assembly that contained all our domain concepts, data access, and graphical user interface. So, we want to move our domain objects into our new class library. For this particular refactoring we're dealing with the following types: `Invoice` and `InvoiceLineItem`. We can either create new `Invoice` and `InvoiceLineItem` classes in our new class library and copy the body of the classes over, or we can drag-and-drop the classes to the class library in the Solution Explorer – holding *Shift* down to move the files instead of copying them. Drag-moving files in the Solution Explorer copies the files verbatim, so we have to update our namespaces in each file. Simply rename `InvoicingFrontEnd` to `Invoicing.Domain` in both `Invoice.cs` and `InvoiceLineItem.cs`.

Now that we've moved our domain classes to their own class library, we now need to make sure we can access these new classes in our front-end. Of course, in order to do that we need to add a reference to our new class library project to the front-end. We'll go ahead and do that now. Now that we know we can access our domain objects from our Domain class library, we need to make sure that wherever we do, we have the appropriate using directive. Since we no longer have an `Invoice` or `InvoiceLineItem` class in our application project, we can't use the *Find All References* feature to see where they all are and simply click on them one-by-one in the *Find Symbol Results* window. We could go ahead and search for the words "Invoice" and "InvoiceLineItem", but our results could be littered with a bunch of false positives. The easiest method is to simply compile our solution and double-click each error (or press `F8`) and select using `Invoice.Domain` from the smart-tag for each error (`Ctrl+.[Ctrl+period]` or `Shift+Alt+F10`).

Once the solution compiles, run all the unit tests such that they pass and verify that we have not broken anything.

At this point, we've now refactored to a Domain Layer.

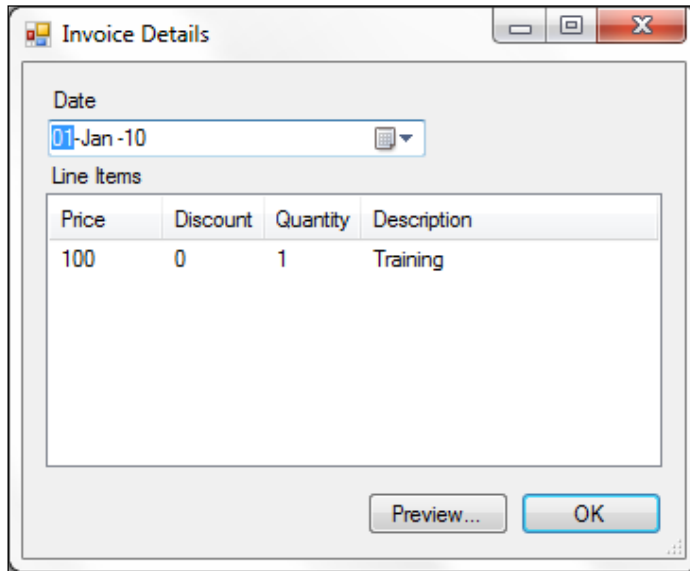
Data Access Layers

A Data Access Layer is a low-level layer whose responsibility is to encapsulate the implementation details of persisting objects to whatever data store we've chosen for our software system. This data store is commonly a **Relational Database Management System (RDBMS)**. Having a Data Access Layer means that the fact that we've chosen a particular RDBMS or that we've even chosen to store data in a RDBMS won't influence code outside the Data Access Layer.

Refactoring UI data access to Data Access Layer

We'll continue our refactoring journey in this chapter from a user interface that includes the data-access code within it.

We have a View Invoice form that accepts the unique ID for a particular invoice, loads it from the database, then displays the aggregate information contained within it. This form, when displaying a particular invoice, looks like this:



There is a **Date** field that shows the date the invoice was produced, and a **Line Items** grid that shows the line items associated to this invoice. There is also a **Preview...** button which provides the ability to preview how the invoice would look, if printed.

The code for the form is as follows:

```
/// <summary>
/// Sample form coupled to data-access
/// </summary>
public partial class ViewInvoiceForm : Form
{
    private Invoice invoice;

    public ViewInvoiceForm(Guid invoiceId)
    {
        using (SqlConnection connection =
            new SqlConnection(
                Properties.Settings.Default.ConnectionString))
        {
            connection.Open();
            DataSet invoiceDataSet = new DataSet("Invoice");
            using (SqlCommand command =
```

```
new SqlCommand("SELECT Id, Date, Title, " +
    "Status FROM Invoice WHERE (Id = @ID)", connection))
{
    command.Parameters.Add("@ID",
        SqlDbType.UniqueIdentifier);
    command.Parameters["@ID"].Value = invoiceId;
    using (SqlDataAdapter invoiceDataAdapter =
        new SqlDataAdapter())
    {
        invoiceDataAdapter.TableMappings.Add(
            "Table", "Invoices");
        command.CommandType = CommandType.Text;
        invoiceDataAdapter.SelectCommand = command;
        invoiceDataAdapter.Fill(invoiceDataSet);
    }
}
DataSet lineItemDataSet =
    new DataSet("LineItems");
using (SqlCommand command =
    new SqlCommand("SELECT InvoiceId, Price, " +
        "Discount, Quantity, Description, " +
        "TaxRate1, TaxRate2 FROM LineItem " +
        "WHERE (InvoiceId = @ID)",
        connection))
{
    command.Parameters.Add("@ID",
        SqlDbType.UniqueIdentifier);
    command.Parameters["@ID"].Value = invoiceId;
    using (SqlDataAdapter lineItemDataAdapter
        = new SqlDataAdapter())
    {
        lineItemDataAdapter.TableMappings.Add(
            "Table", "LineItems");
        command.CommandType = CommandType.Text;
        lineItemDataAdapter.SelectCommand = command;
        lineItemDataAdapter.Fill(lineItemDataSet);
    }
}
List<InvoiceLineItem> lineItems =
    new List<InvoiceLineItem>(
        lineItemDataSet.Tables["LineItems"]
            .Rows.Count);
foreach (DataRow row in
```

```
        lineItemDataSet.Tables["LineItems"].Rows)
    {
        InvoiceLineItem invoiceLineItem =
            new InvoiceLineItem()
            {
                Description = row["Description"] as String,
                Discount = (float)(double)row["Discount"],
                Price = (float)(Decimal)row["Price"],
                Quantity = (int)row["Quantity"],
            };
        if (row["TaxRate1"] != DBNull.Value)
        {
            if ((String)row["TaxRate1"] == "GST")
            {
                invoiceLineItem.TaxRate1 =
                    new FederalGST();
            }
            else if ((String)row["TaxRate1"] == "PST")
            {
                invoiceLineItem.TaxRate1 =new OntarioPST();
            }
        }
        if (row["TaxRate2"] != DBNull.Value)
        {
            if ((String)row["TaxRate2"] == "GST")
            {
                invoiceLineItem.TaxRate2 =
                    new FederalGST();
            }
            else if ((String)row["TaxRate2"] == "PST")
            {
                invoiceLineItem.TaxRate2 =
                    new OntarioPST();
            }
        }
        lineItems.Add(invoiceLineItem);
    }
    DataRow dataRow =
        invoiceDataSet.Tables["Invoices"].Rows[0];
    invoice = new Invoice(
        (String)dataRow["Title"],
        lineItems,
        (DateTime)dataRow["Date"])
    {
```

```
        Id = (Guid)dataRow["Id"]
    };
}
InitializeComponent();
Populate();
}
/// <summary>
/// Fill in all the form's fields
/// </summary>
private void Populate()
{
    dateTimePicker.Value = invoice.Date;
    foreach (var item in invoice.LineItems)
    {
        lineItemsListView.Items.Add(
            new ListViewItem(new string[]
            {
                item.Price.ToString(),
                item.Discount.ToString(),
                item.Quantity.ToString(),
                item.Description }));
    }
}
private void previewButton_Click(object sender,
    EventArgs e)
{
    using (var form =
        InvoicePreviewFormFactory.Create(invoice))
    {
        form.ShowDialog(this);
    }
}
}
```

The constructor is responsible for (in addition to calling `InitializeComponent`) loading the invoice from the database based upon the invoice's ID and invoking the `Populate` method. The `Populate` method flattens the `Invoice` attribute data and populates the controls in the form. The `previewButton_Click` method processes the `Click` event on the `previewButton` button control and displays an `InvoicePreviewForm` form to show a visual representation of what the invoice would look like if printed.

Obviously, the constructor of this form contains lots of code to deal with loading an invoice from the database. As it stands, the implementation of this form works fine, but this data access code introduces some potential issues. It's extremely likely that somewhere else in our invoicing application that the ability of loading an invoice from the database will be required. We could keep it in our View Invoice form and either copy it to other places or write similar code; but, any changes to any of these different blocks of invoice loading code would mean porting those changes to all the other instances of that code. It is not very maintainable. We're also clearly tied to a relational database and that relational database is SQL Server®. If we want to change the RDBMS or switch away from a relational database (or a hybrid approach) we'd have to change our user interface code to accomplish that. Or, if we wanted to implement caching or lazy loading, we would have a hard time tracking down all the data access code throughout the user interface to make that change and make the cache/loading data available throughout the application—destabilizing the UI code for no real value.

So, let's create a Data Access Layer and move the data access code to the Data Access Layer and use the Data Access Layer from the form.

In the data-access code in our `ViewInvoiceForm`, we have some very apparent scenarios: we want to load invoice line item data from the database and we want to load invoice data from the database. This process is effectively to use a `SqlDataAdapter` to execute a `SqlCommand` to populate a `DataSet`.

Just as we did with the Domain Layer, we'll start our Data Access Layer by creating a new class library for it. Using the word "Layer" in my layer components doesn't add any value to me, so I don't generally use the word "Layer" in the name. I simply name it "Data". So, I would add a new class library to our Invoicing solution and name it "Invoicing.Data".

Let's rename the automatically generated `Class1` class to `DataAccess` and we'll begin moving our data access functionality into it. Since we need to load invoice line item data and invoice data, let's add a `LoadInvoiceLineItems` method and a `LoadInvoice` method. This would be something similar to the following:

```
namespace Invoicing.Data
{
    public class DataAccess : IDataAccess
    {
        public DataSet LoadInvoiceLineItems(Guid invoiceId)
        {
        }

        public DataSet LoadInvoice(Guid invoiceId)
        {
        }
    }
}
```

```

    }
  }
}

```

We now have a basic interface that we use to implement data access. We want the other layer to access this layer via an interface, so we've also implemented the following interface:

```

public interface IDataAccess
{
    DataSet LoadInvoice(Guid invoiceId);
    DataSet LoadInvoiceLineItems(Guid invoiceId);
}

```

We can now move the code that fills the DataSet for each type of data into each new method and add connection string injection. For example:

```

/// <summary>
/// Sample IDataAccess implementation supporting
/// invoice and invoice line item data
/// </summary>
public class DataAccess : IDataAccess
{
    private string connectionString;
    public DataAccess(string connectionString)
    {
        this.connectionString = connectionString;
    }
    public DataSet LoadInvoiceLineItems(Guid invoiceId)
    {
        using (SqlConnection connection =
            new SqlConnection(connectionString))
        {
            connection.Open();
            DataSet lineItemDataSet =
                new DataSet("LineItems");
            using (SqlCommand command =
                new SqlCommand("SELECT InvoiceId, Price, " +
                    "Discount,Quantity, Description, " +
                    "TaxRate1, TaxRate2 FROM LineItem " +
                    "WHERE (InvoiceId = @ID)",
                    connection))
            {
                command.Parameters.Add("@ID",
                    SqlDbType.UniqueIdentifier);
                command.Parameters["@ID"].Value = invoiceId;
            }
        }
    }
}

```

```
        using (SqlDataAdapter lineItemDataAdapter
            = new SqlDataAdapter())
        {
            lineItemDataAdapter.TableMappings.Add(
                "Table", "LineItems");

            command.CommandType = CommandType.Text;
            lineItemDataAdapter.SelectCommand = command;

            lineItemDataAdapter.Fill(lineItemDataSet);
        }
    }
    return lineItemDataSet;
}
}

public DataSet LoadInvoice(Guid invoiceId)
{
    using (SqlConnection connection =
        new SqlConnection(connectionString))
    {
        connection.Open();
        DataSet invoiceDataSet = new DataSet("Invoice");
        using (SqlCommand command =
            new SqlCommand("SELECT Id, Date, Title," +
                " Status FROM Invoice WHERE (Id = @ID)",
                connection))
        {
            command.Parameters.Add("@ID",
                SqlDbType.UniqueIdentifier);
            command.Parameters["@ID"].Value = invoiceId;
            using (SqlDataAdapter invoiceDataAdapter =
                new SqlDataAdapter())
            {
                invoiceDataAdapter.TableMappings.Add("Table", "Invoices");
                command.CommandType = CommandType.Text;
                invoiceDataAdapter.SelectCommand = command;

                invoiceDataAdapter.Fill(invoiceDataSet);
            }
        }
    }
    return invoiceDataSet;
}
}
}
```

At this point, we have a Data Access Layer: a layer concerned with how to get data out of a data store. We could now use this Data Access Layer in our form, as follows:

```
/// <summary>
/// Construct ViewInvoiceForm based on
/// <paramref name="invoiceId"/>
/// </summary>
/// <param name="invoiceId"></param>
public ViewInvoiceForm(Guid invoiceId)
{
    IDataAccess dataAccess = new
        DataAccess(
            Properties.Settings.Default.ConnectionString);
    DataSet lineItemDataSet =
        dataAccess.LoadInvoiceLineItems(invoiceId);
    List<InvoiceLineItem> lineItems =
        new List<InvoiceLineItem>(
            lineItemDataSet.Tables["LineItems"]
                .Rows.Count);
    foreach (DataRow row in
        lineItemDataSet.Tables["LineItems"].Rows)
    {
        InvoiceLineItem invoiceLineItem =
            new InvoiceLineItem()
            {
                Description = row["Description"] as String,
                Discount = (float)(double)row["Discount"],
                Price = (float)(Decimal)row["Price"],
                Quantity = (int)row["Quantity"],
            };
        if (row["TaxRate1"] != DBNull.Value)
        {
            if ((String)row["TaxRate1"] == "GST")
            {
                invoiceLineItem.TaxRate1 =
                    new FederalGST();
            }
            else if ((String)row["TaxRate1"] == "PST")
            {
                invoiceLineItem.TaxRate1 =new OntarioPST();
            }
        }
        if (row["TaxRate2"] != DBNull.Value)
        {
```

```
        if ((String)row["TaxRate2"] == "GST")
        {
            invoiceLineItem.TaxRate2 =new FederalGST();
        }
        else if ((String)row["TaxRate2"] == "PST")
        {
            invoiceLineItem.TaxRate2 =new OntarioPST();
        }
    }
    lineItems.Add(invoiceLineItem);
}

DataSet invoiceDataSet = dataAccess.LoadInvoice(invoiceId);
DataRow dataRow =
    invoiceDataSet.Tables["Invoices"].Rows[0];
invoice = new Invoice(
    (String)dataRow["Title"],
    lineItems,
    (DateTime)dataRow["Date"])
{
    Id = (Guid)dataRow["Id"]
};
InitializeComponent();
Populate();
}
```

In the above code, we now create an `IDataAccess` object and make use of its `LoadInvoiceLineItems` and `LoadInvoice` methods instead of directly using `SqlCommand`, `SqlConnection`, and `SqlDataAdapter`. The code to instantiate `InvoiceLineItem` objects and `Invoice` objects continues to remain.

This remaining code is also not really the concern of the user interface and involves quite a bit of business-specific logic about how to construct `InvoiceLineItem` objects and `Invoice` objects from `InvoiceLineItem` objects, and so on. The current code also interfaces directly with the Data Access Layer as well as interfaces with objects in the Domain Layer—clearly violating our layering rules. We can introduce a domain-specific concept called the **Repository Pattern** to provide an abstract collection-like interface that other code can use to *query* and *save* domain data. The repository would know how to interface with the Data Access Layer and take on the business-specific logic of creating various domain objects.

To refactor the remaining code from the `ViewInvoiceForm` class, we simply create an `IInvoiceRepository` interface in our Domain project that contains a `Load` method that takes a `Guid invoiceId` parameter such as the following:

```
namespace Invoicing.Domain
{
    public interface IInvoiceRepository
    {
        Invoice Load(Guid invoiceId);
    }
}
```

We then create an `InvoiceRepository` class that derives from `IInvoiceRepository`, has a constructor that accepts an `IDataAccess` object, a `Load` method that takes a `Guid invoiceId` parameter, and has an `IDataAccess` `dataAccess` field. It should look something like this:

```
namespace Invoicing.Domain
{
    /// <summary>
    /// An in-memory-collection-like
    /// interface to Invoice objects
    /// </summary>
    class InvoiceRepository : IInvoiceRepository
    {
        IDataAccess dataAccess;

        public InvoiceRepository(IDataAccess dataAccess)
        {
            this.dataAccess = dataAccess;
        }

        public Invoice Load(Guid invoiceId)
        {
            DataSet lineItemDataSet =
                dataAccess.LoadInvoiceLineItems(invoiceId);
            List<InvoiceLineItem> lineItems =
                new List<InvoiceLineItem>(
                    lineItemDataSet.Tables["LineItems"]
                    .Rows.Count);
            foreach (DataRow row in
                lineItemDataSet.Tables["LineItems"].Rows)
            {
                InvoiceLineItem invoiceLineItem =new InvoiceLineItem()
                {
                    Description = row["Description"] as String,
                }
            }
        }
    }
}
```

```
        Discount = (float)(double)row["Discount"],
        Price = (float)(Decimal)row["Price"],
        Quantity = (int)row["Quantity"],
    };
    if (row["TaxRate1"] != DBNull.Value)
    {
        if ((String)row["TaxRate1"] == "GST")
        {
            invoiceLineItem.TaxRate1 =new FederalGST();
        }
        else if ((String)row["TaxRate1"] == "PST")
        {
            invoiceLineItem.TaxRate1 =new OntarioPST();
        }
    }
    if (row["TaxRate2"] != DBNull.Value)
    {
        if ((String)row["TaxRate2"] == "GST")
        {
            invoiceLineItem.TaxRate2 =new FederalGST();
        }
        else if ((String)row["TaxRate2"] == "PST")
        {
            invoiceLineItem.TaxRate2 =new OntarioPST();
        }
    }
    lineItems.Add(invoiceLineItem);
}

DataSet invoiceDataSet =
    dataAccess.LoadInvoice(invoiceId);
DataRow dataRow =
    invoiceDataSet.Tables["Invoices"].Rows[0];
return new Invoice(
    (String)dataRow["Title"],
    lineItems,
    (DateTime)dataRow["Date"])
{
    Id = (Guid)dataRow["Id"]
};
}
}
}
```

The code in the `Load` method is effectively the same as what we have just refactored in the form.

To complete refactoring to a Data Access Layer, we simply update the form to make use of an `IInvoiceRepository` object by adding an `IInvoiceRepository` parameter on the constructor and making a call to `IInvoiceRepository.Load`. For example:

```
public ViewInvoiceForm(IInvoiceRepository invoiceRepository,
    Guid invoiceId)
{
    invoice = invoiceRepository.Load(invoiceId);
    InitializeComponent();
    Populate();
}
```

Refactoring domain data access to Data Access Layer

Of course, starting with all the data access code in the user interface code is only one way to start out. The other possibility is that the domain objects contain code to perform data access of themselves. (Okay, there are probably a large finite number of possibilities here; I'll concentrate on these two very common possibilities.) These two possibilities are mutually exclusive for any particular domain object and rather than showing refactoring to a Data Access Layer that contained both possibilities, I've chosen to separate them into two methods of refactoring for clarity.

The other possibility is that the data access code could very easily have been implemented right in the class that needs to be loaded:

```
/// <summary>
/// Sample domain class that is
/// coupled to data-access
/// </summary>
public class Invoice
{
    //...

    public static Invoice Load(Guid invoiceId)
    {
        using (SqlConnection connection =
            new SqlConnection(
                Properties.Settings.Default.ConnectionString))
        {
            connection.Open();
```



```
DataSet invoiceDataSet = new DataSet("Invoice");
using (SqlCommand command =
    new SqlCommand("SELECT Id, Date, Title," +
        " Status FROM Invoice WHERE (Id = @ID)",
        connection))
{
    command.Parameters.Add("@ID", SqlDbType.UniqueIdentifier);
    command.Parameters["@ID"].Value = invoiceId;
    using (SqlDataAdapter invoiceDataAdapter =
        new SqlDataAdapter())
    {
        invoiceDataAdapter.TableMappings.Add("Table", "Invoices");
        command.CommandType = CommandType.Text;
        invoiceDataAdapter.SelectCommand = command;
        invoiceDataAdapter.Fill(invoiceDataSet);
    }
}
DataSet lineItemDataSet =
    new DataSet("LineItems");
using (SqlCommand command =
    new SqlCommand("SELECT InvoiceId, Price," +
        "Discount,Quantity, Description, " +
        "TaxRate1, TaxRate2 FROM LineItem " +
        "WHERE (InvoiceId = @ID)",connection))
{
    command.Parameters.Add("@ID", SqlDbType.UniqueIdentifier);
    command.Parameters["@ID"].Value = invoiceId;
    using (SqlDataAdapter lineItemDataAdapter
        = new SqlDataAdapter())
    {
        lineItemDataAdapter.TableMappings.Add(
            "Table", "LineItems");

        command.CommandType = CommandType.Text;
        lineItemDataAdapter.SelectCommand = command;

        lineItemDataAdapter.Fill(lineItemDataSet);
    }
}
List<InvoiceLineItem> lineItems =
    new List<InvoiceLineItem>(
        lineItemDataSet.Tables["LineItems"].Rows.Count);
foreach (DataRow row in
    lineItemDataSet.Tables["LineItems"].Rows)
{
```

```
InvoiceLineItem invoiceLineItem =
    new InvoiceLineItem()
    {
        Description = row["Description"] as String,
        Discount = (float)(double)row["Discount"],
        Price = (float)(Decimal)row["Price"],
        Quantity = (int)row["Quantity"],
    };
if (row["TaxRate1"] != DBNull.Value)
{
    if ((String)row["TaxRate1"] == "GST")
    {
        invoiceLineItem.TaxRate1 = new FederalGST();
    }
    else if ((String)row["TaxRate1"] == "PST")
    {
        invoiceLineItem.TaxRate1 =new OntarioPST();
    }
}
if (row["TaxRate2"] != DBNull.Value)
{
    if ((String)row["TaxRate2"] == "GST")
    {
        invoiceLineItem.TaxRate2 =new FederalGST();
    }
    else if ((String)row["TaxRate2"] == "PST")
    {
        invoiceLineItem.TaxRate2 =new OntarioPST();
    }
}
lineItems.Add(invoiceLineItem);
}
DataRow dataRow =
    invoiceDataSet.Tables["Invoices"].Rows[0];
return new Invoice(
    (String)dataRow["Title"],lineItems,
    (DateTime)dataRow["Date"])
{
    Id = (Guid)dataRow["Id"]
};
}
}
```

With this version of our `Invoice` class, we have a static `Load` method that performs the same logic that we've seen previously. Obviously, we want to avoid an instance method to load an `Invoice` object because the act of loading the object should create it – we'd have to create an empty object and tell it to load itself, which would not be very intuitive. The configuration of the connection string to connect to the database is abstracted away through the `Properties` object, but now our domain entity is coupled back to the main (UI) assembly. This drastically hinders how we can layer our application. As is, `Invoice` simply can't be put into a Domain Layer because it would introduce a circular dependency. As a static method, we also limit what we can do: inheritance, interfaces, and polymorphism are pretty much out of the window. As it stands, the `Invoice` class is tightly coupled, not only to types in the `System.Data` namespace, but now tightly coupled to SQL Server. We've encapsulated *which* SQL Server database elsewhere (in `Properties`), so changing to a different server won't impact our code; but, if we wanted to change to a different type of data source such as Oracle, MySQL, or Azure Database, we'd have to modify code within the `Invoice` class. Why should changing the database require changes to `Invoice.cs`?

Also, what if we wanted our `Invoice` class to be persisted to two different types of databases? What if we wanted to make our system multi-tier with one tier dealing with `Invoice` objects in SQL Server, and another dealing with `Invoice` objects within an in-memory cache? We'd have to build those two types of persistence sources into `Invoice` to support that. Quickly `Invoice` gets overrun with data access code making it hard to read, hard to maintain, and increasing the potential for error.

Domain entities are representations of real-world concepts in a business. They need to be modeled in various different places in many different ways. They need to be extremely flexible to be re-used in these ways. In order to do that, they need to be entirely decoupled from everything else in the system. This specific aspect of maintaining the Single Responsibility Principle is called Plain Old CLR Objects.

Plain Old CLR Objects

Plain Old CLR Objects (POCO) is a specific design technique of implementing Single Responsibility that specifically separates object persistence from domain objects.

In our example, we've completely written from scratch a means of storing our object to a SQL Server database by means of ADO.NET. All this heavy lifting could have been done by a persistence framework. That framework could have required specific implementation details of our domain objects. POCO stemmed from the Java™ community (which called it POJO) in response to JavaBeans™, which required particular conventions for implementing properties, implementing specific interfaces, and so on.

So, fully refactoring from domain-object-based persistence to a Data Access Layer means making our Domain object a POCO and moving the data access to another layer. Moving the data access out of our `Invoice` object into a Data Access Layer results in the same thing as we've already done, so we won't detail the `DataAccess` and `InvoiceRepository` classes again.

Continuing with the refactoring, the `Load` method is completely removed from `Invoice`. This leaves us with the modifications of where this, now missing, `Invoice.Load` method is used. For example, let's say that our `ViewInvoiceForm` constructor used `Invoice.Load` as follows:

```
public ViewInvoiceForm(IInvoiceRepository invoiceRepository,
    Guid invoiceId)
{
    invoice = Invoice.Load(invoiceId);
    InitializeComponent();
    Populate();
}
```

We would simply refactor that to the same code we ended up with in the previous possibility:

```
public ViewInvoiceForm(IInvoiceRepository invoiceRepository,
    Guid invoiceId)
{
    invoice = invoiceRepository.Load(invoiceId);
    InitializeComponent();
    Populate();
}
```

At this point, we could simply accept what we have left as the User Interface Layer, after all, it is using our model through a Domain Layer and the Domain Layer is using a Data Access Layer – what's left must be a User interface layer.

User interface layers

User interface layers are fairly apparent: they're the layer of the software system whose responsibility it is to interface with the user, usually through a graphic user interface.

What we have left in the user interface is, of course, its own layer; this even satisfies our layering rules it (communicates in one direction with only the adjacent lower layer). But, the user interface code is still tightly coupled to the model and effectively contains business logic code with regard to dealing with the invoice repository, flattening invoice objects, and un-flattening data into invoice objects.

The user interface is the most unstable part of a software system. The user interface is the canvas on which user provides their feedback about the system. The user interface is how the user thinks about the system and how they communicate about the system; in their mind the user interface is the entire system. Almost all changes to a system other than logic defects are changes to how the system behaves: tweaks to the UI.

Business logic, on the other hand, is generally stable. A business really must know how their business operates and the rules by which it operates. The Business Logic layer is a logical representation of the entities within the business, how the business interoperates with those entities, and the rules by which they interoperate. The business logic should be the most stable part of the system, because it's likely to have less need to change. It's in our best interest to keep that stable logic in one component and have other, less stable components depend on it. This way, as unstable components get modified, the modifications can't inadvertently affect stable logic. If we have business logic in our user interface and that user interface is frequently changing, we have a greater chance of inadvertently changing the business logic code and introducing a defect.

We've taken a great step so far to push out much of the business logic from the user interface code and put it into another component: the Domain Layer. But, we can take that a step further to make the user interface entirely decoupled from the Domain Layer. We can do that with the **Model View Presenter** pattern.

Model View Presenter (MVP)

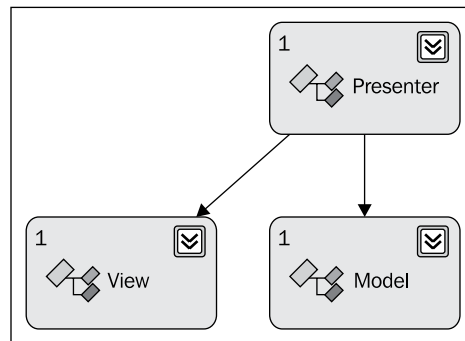
The **Model View Presenter (MVP)** pattern facilitates a layered architecture (but doesn't implement layers on its own). We're unable to put business logic into the appropriate layer if it's tangled up in the UI (View) and not decoupled into an independent class that can live within a distinct layer.

As we've discussed, the Model is a representation of our business entities and logic. It's akin to what we've described as being a responsibility of a Business Logic Layer or, as we've chosen to call it, the Domain Layer.

The View is appropriately the user-interface component of MVP – how the user physically views the system. In a WinForms application, the view implementation is a `Form`.

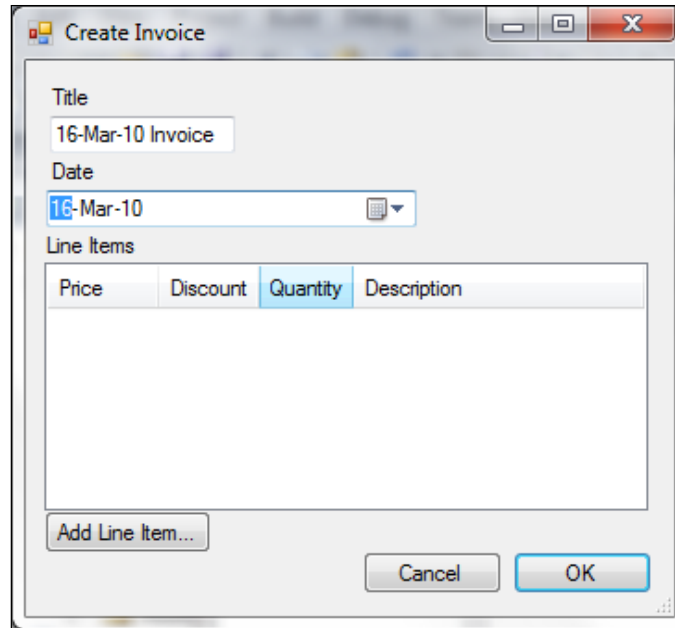
The final component in MVP is the Presenter. There are various flavors of MVP; all have a consistent definition of the Model, but the separation of responsibilities between the View and the Presenter changes. One variant is considered the *supervising controller*, where the Presenter (or controller) takes on only some View/Model synchronization. The other variant is considered the *passive View*, where the View is basically "dumb"; it knows about processing events from the user and the fields within it but delegates populating those fields based on domain entities to the Presenter. A passive View is effectively implemented with Inversion of Control, where it's told to do things (by the Presenter) rather than just doing them for itself. The passive View variant allows us to obtain separation of UI concerns from business logic and results in a more decoupled user interface design. It is this type of MVP implementation we'll detail here.

The following diagram shows the association between the Model, the View, and the Presenter with the MVP pattern:



In reality, the Model is more conceptual in this diagram and the Presenter will actually use specific Model classes rather than a single class. The "Model" is more like a namespace than an actual type. There is an implementation of the View that the Presenter will use, via an interface. The View and the Presenter can live in different layers and must be separated by abstractions (interfaces).

For our MVP refactoring, we'll refactor `CreateInvoiceForm`. The Create Invoice form is used to create and edit invoices. It's very similar to the View Invoice form except it will allow editing of the invoice title, adding line items, and not providing a preview. The Create Invoice form has the following appearance:



The following is the `CreateInvoiceForm` class:

```
/// <summary>
/// Create invoice form coupled to the Model
/// </summary>
public partial class CreateInvoiceForm : Form
{
    public CreateInvoiceForm(Invoice invoiceToEdit)
    {
        InvoiceLineItems = new
            List<InvoiceLineItem>(invoiceToEdit.LineItems);
        Title = invoiceToEdit.Title;
        Date = invoiceToEdit.Date;
        InitializeComponent();
    }

    public CreateInvoiceForm()
    {
        InvoiceLineItems = new List<InvoiceLineItem>();
    }
}
```

```
Title = DateTime.Now.ToString("d") + " Invoice";
Date = DateTime.Now;
InitializeComponent();
}

private void CreateInvoiceForm_Load(object sender,
EventArgs e)
{
    titleTextBox.Text = Title;
    dateTimePicker.Value = Date;
    Populate();
}

private void okButton_Click(object sender, EventArgs e)
{
    DialogResult = DialogResult.OK;
    Close();
}

private void cancelButton_Click(object sender, EventArgs e)
{
    DialogResult = DialogResult.Cancel;
    Close();
}

private void Populate()
{
    lineItemsListView.Items.Clear();
    dateTimePicker.Value = DateTime.Now;
    foreach (var item in InvoiceLineItems)
    {
        lineItemsListView.Items.Add(
            new ListViewItem(new string[] {
                item.Price.ToString(),
                item.Discount.ToString(),
                item.Quantity.ToString(),
                item.Description
            }));
    }
}

private void addLineItemButton_Click(object sender,
EventArgs e)
{
    using (AddLineItemForm form = new AddLineItemForm())
    {
        if (form.ShowDialog(this) == DialogResult.OK)
```



```
        {
            InvoiceLineItems.Add(form.CreateLineItem());
            Populate();
        }
    }
}

public Invoice GetInvoice()
{
    return new Invoice(titleTextBox.Text,
        InvoiceLineItems, dateTimePicker.Value);
}

public ICollection<InvoiceLineItem> InvoiceLineItems
{ get; set; }

public DateTime Date { get; set; }

public string Title { get; set; }
}
```

We first want to decouple the Create Invoice form from `Invoicing.Domain.Invoice`. This is easily done by deleting the constructor that takes the `Invoice` parameter (that responsibility will move to the Presenter, which we'll see in a moment). Next, we want to decouple from `Invoicing.Domain.InvoiceLineItem`. To that effect, we'll create a **Data Transfer Object (DTO)** for the line item data.



Data Transfer Object is a design pattern to encapsulate the data-only aspects of domain data. Commonly referred to as a **DTO**, the objects do not contain behavior and thus do not model business logic or business rules and are meant only to transfer data from one subsystem to another.

The line item DTO is as follows:

```
/// <summary>
/// Invoicing.Domain.InvoiceLineItem data
/// </summary>
public class InvoiceLineItemDTO
{
    public float Quantity { get; set; }
    public float Price { get; set; }
    public float Discount { get; set; }
    public string Description { get; set; }
    public ITaxRate TaxRate1 { get; set; }
}
```

```
public ITaxRate TaxRate2 { get; set; }
public InvoiceLineItemDTO(InvoiceLineItem invoiceLineItem)
{
    Quantity = invoiceLineItem.Quantity;
    Price = invoiceLineItem.Price;
    Discount = invoiceLineItem.Discount;
    Description = invoiceLineItem.Description;
    TaxRate1 = invoiceLineItem.TaxRate1;
    TaxRate2 = invoiceLineItem.TaxRate2;
}
public InvoiceLineItemDTO()
{
}
public InvoiceLineItem ToInvoiceLineItem()
{
    return new InvoiceLineItem()
    {
        Description = Description,
        Discount = Discount,
        Price = Price,
        Quantity = Quantity,
        TaxRate1 = TaxRate1,
        TaxRate2 = TaxRate2
    };
}
}
```

This DTO will allow our view to evolve more independently from the Domain. The next step in the refactoring is to change the `AddLineItemForm` to create the DTO object instead of the domain object. The changed method would be like the following:

```
public InvoiceLineItemDTO CreateLineItemDTO()
{
    return new InvoiceLineItemDTO()
    {
        Quantity = (int)quantityNumericUpDown.Value,
        Description = description,
        Discount = (float)discountNumericUpDown.Value,
        Price = (float)priceNumericUpDown.Value
    };
}
```

The next step in the refactoring is to make use of this new DTO by replacing use of `InvoiceLineItem` with `InvoiceLineItemDTO` in our `CreateInvoiceForm` class. This results in the following:

```
/// <summary>
/// Create invoice form decoupled from the Model
/// </summary>
public partial class CreateInvoiceForm : Form
{
    public CreateInvoiceForm()
    {
        InvoiceLineItemDTOs = new List<InvoiceLineItemDTO>();
        Title = DateTime.Now.ToString("d") + " Invoice";
        Date = DateTime.Now;
        InitializeComponent();
    }

    private void CreateInvoiceForm_Load(object sender,
    EventArgs e)
    {
        titleTextBox.Text = Title;
        dateTimePicker.Value = Date;

        Populate();
    }

    private void okButton_Click(object sender, EventArgs e)
    {
        DialogResult = DialogResult.OK;
        Close();
    }

    private void cancelButton_Click(object sender, EventArgs e)
    {
        DialogResult = DialogResult.Cancel;
        Close();
    }

    private void Populate()
    {
        lineItemsListView.Items.Clear();
        dateTimePicker.Value = DateTime.Now;
        foreach (var item in InvoiceLineItemDTOs)
        {
            lineItemsListView.Items.Add(
                new ListViewItem(new string[] {
                    item.Price.ToString(),
                    item.Discount.ToString(),
                })
            );
        }
    }
}
```

```

        item.Quantity.ToString(),
        item.Description
    }));
    }
}

private void addLineItemButton_Click(object sender,
    EventArgs e)
{
    using (AddLineItemForm form = new AddLineItemForm())
    {
        if (form.ShowDialog(this) == DialogResult.OK)
        {
            InvoiceLineItemDTOs.Add(form.CreateLineItemDTO());
            Populate();
        }
    }
}

public ICollection<InvoiceLineItemDTO> InvoiceLineItemDTOs
    { get; set; }

public DateTime Date { get; set; }

public string Title { get; set; }
}

```

The next step in the refactoring is to support the fact that the Presenter will deal with the View through an abstraction, the `ICreateInvoiceView` interface:

```

public interface ICreateInvoiceView
{
    ICollection<InvoiceLineItemDTO> InvoiceLineItemDTOs
        { get; }
    DateTime Date { get; set; }
    string Title { get; set; }
}

```

To proceed with the refactoring, we can now implement the Presenter. The Presenter will initialize the View and create a domain `Invoice` object from the data contained in the View when the user presses the OK button. The `CreateInvoicePresenter` class is as follows:

```

/// <summary>
/// Encapsulation of Model to View interaction
/// </summary>
class CreateInvoicePresenter
{

```

```
private ICreateInvoiceView view;
public CreateInvoicePresenter(ICreateInvoiceView view)
{
    this.view = view;
}
public void Start()
{
    // TODO: wire up the other data, subscribe to events,
    // etc.
}
public Invoice GetInvoice()
{
    List<InvoiceLineItem> invoiceLineItems =
        new List<InvoiceLineItem>();
    foreach (InvoiceLineItemDTO invoiceLineItemDTO in
        view.InvoiceLineItemDTOS)
    {
        invoiceLineItems.Add(
            invoiceLineItemDTO.ToInvoiceLineItem());
    }
    return new Invoice(view.Title, invoiceLineItems, view.Date);
}
}
```

The final step of the refactoring process is to change the creation and use of the `CreateInvoiceForm` form. Previously, we had something similar to the following to bring up the `CreateInvoiceForm` form and create an `Invoice` object:

```
using (CreateInvoiceForm form = new CreateInvoiceForm())
{
    if (form.ShowDialog(this) == DialogResult.OK)
    {
        invoices.Add(form.GetInvoice());
        //...
    }
}
```

We now want to create a `Presenter` after creating the `View`, have it initialize the data in the `View`, and create the `Invoice` object. This could be performed as follows:

```
using (CreateInvoiceForm form = new CreateInvoiceForm())
{
    CreateInvoicePresenter presenter =
        new CreateInvoicePresenter(form);
}
```

```
presenter.Start();
if (form.ShowDialog(this) ==
    System.Windows.Forms.DialogResult.OK)
{
    invoices.Add(presenter.GetInvoice());
    //...
}
}
```

To support *editing* an invoice, an `EditInvoicePresenter` class could be created that initialized the View with the data from an `Invoice` object, something like the following:

```
/// <summary>
/// Encapsulation of Model to View interaction
/// </summary>
class EditInvoicePresenter
{
    private ICreateInvoiceView view;

    public EditInvoicePresenter(ICreateInvoiceView view)
    {
        this.view = view;
    }

    public void Start(Invoice invoice)
    {
        foreach (var invoiceLineItem in invoice.LineItems)
        {
            view.InvoiceLineItemDTOs.Add(
                new InvoiceLineItemDTO(invoiceLineItem));
        }
        view.Title = invoice.Title;
        view.Date = invoice.Date;
    }

    public Invoice GetInvoice()
    {
        List<InvoiceLineItem> invoiceLineItems =
            new List<InvoiceLineItem>();
        foreach (InvoiceLineItemDTO invoiceLineItemDTO in
            view.InvoiceLineItemDTOs)
        {
            invoiceLineItems.Add(
                invoiceLineItemDTO.ToInvoiceLineItem());
        }
        return new Invoice(view.Title, invoiceLineItems,

```

```
        view.Date);  
    }  
}
```

The `EditInvoicePresenter` class differs from the `CreateInvoicePresenter` class by accepting an `Invoice` parameter in the `Start` method and initializing the `ICreateInvoiceView` properties. Going by that route, it may be worthwhile to rename `ICreateInvoiceView` to something more appropriate, like `IEditInvoiceView` (if you believe creating a new invoice is the same as editing a blank invoice).

Additional layers

We've discussed some common layers that often apply to many different software systems, large and small. As systems become more complex, you may find the need to address the complexity through additional layers. These layers may actually then become tiers – physically separated from the other tiers via the process boundary and communicated with via some out-of-process interface. These tiers may be communicated with on other computers via network communications (sockets, HTTPS, HTTP, WCF, and so on) or they may be communicated with over process boundaries on the same computer through named pipes, COM, WCF, and so on.

How you layer your application depends entirely on its complexity, architecture, and deployment needs. This book isn't a reference on how to layer all applications or how to refactor to every possible layered architecture. However, there are, some other layers you may run into or that may be applicable to your particular software system:

- **Application layer:** This layer involves housing logic that coordinates tasks between domain entities or other layers but doesn't contain business logic or business rules. This layer shouldn't exist unless it adds value. If you can't think of something that would fit in this layer, you don't need it.
- **Service layer :** This is another name for the application layer.
- **Infrastructure layer:** A general layer that houses infrastructure-specific logic of the system. Logic that might fit within this layer: graphical drawing (controls, etc.), messaging, persistence (in lieu of a Data Access Layer), logging, and so on.
- **Presentation layer:** This is another name for the User Interface Layer.

Summary

We've seen that componentization into layers keeps specific functionality decoupled from the rest of the system. When we need to make changes to specific functionality we can do that more easily with a layered architecture – we can keep logic that is proven to work and is more stable separate from logic that is less stable and more subject to change. Refactoring to a layered architecture allows us to support change better and minimize the impact of those inevitable changes.

In the next chapter, we'll get into more detail with architectural patterns, what improvements they offer, and how they facilitate the evolution of a software system. Architectural patterns such as specification, strategy, and observer patterns will be detailed, as well as what can be refactored to them.

9

Improving Architectural Behavior

In the previous chapter, we examined grouping similar functionality with like external dependencies into layers. These layers componentized a type of functionality to decouple it from the rest of the system with specific relationships between layers to maintain that decoupling.

We'll continue the architectural refactoring theme, focusing on refactoring architectural behavior. We're not concerned with the external behavior of our system, only with how our system behaves with regard to the behavior of algorithmic execution, assignment of the responsibility of executing those algorithms, and communication between the objects that have this responsibility.

This chapter will comprise the following areas of refactoring architectural behavior:

- Behavioral patterns
- Refactoring to behavioral patterns
- Strategy pattern
- Specification pattern
- Publish/Subscribe control paradigm
- Observer pattern

Behavioral patterns

Behavioral patterns are software design patterns that attempt to systematize common ways of dealing with algorithms and assigning the responsibility of those algorithms between classes. For the most part, behavioral patterns create or promote class-level separation of algorithms and their use. We gain explicitness from designing our system in this way.

There are two types of behavioral patterns: class patterns and object patterns. Class patterns use inheritance to implement making the distribution of responsibility explicit. Object patterns use object composition to implement making the distribution of responsibility explicit. Inheritance as a means to keep responsibility separate is probably fairly obvious; object composition might require a bit of explanation. Object composition uses distinct classes in the same way that inheritance would need to; but those classes are unrelated and would be used in groups within another to implement specific functionality.

So, why would we want to refactor to any behavioral pattern in general? I'm glad you asked. There are really two reasons for using any sort of pattern or principle. The first is the warm-and-fuzzy feeling you get from *correctness*. Making concepts explicit through the use of a class definition, future-proofing, easier to recognize, and so on are all viable reasons for implementing any sort of pattern or principle; but that deals mostly with new design. We're dealing with existing design. While *correctness* may be a good thing to strive for in an existing code base, we need to avoid making changes for the sake of changes. We have to deal with the fact that we have a code base that *works* (*works* as in it has a certain level of quality and users are using it in a certain way). Making changes risks affecting how the software behaves, and we want to avoid that unless we have a very good reason. We also have to deal with the fact that, despite our aptitude for writing code, we have deadlines that affect the financial viability of the project.

The second reason to refactor to a behavioral pattern is that we know that continued use of specific business logic or a business rule, as it is, is going to cause maintainability issues or hinder our efforts at adding functionality to the system.

To focus what we should (and shouldn't) refactor to a behavioral pattern, there's some specific changes we need to make to our system. One change that hints at refactoring to a behavioral pattern is copying business logic from one place to another.

Don't Repeat Yourself (DRY)

I couldn't resist the irony of re-addressing DRY in this chapter. Truth be told, it's not just about the irony, one of the main intentions of behavioral patterns is to address the inherent repetition of the type of business logic that behavioral patterns address.

Reuse is a bit of a fallacy with object-oriented design. Much of what we do in object-oriented languages is making our code base more maintainable so we can focus on the value we're trying to add to our users. It's not that we're trying to make everything reusable; we're trying to make sure important concepts in the domain we're modeling are explicit and that these important concepts aren't implicitly repeated throughout our system. Business logic and business rules that aren't explicit are harder to detect, harder to update (when repeated) and harder not to repeat.

We covered it in Chapter 2; but, DRY helps us have a one-stop-shop for each of our important domain logic and rules to avoid having to hunt down multiple instances of our domain logic and rules and update them, should our business rules need to change. Behavioral patterns help us accomplish this.

So, knowing that we're about to repeat ourselves in code is a good impetus for refactoring to a behavioral pattern to keep the repetition to repeated method calls or repeated class instantiations. Another reason for refactoring to a behavioral pattern is that we've found repeated business logic and we know we need to change it. If we're going to change it, we might as well encapsulate it into an implementation of a behavioral pattern, so further changes will be easier to make. Some code smells come into play when thinking about refactoring to behavioral patterns. When adding another condition to a complex conditional, addressing a Conditional Complexity code smell is a good motivation to refactor to a behavioral pattern. If you have to modify a complex conditional in any way, simplifying it before modifying it will help make it easier to modify in the future.

We've shown some criteria by which we can focus our refactoring efforts as it applies to behavioral patterns in general. Let's now look at some specific behavioral patterns.

Strategy pattern

The strategy pattern is sometimes known as the Policy pattern. Strategy encapsulates a group of specifically related business rules or *policies* that the system is trying to model. It provides an object-oriented way, through polymorphism, to alternate business rules at run-time, based upon criteria.

Strategy is commonly implemented as a class behavioural pattern by defining an interface or an abstract class, then having each specific business rule be implemented by a class that implements that interface or is a subclass.

```
// Trace output for current operation
TraceMethod traceMethod = GetTraceMethod();
String traceText = "testing";
if (traceMethod == TraceMethod.Trace)
{
    Trace.WriteLine(traceText);
}
else if (traceMethod == TraceMethod.Console)
{
    Console.WriteLine(traceText);
}
else if (traceMethod == TraceMethod.Debug)
{
    Debug.WriteLine(traceText);
}
```

In the above code, we have three specific blocks of business logic: one that uses `System.Trace`, one that uses `System.Console`, and one that uses `System.Diagnostics.Debug`. Which `WriteLine` method that is called is based on a condition, the state of the `TraceMethod` variable `traceMethod`. Each one of these lines of code is a specific policy.

Detecting need for strategy pattern

The strategy pattern is just an implementation of a runtime selectable policy. If there is a single policy (business rule) that applies to a single scenario, there's no need to implement the Strategy Pattern. If you find that additional policy needs to be invoked in a given situation, based on some condition – even if there is currently only one policy being invoked, refactoring to the Strategy Pattern will ease the addition of another policy.

Use of a policy can manifest itself within code in a variety of ways. One way is through multiple `if` statements that execute code with similar result. This can often manifest as multiple returns based on several conditions. For example, in order to calculate tax for our viewable invoice in `InvoiceRenderingService`, we could have a simple block of code as follows:

```
// calculate tax
if (!invoice.Customer.IsTaxExempt)
{
    invoiceTotalTax += (lineItemSubTotal *
        (decimal)(invoiceLineItem.TaxRate1 == null ?
            0F : invoiceLineItem.TaxRate1.Percentage))
    + (lineItemSubTotal *
```

```

(decimal) (invoiceLineItem.TaxRate2 == null ?
    OF : invoiceLineItem.TaxRate2.Percentage));
}

```

This code simply increments our invoice's total tax by the amount of the invoice line item price by the tax rates (if present), if the particular customer is not tax exempt.

We have two distinct policies here when it comes to calculating tax. One is that tax is calculated based on multiplying the invoice line item's price by any tax percentage applicable to that line item. The other, which may not seem like a calculation at all, is to not calculate any tax. The addition of another scenario for calculating tax, or changing one of the two existing scenarios means we need to change our `InvoiceRenderingService` class—that is, `InvoiceRenderingService` has more than one reason to change (See Single Responsibility Principle).

Refactoring to strategy pattern

Our goal isn't change for the sake of change, regardless of whether the resulting change results in a *better* design. I'll have to leave it up to you, dear reader, to ultimately decide the *why* when it comes to refactoring to the strategy pattern. In terms of refactoring to use the Strategy Pattern, how do we know what is a candidate for replacing with a Strategy Pattern implementation?

Strategy has some basic criteria for usage. The base criteria are that two or more related blocks of business logic code are being executed based upon some condition. The blocks of business logic code are mutually exclusive but related. The business logic also has to have similar or identical semantics. That is, each piece of business logic is used in a very similar way. We can't make use of polymorphism if the blocks of business logic have completely different usage patterns.

Well, I've given up the secret to refactoring to the strategy pattern: polymorphism. Okay, I hope it wasn't that much of a secret—it should have been obvious. Typically, the polymorphism used when implementing the Strategy Pattern is to use **Subtype Polymorphism**.



Subtype Polymorphism is often what is referred to as "polymorphism" when discussing object-oriented languages. Subtype Polymorphism is the ability for one of many types to be used for the invocation of a particular method. This is done through inheritance. Subtype class objects can be used wherever a supertype is expected.

In the case of implementing Strategy, the most common form of *inheritance* is implementing an interface. So, the first step in refactoring to the Strategy Pattern is the design and declaration of an interface that can consistently be used to implement the possible policies that we'd like to implement. In our tax calculation case, it's fairly straight-forward: we need a method that takes an amount and one or more `ITaxRate` objects, then returns the tax to be charged on the item. In order to implement that, we may have the following interface:

```
public interface ITaxCalculationStrategy
{
    float CalculatTaxReceivable(float amount,
        params ITaxRate[] taxRates);
}
```

We'd then replace our existing tax calculation code with something like the following:

```
invoiceTotalTax += taxCalculationStrategy.CalculatTaxReceivable(
    lineItemSubTotal,
    invoiceLineItem.TaxRate1, invoiceLineItem.TaxRate2);
```

This code, of course, assumes there is an `ITaxCalculationStrategy` variable `taxCalculationStrategy` that gets declared and initialized elsewhere. This variable could be a member variable in the containing class that get's initialized through Dependency Injection, or it could be initialized within this class.

In any case, we need to create some implementations of `ITaxCalculationStrategy` in order for our `taxCalculationStrategy` to be initialized, regardless of how it gets initialized. To that effect, let's split out our original code into two `ITaxCaulcuationStrategy` implementations. The more processing-intensive of the two is the non-tax exempt calculation. We'd refactor that simply by creating a new class, say `NormalTaxCalculationStrategy`, have it derive from `ITaxCalculationStrategy` and let Visual Studio® implement the stub methods by right clicking `ITaxCalculationStrategy` in the `NormalTaxCalculationStrategy` class definition and selecting **Implement Interface\Implement Interface**. That results in the following code:

```
class NormalTaxCalculationStrategy : ITaxCalculationStrategy
{
    public float CalculatTaxReceivable(float amount,
        params ITaxRate[] taxRates)
    {
        throw new NotImplementedException();
    }
}
```

To complete this particular step of the refactoring, we need to change our original code slightly. Our original code assumed there were only two possible tax rates which, unfortunately, is not the case everywhere. The `CalculateTaxReceivable` accepts an array of `ITaxRate` objects. So, instead of simply adding the amount multiplied by two tax rates; we'll iterate through the `ITaxRate` objects tallying the total tax payable. A completed implementation of `NormalTaxCalculationStrategy` may look like this:

```
class TaxCalculationStrategy : ITaxCalculationStrategy
{
    public float CalculatTaxReceivable(float amount,
        params ITaxRate[] taxRates)
    {
        float result = 0;
        foreach (var taxRate in taxRates)
        {
            if (taxRate != null)
            {
                result += amount * taxRate.Percentage;
            }
        }
        return result;
    }
}
```

The process for refactoring the tax exempt calculation into an `ITaxCalculation` implementation is very similar. What differs (besides the name of the class, which we'll call `ExemptTaxCalculationStrategy`) is the implementation of the `CalculateTaxReceivable` method; which in our case simply returns zero. The tax exception calculation refactored to an `ExemptTaxCalculationStrategy` class, unsurprisingly, may look like the following:

```
/// <summary>
/// Example tax exempt tax calculation policy
/// </summary>
public class ExemptTaxCalculationStrategy :
    ITaxCalculationStrategy
{
    public float CalculatTaxReceivable(float amount,
        params ITaxRate[] taxRates)
    {
        return 0;
    }
}
```


I've kind of skimmed over the actual instantiation of the `ITaxCalculation` objects, on purpose. At this point, the determination of which `ITaxCalculation` object to use depends on context. Also, the frequency of instantiation of these objects has many options. Let's start with determining instantiation strategy.

In our original pre-`ITaxCalculationStrategy` code, we effectively had the selection of the strategy within the `InvoiceRenderingService.RenderReadableInvoice` method each time the line item subtotal tax was calculated. We could go ahead and do effectively the same thing with our `ITaxCalculationStrategy` implementations and end up with something similar to the following:

```
ITaxCalculationStrategy taxCalculationStrategy;
if (!invoice.Customer.IsTaxExempt)
{
    taxCalculationStrategy =
        new ExemptTaxCalculationStrategy();
}
else
{
    taxCalculationStrategy =
        new NormalTaxCalculationStrategy();
}

invoiceTotalTax += (decimal)
    taxCalculationStrategy.CalculatTaxReceivable(
        (float)lineItemSubTotal,
        new ITaxRate[] {invoiceLineItem.TaxRate1,
            invoiceLineItem.TaxRate2});
```

This, of course, is functional. We create a single new `ITaxCalculationStrategy` object for each line item in the invoice whenever we render a readable invoice. It's the same object that gets instantiated and no two instantiations are used at the same time. Clearly this isn't the best approach to instantiating our `ITaxCalculationStrategy`.

Many would have the obvious inclination to move the instantiation of the object outside the loop. We then get a single instantiation per rendering. The `InvoiceRenderingService` class is still responsible for making the decision and we still have multiple `ITaxCalculationStrategy` objects being created unnecessarily. We could make an instance of each of the `ITaxCalculationStrategy` implementations static to the `InvoiceRenderingService` class; but, we then add the responsibility of managing a collection of these objects.

Remembering Chapter 7 and Dependency Injection, the most flexible way for `InvoiceRenderingService` to deal with which `ITaxCalculationStrategy` is to delegate to the code that creates our `InvoiceRenderingService`. Since the criteria by which an `ITaxCalculationStrategy` is independent from the criteria for instantiating `InvoiceRenderingService`, this injection is best done with an `ITaxCalculationStrategy` property that populates an `ITaxCalculationStrategy` field. This leaves the instantiation and `taxCalculationStrategy` variable out of our `RenderReadableInvoice` method. Our final refactoring of `RenderReadableInvoice` is to simply replace the previous tax calculation with the following:

```
invoiceTotalTax += (decimal)
    taxCalculationStrategy.CalculateTaxReceivable(
        (float)lineItemSubTotal,
        new ITaxRate[] {invoiceLineItem.TaxRate1,
            invoiceLineItem.TaxRate2});
```

Our refactored `RenderingService` (the other refactored parts) would look like this:

```
public class InvoiceRenderingService :
    IInvoiceRenderingService
{
    ITaxCalculationStrategy taxCalculationStrategy;
    ITaxCalculationStrategy TaxCalculationStrategy
    {
        set
        {
            taxCalculationStrategy = value;
        }
    }
    //...
}
```

Specification pattern

Within any code base there are always test for conditions that satisfy certain criteria. In a more procedural code base these tests are often strewn throughout the code. This, of course, works fine when we don't take into consideration extensibility and maintainability when we define *works*.

Sometimes, these tests aren't related to the domain we're modelling; in which case they're often constant and often of no direct correlation to our domain. The state of multiple UI controls, for example, is a common test to modify the state of another UI control. For example:

```
if (surnameTextBox.Text.Length != 0 &&
    givenNameTextBox.Text.Length != 0)
```

```
{
    continueButton.Enabled = true;
}
else
{
    continueButton.Enabled = false;
}
```

In this example, we only want the continue button to be enabled when the user has entered both a given name and a surname. The state of the continue button is an implementation detail of the UI, and not really a domain concern and not a business rule.

There are other tests that we perform in code that are business concerns; they're implementing a specific business rule. As has happened many times before, they can be implicit within other code, for example:

```
// an invoice is past due when it hasn't been paid
// in 30 days (and a grace period of 10 days)
if (DateTime.Now > invoice.Date.AddDays(30 + 10))
{
    DrawRotatedString(graphics, "PAST DUE", Font,
        redBrush, graphics.VisibleClipBounds,
        geometricallyCenteredStringFormat, 45.0);
}
```

In this code there is a test on an `Invoice` instance to compare its date to see if it's beyond a certain date. If it is beyond a certain date (and assuming it hasn't been paid) then it's past due. In this case, we display "PAST DUE" on the invoice.

This particular test is a very specific domain logic – a domain business rule. This business rule needs to be used throughout the system, not just when we print out an invoice. This is a particular type of business rule, it's a predicate. A predicate is a Boolean test on one or more criteria.

The specification pattern encapsulates the test of those criteria. The "specification" is the definition of the test. In our example, the specification is that past due invoices are 40 days old or older.

Detecting need for specification pattern

The specification pattern is just an implementation of a predicate; a specific test of values or objects based upon some criteria. You have to decide at some point to what degree you want to implement specification patterns throughout your code base. As we've discussed in prior chapters, we generally want to keep our domain layer very concise, explicit, cohesive, and loosely coupled. I suggest limit refactoring to specification pattern within the domain layer to make domain business rules very explicit and less likely to be repeated.

What is a candidate for specification pattern? Any specific logic that evaluates whether an object passes certain criteria could be a candidate. In other words, anything that involves one of our domain types in an `if` statement.

Now that we've found a candidate for use of specification pattern implementation, we need to focus on whether we should continue with the implementation based on the need or expectation of change or reuse. If this is just an exercise in implementing the specification pattern and there is no known need to implement the specification pattern other than it *can* be implemented here, our efforts and time might be better focused elsewhere.

Refactoring to specification pattern

With the goal of modifying or extending the domain predicates in our implementation, we can begin to make those relevant domain concepts explicit through the use of the specification pattern. At its essence, the specification pattern is simply making a predicate explicit by encapsulating it in its own class. In its simplest form, this could be a class that has an `IsSatisfied` method to perform the test. For example:

```
class UnpaidInvoiceSpecification
{
    public bool IsSatisfiedBy(Invoice invoice)
    {
        return invoice.Status != InvoiceStatus.Paid;
    }
}
```

In this simple specification implementation we perform the test in the `IsSatisfiedBy` method. If the invoice's status is not `Paid` then it satisfies the unpaid specification. More complex predicates could have state related to the criteria accepted within the constructor. Specifications generally perform the test on a single parameter to `IsSatisfiedBy` and other criteria are an attribute of the specification.

We could then make use of this specification, for example, to find unpaid and paid invoices:

```
List<Invoice> unpaidInvoices = new List<Invoice>();
List<Invoice> paidInvoices = new List<Invoice>();

foreach (Invoice invoice in invoices)
{
    if (unpaidInvoiceSpecification.IsSatisfiedBy(invoice))
    {
        unpaidInvoices.Add(invoice);
    }
    else
    {
        paidInvoices.Add(invoice);
    }
}
```

In this simple example, we iterate through a collection of invoices, executing the specification for each one and add the invoice to another collection depending on how it satisfied the specification.

If how we implemented the way the attribute of an invoice is being paid, or unpaid, changes, then the client code that depends on it doesn't need to change – it simply goes on making use of our `UnpaidInvoiceSpecification` class and it's none the wiser. We've successfully hidden this implementation detail from the client code.

Looking at this example, one thing may become apparent to many readers: its lack of scalability. Clearly, to find all unpaid invoices we need to load all the invoices into memory (into an `IEnumerable<T>` implementation) and evaluate the specification on each. Sure, the implementation of `IEnumerable<T>` may not need to load everything into memory; but, we end up storing the results in memory. Often, when we design specification implementations we like to take into account using that specification with our flavour of data access. We obviously don't want to include data-access specific code in our specification so we often off-load that heavy lifting to our Repository. For example:

```
class UnpaidInvoiceSpecification
{
    public bool IsSatisfiedBy(Invoice invoice)
    {
        return invoice.Status != InvoiceStatus.Paid;
    }
    public IEnumerable<Invoice> SatisfyingElementsFrom(
        InvoiceRepository invoiceRepository)
    {

```

```
        return
            invoiceRepository.SelectWhereInvoiceStatusNotPaid();
    }
}
```

We've now added a `SatisfyingElementsFrom` method that delegates the actual evaluation of the test to the repository – which delegates that to the RDBMS through whatever magic (like SQL) is required. This is a reasonable trade-off in certain situations. We've still got an explicit point of entry for implementations of our policy. If you're not in a situation where you're forced to implement something like this (that is, you're not very restricted in database query generation within the application) there's an alternative that lets us keep our specification implementation cohesive and leaves all the logic acting upon the criteria within our specification. Through the use of LINQ, we can make sure our specification implementation truly encapsulates the logic to evaluate the criteria. For example:

```
class UnpaidInvoiceSpecification
{
    public bool IsSatisfiedBy(Invoice invoice)
    {
        return invoice.Status != InvoiceStatus.Paid;
    }

    public IEnumerable<Invoice>
        SatisfyElementsFrom(IQueryable<Invoice> invoices)
    {
        return from invoice in invoices
            where IsSatisfiedBy(invoice)
            select invoice;
    }
}
```

We've changed `SatisfyElementsFrom` to accept an `IQueryable<T>` parameter, on which we perform a LINQ query using our `IsSatisfiedBy` method without having to repeat ourselves. The result of the `IQueryable<T>` parameter could be returned from somewhere else, like a `Repository`:

```
UnpaidInvoiceSpecification unpaidInvoiceSpecification =
    new UnpaidInvoiceSpecification();

IEnumerable<Invoice> unpaidInvoices =
    unpaidInvoiceSpecification.SatisfyElementsFrom(
        invoiceRepository.GetAllInvoices());
```

This gives us the ability to avoid repeating ourselves and have one tidy place where this specific business rule is implemented. It also doesn't need, in itself, to load all the invoices into memory to perform the action upon it, the `IQueryable<T>` would perform lazy loading or caching as it sees fit. But, it suffers from forcing the evaluation of the criteria on the local computer and not in the database. Clearly, it would be quicker and more efficient to only ask the database for the data that satisfies our criteria, not *everything*, then throw away what we don't need.

Fortunately, we can modify our specification slightly and keep the data-access-specific code in the Repository:

```
class UnpaidInvoiceSpecification : Specification<Invoice>
{
    public override bool IsSatisfiedBy(Invoice invoice)
    {
        return IsSatisfiedBy().Compile()(invoice);
    }
    public Expression<Predicate<Invoice>> IsSatisfiedBy()
    {
        return invoice => invoice.Status != InvoiceStatus.Paid;
    }
}
```

Where `UnpaidInvoiceSpecification` derives from the following abstraction:

```
public abstract class Specification<T>
{
    public abstract bool IsSatisfiedBy(T candidate);
    public abstract Expression<Predicate<T>> IsSatisfiedBy();
}
```

In this abstract class, we define our `IsSatisfiedBy` overloads so that other classes may use specifications as abstractions, as we'll see shortly. The first `IsSatisfiedBy` overload is what we've seen already: a method that takes a candidate object to test if it matches our criteria. The second is a new method that returns an `Expression<T>` object that may be used by other methods either directly or by first compiling the expression. It will become apparent why we chose to implement an abstract class instead of an interface later.

In `UnpaidInvoiceSpecification`, we've added an `IsSatisfiedBy` overload that returns an `Expression<Predicate<Invoice>>` object. The original `IsSatisfiedBy` method is updated to make use of this expression, compiling it to a delegate then executing it with a specific `invoice` instance so we don't repeat ourselves. The `Expression<>` object can now be passed along to another LINQ statement that can be used by applicable LINQ providers to build a database query that will only retrieve data from the database that satisfies our specification. For example:

```
public class InvoiceRepository
{
    private IQueryable<Invoice> invoiceQueryable;

    public IQueryable<Invoice>
        FindBySpecification(
            Specification<Invoice> specification)
    {
        return invoiceQueryable.Where(
            specification.IsSatisfiedBy());
    }
    //...
}
```

In this invoice repository, we have a `FindBySpecification` method that accepts an abstract `Invoice` specification instance. The `Expression<Predicate<Invoice>>` returned by the `IsSatisfiedBy()` method is simply passed on to an `IQueryable<Invoice>` instance. Depending on the LINQ provider that supplied the `IQueryable<Invoice>` instance, it will either pre-compile it to a delegate then invoke the delegate for every instance of `Invoice` that the collection contains, or use the expression to generate a query that will be executed directly on the database to return only `Invoice` data that fulfills our criteria. This will drastically reduce the amount of data that we receive from the database and drastically reduces the number of `Invoice` objects that we need to process. It's very likely that processing more `Invoice` objects is more time consuming than filtering out `Invoice` data at the database level.

It's important to note, now that we've introduced an abstract `Specification<T>` here, that we've introduced flexibility in how we deal with specification implementations. Up until now we've dealt directly with an `UnpaidInvoiceSpecification` instance and didn't support other specifications. The `InvoiceRepository.FindBySpecification` method accepts any type of invoice specification, not just our `UnpaidInvoiceSpecification`.

One of the things that come up often in terms of business rules that are predicates is their need to be chained. In terms of a Boolean test, they're not always used alone — you often want to use more than one predicate business rule in a single conditional. I'm sure almost every reader has encountered implementations of predicates business rule — like what we exemplified earlier — directly within the `if` statement. It's not that this doesn't work; the frequency by which it is implemented this way proves that it does. It's that specific business rule — a relevant domain concept — is implemented implicitly. In order to chain that predicate with other predicates in different places means repeating ourselves all over the code base.

The beauty of the specification pattern is its ability to fulfil chaining of specifications on a particular domain entity. With any Boolean logic we often need to use logical operators when we perform our test. We may want to decide if two criterion are met, or that one criterion is met and another is not, or that either one of two criterion are met. We could continue with every permutation of this; but these give us a base by which to build all those other permutations: AND, NOT, and OR. Through a definite hierarchy, we can build predicates (specifications) that can be chained through the use of these logical operators. Now, we're not talking about the logical operators in the language sense; we're talking about them in the abstract sense.

To support chaining of specifications we need to design a fluent interface on the abstract `Specification<T>` class that will be reused by anything that implements `Specification<T>`. To do that, we essentially create three new abstract implementations of `Specification<T>`:

```
public abstract class Specification<T>
{
    public abstract bool IsSatisfiedBy(T candidate);
    public abstract Expression<Predicate<T>>
        IsSatisfiedBy();
    public Specification<T> And(Specification<T> other)
    {
        return new AndSpecification<T>(this, other);
    }
    public Specification<T> Or(Specification<T> other)
    {
        return new OrSpecification<T>(this, other);
    }
    public Specification<T> Not()
    {
        return new NotSpecification<T>(this);
    }
    private class AndSpecification<T> : Specification<T>
```

```
{
    private Specification<T> left;
    private Specification<T> right;
    public AndSpecification(Specification<T> left,
        Specification<T> right)
    {
        this.left = left;
        this.right = right;
    }
    public override bool IsSatisfiedBy(T candidate)
    {
        return IsSatisfiedBy().Compile()(candidate);
    }
    public override Expression<Predicate<T>> IsSatisfiedBy()
    {
        return left.IsSatisfiedBy()
            .And(right.IsSatisfiedBy());
    }
}

private class OrSpecification<T> : Specification<T>
{
    private Specification<T> left;
    private Specification<T> right;
    public OrSpecification(Specification<T> left,
        Specification<T> right)
    {
        this.left = left;
        this.right = right;
    }
    public override bool IsSatisfiedBy(T candidate)
    {
        return IsSatisfiedBy().Compile()(candidate);
    }
    public override Expression<Predicate<T>> IsSatisfiedBy()
    {
        return left.IsSatisfiedBy()
            .Or(right.IsSatisfiedBy());
    }
}

private class NotSpecification<T> : Specification<T>
{
    private Specification<T> specification;
    public NotSpecification(Specification<T> specification)
```

```
    {
        this.specification = specification;
    }
    public override bool IsSatisfiedBy(T candidate)
    {
        return IsSatisfiedBy().Compile()(candidate);
    }
    public override
        Expression<Predicate<T>> IsSatisfiedBy()
    {
        var expression = specification.IsSatisfiedBy();
        return Expression.Lambda<Predicate<T>>(
            Expression.Not(expression.Body),
            expression.Parameters.Single());
    }
}
```

With this new abstract `Specification<T>` class, we've added three public methods: `And`, `Or`, and `Not`.

The `And` method accepts another `Specification<T>` parameter and performs a logical binary *and* between the current specification and the parameter. It does this by returning a new `Specification<T>` object, the private `AndSpecification<T>` class. The `AndSpecification<T>` class acts just like our other `Specification<T>` objects by effectively invoking the `IsSatisfiedBy` methods on the two other specifications and performing a logical *and* on their results.

The `Or` method accepts another `Specification<T>` parameter and performs a logical binary *or* between the current specification and the parameter. It does this by returning a new `Specification<T>` object, the private `OrSpecification<T>` class. The `OrSpecification<T>` class acts just like our other `Specification<T>` objects by effectively invoking the `IsSatisfiedBy` methods on the two other specifications and performing a logical *or* on their results.

The `Not` method performs a logical unary *not* between on the result of the specification's `IsSatisfiedBy` method result. It does this by returning a new `Specification<T>` object, the private `NotSpecification<T>` class. The `NotSpecification<T>` class acts just like our other `Specification<T>` objects by effectively invoking the `IsSatisfiedBy` method of the other specification and performing a logical *not* on the result.

In order for `AndSpecification<T>`, `OrSpecification<T>`, and `NotSpecification<T>` to do what it needs to do to with logical operations to support LINQ providers that build queries based upon expressions we made use of a couple infrastructure classes. The first class, `ExpressionParameterExchangerVisitor` is as follows:

```
public class ExpressionParameterExchangerVisitor :
    ExpressionVisitor
{
    private readonly Dictionary<ParameterExpression,
        ParameterExpression> map;

    private ExpressionParameterExchangerVisitor(
        Dictionary<ParameterExpression, ParameterExpression>
            map)
    {
        this.map = map ??
            new Dictionary<ParameterExpression,
                ParameterExpression>();
    }

    protected override Expression
        VisitParameter(ParameterExpression p)
    {
        ParameterExpression replacement;
        if (map.TryGetValue(p, out replacement))
        {
            p = replacement;
        }

        return base.VisitParameter(p);
    }

    public static Expression
        ReplaceParameters(Dictionary<ParameterExpression,
            ParameterExpression> map, Expression exp)
    {
        return new ExpressionParameterExchangerVisitor(map)
            .Visit(exp);
    }
}
```

The `ExpressionParameterExchangerVisitor` class implements an `Expression` visitor that visits each parameter of an expression. When it visits a parameter, and that parameter is found in the map, it's replaced with another parameter. This class is used by `ExpressionExtensions.Combine` to ensure that two combined predicates use one set of parameters (or parameter, in this case).

The second class is an extension method class that adds the following extension methods to `Expression<T>`: `Combine`, `And`, and `Or`. This class is as follows:

```
public static class ExpressionExtensions
{
    private static Expression<T>
    Combine<T>(this Expression<T> first,
              Expression<T> second,
              Func<Expression, Expression, BinaryExpression>
              operation)
    {
        // build a map of parameters from both
        // the second and first expressions
        var map = first.Parameters.Select((f, i) =>
            new { f, s = second.Parameters[i] })
            .ToDictionary(p => p.s, p => p.f);
        // make sure the second expression uses
        // the same parameters as the first
        var secondBody =
            ExpressionParameterExchangerVisitor.ReplaceParameters(
                map, second.Body);
        // create new expression from an operation on the first
        // and second expression bodies.
        return
            Expression.Lambda<T>(operation(first.Body,
                secondBody), first.Parameters);
    }

    public static
    Expression<Predicate<T>>
    And<T>(this Expression<Predicate<T>> first,
          Expression<Predicate<T>> second)
    {
        return first.Combine(second, Expression.And);
    }

    public static
    Expression<Predicate<T>>
    Or<T>(this Expression<Predicate<T>> first,
          Expression<Predicate<T>> second)
    {
        return first.Combine(second, Expression.Or);
    }
}
```

The `Combine` method combines one predicate with another predicate. We need to combine the two predicates into one in order for the LINQ provider to parse it and be able to generate a query statement based on the expressions. The `And` and `Or` method use the `Combine` method to combine two predicate expressions with specific operation: `And` and `Or` respectively. These two methods make it easier to read code that combines two predicates (that is, `AndSpecification<T>` and `OrSpecification<T>`).

Let's look at how we'd make use of this comprehensive foundation for implementing flexible specifications. There's not much point in performing Boolean logic when you only have one predicate; so, let's add another short and sweet invoice specification based upon a business rule:

```
class OverdueInvoiceSpecification : Specification<Invoice>
{
    private const int GRACE_PERIOD_DAYS = 10;
    private const int NET_DUE_DAYS = 30;
    private DateTime currentDate;

    public OverdueInvoiceSpecification(DateTime currentDate)
    {
        this.currentDate = currentDate;
    }

    public override bool IsSatisfiedBy(Invoice invoice)
    {
        return IsSatisfiedBy().Compile()(invoice);
    }

    public override
        Expression<Predicate<Invoice>> IsSatisfiedBy()
    {
        return invoice => currentDate >
            invoice.Date.AddDays(NET_DUE_DAYS)
                .AddDays(GRACE_PERIOD_DAYS);
    }
}
```

In this specification, we've gotten slightly more complex such that we've parameterized the specification to accept a current date. We could have simply used `DateTime.Now` as the date to compare; but then we've got the potential for the specification to have different results depending on the time in which it's executed. To avoid that, we store the date to compare within the state of the specification.

Our `IsSatisfiedBy()` method compares the current date with the overdue date (net due timeframe plus a grace period) and checks to see if the current date is beyond that. If so, the invoice is overdue.

Knowing which invoices are overdue is not as useful without knowing whether the invoice is unpaid (although, the inverse is not always true). Now that we have our logical operators for specifications, we can use both of these specifications to discern whether an invoice is overdue and unpaid. For example:

```
UnpaidInvoiceSpecification unpaidInvoiceSpecification =
    new UnpaidInvoiceSpecification();
OverdueInvoiceSpecification overdueInvoiceSpecification =
    new OverdueInvoiceSpecification(DateTime.Now);

if (unpaidInvoiceSpecification
    .And(overdueInvoiceSpecification).IsSatisfiedBy(invoice))
{
    SendOverdueReminderEmail(invoice);
}
```

In this snippet of code, we create our two specifications: `unpaidInvoiceSpecification` and `overdueInvoiceSpecification`. We then use the `Specification<T>.And` method on the `unpaidInvoiceSpecification` object passing it the `overdueInvoiceSpecification` object. This merges the two predicates with an *and* operation. Finally, we call the `IsSatisfiedBy` method passing in the `invoice` object to test if the invoice is unpaid *and* overdue; and, if so, send a reminder e-mail.

The semantics would be similar if we wanted to detect if an invoice was unpaid *or* overdue:

```
if (unpaidInvoiceSpecification
    .Or(overdueInvoiceSpecification).IsSatisfiedBy(invoice))
{
    //...
}
```

The difference here is that we use the `Or` method instead of the `And` method. The *not* operation has slightly different semantics to it because it is a unary operation:

```
if (overdueInvoiceSpecification
    .And(unpaidInvoiceSpecification.Not()).IsSatisfiedBy(invoice))
{
    //...
}
```

With the `Not` method, its result should be passed to one of the binary operations to be chained. In this case, we're testing to see if the invoice is overdue but paid.

With the creation of our specification classes and an understanding of the semantics, we can begin to refactor our existing code. With specifications it's a simple matter of creating (or obtaining from our Dependency Injection container) a Specification object than replacing our inline use of predicate business rules. If we revisit our original example:

```
// an invoice is past due when it hasn't been paid
// in 30 days (and a grace period of 10 days)
if (DateTime.Now > invoice.Date.AddDays(30 + 10))
{
    DrawRotatedString(graphics, "PAST DUE", Font,
        redBrush, graphics.VisibleClipBounds,
        geometricallyCenteredStringFormat, 45.0);
}
```

We can replace this test with a call to our `OverdueInvoiceSpecification` as follows:

```
OverdueInvoiceSpecification overdueInvoiceSpecification =
    new OverdueInvoiceSpecification(DateTime.Now);

if (overdueInvoiceSpecification.IsSatisfiedBy(invoice))
{
    DrawRotatedString(graphics, "PAST DUE", Font,
        redBrush, graphics.VisibleClipBounds,
        geometricallyCenteredStringFormat, 45.0);
}
```

Publish/Subscribe paradigm

In every software system, there are specific times when something needs to occur in response to something else. Sometimes, this is as simple as adding code into an existing sequence of code and sometimes this is adding a method call with a return value into an existing sequence of code. Often what occurs is not relevant to what is currently occurring, in other words it's not its responsibility. For the most part, this is fine; we don't really care how things get done, just that things do get done. This works well when we have a one-to-one relationship to what needs to occur in response to something else. This even works when we have a one-to-x relationship; where x is a fixed number.

This model starts to break down when you need to be able support a variable number of responses or an unknown set of responses. In our one-to-one model, the response is directly coupled to the action that is occurring. If that's all that needs to happen then that's great. However, when we have a variable number of responses, an unknown set of responses, or an asynchronous action with a response, then we obviously can't implement that coupling without both sides at the table.

The publish/subscribe paradigm, at the lowest level, recognizes that there are situations where there are external entities (subscribers) that would like to be involved with the result of a particular action performed by a subsystem (publisher).

Publish/subscribe is based on the model that something would like to be informed of information when it is available and that something publishing that information may have zero or more subscribers to that information.

Observer pattern

The seminal definition of the Observer Pattern involved the publisher containing publisher-specific code to keep track of all the subscribers through a collection of interface or base class subscribers. The publisher would know how to communicate with subscribers because they implemented a specific interface. When the publisher published information that subscribers subscribed to, the publisher would iterate through all the subscribers invoking whatever method it required of the subscribers to inform them of the information. This required subscribers to implement a specific interface, or derive from a specific base class, and implement specific methods with specific names and a specific signature. The Observer pattern is one method of implementing Publish/Subscribe.

Although much of that management could be delegated to another class; this is pretty typical for most programming languages. .NET languages like C# and VB have the ability to implement events. An event is something a type can specifically add to their interface so that subscribers can optionally subscribe to be informed of certain information. Events are implemented with delegates. Something that publishes events effectively manages a collection of delegates associated with the event. When the class reaches the point where it has information that subscribers are interested in, it iterates the collection and invokes each delegate with the information that it is interested in. This implementation frees the publisher from providing an interface or subclass for it to understand how to communicate with subscribers and also frees the subscribers from having to implement the coupling semantics through that specific interface or subclass. We've avoided tangible ceremony that really didn't add any value to either side. Events essentially use **Duck Typing** to determine the coupling semantics.



Duck Typing is a style of coupling semantics whereby the semantics is not determined by the inheritance of a specific class or implementation of a specific implementation, but by a specific method signature.

At its heart, the Observer Pattern recognizes that certain actions performed by one class might need to be observed by many other classes. We use events to implement the Observer Pattern in C#.

Detecting the need for the observer pattern

There are all sorts of places in most .NET applications that use the Observer Pattern. These usages are probably pretty obvious. In WinForm applications, for example, events are a very common way for user interface actions to be communicated back to application code. The click of the mouse on a button, for example, is communicated back to an application via the `Button.Click` event.

It's easy to add an event and wire up an event handler. But, it's harder to detect code that could benefit from an Observer Pattern implemented as an event.

It's the other areas of the code that could benefit from the Observer Pattern where it's difficult to detect. One particular area that could benefit from an Observer Pattern implementation is providing an API for external applications to make use of our software system.

Another area where the Observer Pattern shines is with the Model View Presenter (MVP) pattern. Rather than having a circular dependency between the View and the Presenter, the Presenter can be an Observer of the View and the View could publish events that the Presenter subscribes to.

Refactoring to the observer pattern

Once we have candidate code that could benefit from the Observer Pattern we need to extract it from its current class and move it to another class. What we need to do is decouple the subscriber from the publisher – which requires that they be separate classes.

Our invoicing application needs to add dashboard information that displays the current state of the system. It will display information like number of invoices, number of unpaid invoices, and so on. We could couple components like the repository to the view and have the repository inform the view of information directly through a method on the view. But, this would violate our layering rules and introduce a dependency cycle between the domain and the presentation layers.

The alternative to this is to use the observer pattern. The view (likely via a presenter, if using the MVP pattern) would be an observer of the repository and be informed of the changes to data that the repository persists.

What information that is observed by the subscriber depends on circumstances. In the case of being able to update our dashboard, the information could be as shallow as *something changed* or it could be as deep as the actual number of number of invoices and the number of unpaid invoices. Generally, I try to keep the information being published to information that would normally be available. If, for a given operation, the number of unpaid invoices is not known, that information wouldn't be published even if the operation may affect the number of unpaid invoices. It's not really the responsibility of that component to get that information if it doesn't already have it. If the subscriber needs to get that information when it gets published information that could mean that information has changed, it has the ability to go get it.

If we revisit our InvoiceRepository class:

```
public class InvoiceRepository : IInvoiceRepository
{
    IDataAccess dataAccess;

    public InvoiceRepository(IDataAccess dataAccess)
    {
        this.dataAccess = dataAccess;
    }

    public void Save(Invoice invoice)
    {
        DataSet invoiceDataSet =
            dataAccess.LoadInvoice(invoice.Id);
        DataRow invoiceRow;
        if (invoiceDataSet == null ||
            invoiceDataSet.Tables.Count == 0)
        {
            if (invoiceDataSet == null) invoiceDataSet =
                new DataSet();
            DataTable invoiceTable =
                invoiceDataSet.Tables.Add("Invoices");
            DataColumn column = new DataColumn("Id",
```

```

        typeof(Guid));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Date", typeof(DateTime));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Title", typeof(String));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Status", typeof(int));
        invoiceTable.Columns.Add(column);
        invoiceRow =
            invoiceDataSet.Tables["Invoices"].NewRow();
        invoiceRow["Id"] = invoice.Id;
    }
    else
    {
        invoiceRow =
            invoiceDataSet.Tables["Invoices"].Rows[0];
    }
    invoiceRow["Date"] = invoice.Date;
    invoiceRow["Status"] = invoice.Status;
    invoiceRow["Title"] = invoice.Title;
    dataAccess.SaveInvoice(invoiceDataSet);
    // Save line item data...
}
//...
}

```

To refactor this to support notification of our view, we would add a `DataUpdated` event to `InvoiceRepository`, and raise that event in the `SaveInvoice` method. For example:

```

public class InvoiceRepository : IInvoiceRepository
{
    IDataAccess dataAccess;

    public InvoiceRepository(IDataAccess dataAccess)
    {
        this.dataAccess = dataAccess;
    }

    public event EventHandler<EventArgs> DataUpdated;
        = (o, a) => { };

    public void Save(Invoice invoice)
    {
        DataSet invoiceDataSet =
            dataAccess.LoadInvoice(invoice.Id);
        DataRow invoiceRow;
    }
}

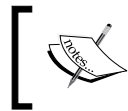
```

```
if (invoiceDataSet == null ||
    invoiceDataSet.Tables.Count == 0)
{
    if (invoiceDataSet == null)
        invoiceDataSet = new DataSet();
    DataTable invoiceTable =
        invoiceDataSet.Tables.Add("Invoices");
    DataColumn column = new DataColumn("Id",
        typeof(Guid));
    invoiceTable.Columns.Add(column);
    column = new DataColumn("Date", typeof(DateTime));
    invoiceTable.Columns.Add(column);
    column = new DataColumn("Title", typeof(String));
    invoiceTable.Columns.Add(column);
    column = new DataColumn("Status", typeof(int));
    invoiceTable.Columns.Add(column);
    invoiceRow =
        invoiceDataSet.Tables["Invoices"].NewRow();
    invoiceRow["Id"] = invoice.Id;
}
else
{
    invoiceRow =
        invoiceDataSet.Tables["Invoices"].Rows[0];
}
invoiceRow["Date"] = invoice.Date;
invoiceRow["Status"] = invoice.Status;
invoiceRow["Title"] = invoice.Title;
dataAccess.SaveInvoice(invoiceDataSet);
// Save line item data...

DataUpdated(this, EventArgs.Empty);
}
//...
}
```

Summary

In this chapter, we've seen how we can refactor architectural behavior to increase cohesion and reduce coupling to support evolving our code base. We've seen through the use of the behavioral patterns Strategy, Specification and Observer that we can make our classes more extensible. This helps us abide by the **Open/Closed Principle** by providing a way of extending our classes' behavior without having to directly modify them.



Open/Closed Principle states software entities (classes, modules, functions, and so on) should be open for extension, but closed for modification.



In the next chapter, we'll detail refactoring architectural structure to make adding features easier and make working with third party libraries easier.

We'll also see how interfacing with the database less time consuming.

10

Improving Architectural Structure

In the previous chapter, we detailed improving architectural behavior where we discussed how the use of patterns helps us improve our architecture's behavior so that we are better equipped to evolve and maintain our code.

In this chapter, we'll look at improving architectural structure through the use of structural patterns. These patterns will include the following:

- Adapter
- Façade
- Proxy

We'll also look at **Object/Relational Mapping (ORM)** as a way of simplifying the architectural structure and reducing the amount of code to reduce the work involved in supporting database changes and expansion.

Structural patterns

Structural patterns define ways of structuring parts of the software system to compose other objects. Just as with behavioral patterns, there are *class* structural patterns and *object* structural patterns. Class structural patterns involve using inheritance as a mechanism to structurally compose new classes from another class or from a class hierarchy. The single-inheritance restriction of .NET means that our ability to use inheritance to compose new classes is restricted. For the most part, structural patterns are implemented as *object* structural patterns, which mean a class is declared as a container for one or more other classes.

Legacy code

Michael Feathers describes Legacy code as "...code without tests". Michael details many ways of managing and dealing with legacy code. The ideal strategy for dealing with legacy code is to make sure there are automated tests for the majority of the code. I say "majority" because the *Pareto principle* (also known as the 80-20 rule) and the *law of diminishing returns* apply to code coverage with automated tests – the work involved in obtaining 100% coverage is prohibitively expensive for the returns. We won't get into refactoring to support unit testing until the next chapter; so, we'll continue with the impetus that we're refactoring to towards a better design. Another strategy for dealing with legacy code is to simply rewrite it – adding supported automated testing as code is written. We've effectively rejected this strategy as being too costly – after all, if we couldn't write it correctly the first time, what makes us think we can do it right this time? Yet, another strategy is to view the legacy code to support adding features or modifying dependant code to simply decouple it with an abstraction.

There are all sorts of reasons why parts of a code base become "legacy". It's never intentional. One of the biggest reasons I've seen for portions of a code base becoming "legacy" is narrow responsibility. One person has historically been responsible for a specific portion of the code base and that person is no longer on the team. No one else on the team has had the budget to come up to speed on the code to fully understand how to evolve and change that part of the code base. As a result, that portion of the code base has languished; it hasn't evolved with the needs and standards of the rest of the code base and has become "legacy code". No one wants to touch it, it's difficult to change and changes often result in defects (from lack of understanding).

Difficult to evolve code isn't only a symptom of an abandoned portion of code; it could be that a portion of code base has always been found to be difficult to modify by members of the team. This often happens when code is written without any clear understanding of either how it works or why it works. It is worked on until it passes some abstract bar of functionality – that is, "it works" – then no one wants to touch it for fear of breaking it (the House of Cards anti-pattern). On more experienced software development teams, this is somewhat rare.

For the most part, this chapter is about how to deal with this legacy code. Whenever possible, I'll point out areas where these refactorings make sense outside of dealing with legacy code; but, the main motivation for these refactorings is to lessen the impact that the legacy code has upon forthcoming changes.

In *Chapter 1*, we outlined working refactoring into the process. The motivation is to accept that change occurs and the impact on requirements means what we've already worked on may need to change regardless of how well it works. It's important for members of the development team to recognize this and have a reasonable understanding of all areas of the software system. It's important to schedule time to analyze existing code to make sure our understanding of the system hasn't changed as that code was written and taken on technical debt. Part of analyzing that code and looking for technical debt, is to refactor the code to remove the technical debt, if feasible. Legacy code is no different, to respond effectively to change, someone has to understand that legacy code and refactor it to make it more responsive to change.

Adapter pattern

Quite simply, the adapter pattern simply adapts one interface into another, more acceptable interface. This structural pattern can be implemented in either of the two implementation methods: *object* and *class*. There are circumstances where we can perform structural *class* adaptation, by inheriting from another class and providing a new, more application-friendly, interface. It's been my experience that this particular implementation of the adapter pattern is rarer. The more common implementation, from my experience, has been the structural *object* implementation, by creating a new class (the adapter) that composes instances of one or more other classes. Neither structural implementation is that different than the other. You can simply view the *class* implementation as an *object* implementation with a single composed object that happens to be the base class. The drawback of the *class* implementation is that the base classes' interface can't be entirely hidden by an abstraction as can be done with the *object* implementation.

Detecting need for the adapter pattern

There are many reasons why using the adapter pattern adds value. A third party library has a fixed interface; if we accept that we need to use said third party library, we accept that we need to use its interface. The interface may not be geared to our particular usage patterns – it has to deal with multiple clients and our usage patterns simply aren't part of the common scenarios. The third party library, while useful, may simply have a poorly written interface.

In terms of code in our code base, we may have legacy code that is simply too costly to make anything other than trivial changes to. Much in the same way as a third party library, our legacy code may suffer from the same flaws that our use of third party libraries does. The legacy code may not have evolved in the same way as the rest of the code base and has become increasingly difficult to work with. We've already dealt with refactoring code to make it easier to use and thus more maintainable, but what about circumstances where we simply can't do that? We may simply not have the resources like time or experienced developers to make that level of change. Often, the lack of experience with that portion of the code base means no one can really estimate the level of effort required to make the change. This is often rare; but still does occur.

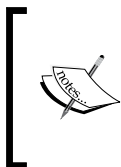
In these circumstances it is often better to simply "wire-off" that portion our code base, decoupling it through an abstraction. This decreases our coupling to that portion of the code base and will allow us to evolve the rest of the code base more efficiently and more effectively.

When we introduce an adapter to any other interface, we can often design that adapted interface to be more testable. At the very least, creating an adapter should make the code that uses that adapter more testable because that adapter decouples its client from what is being adapted and thus more easily isolated during tests.

Refactoring to the adapter pattern

It's hard to show specific examples of refactoring to the adapter pattern that don't seem pedantic to at least some readers. The motivation behind any particular example to show the process more than anything else but I find it helpful to show real-world examples that readers can associate with and at least some find useful.

Implementing the **Model View Presenter (MVP)** pattern in a WinForms application is an example of a structural class adapter pattern implementation. Effectively, we're inheriting from a `Form` object, providing new interface (the View) for the Presenter to use. This allows our presenter to be decoupled from `Form` and allow isolation from it.



Model View Presenter (MVP) pattern is where the responsibilities of the domain, the interaction with the user (UI), and transformation of domain-specific data specifically for the UI (presentation) are decoupled from one another in the Model, View, and Presenter – respectively.

This is useful during test so that we can substitute the view for a **Test Double** like a stub or a mock so that parts of the system can be tested through automation without having to invoke a user interface.



Test Double is something that can be substituted for something else—usually in the context of an automated test.

Apart from testability and increased decoupling, refactoring to the MVP pattern is useful if you need share presentation logic across multiple types of views. The "view" is any display of data in order for a user or client to act upon. This obviously is a user-interface view; but could also include the addition of a web service interface, another user-interface view, an API, and so on.

Back in *Chapter 8*, we refactored the `ViewInvoiceForm` for Dependency Injection. We'll now refactor `ViewInvoiceForm` to implement the Model View Presenter Pattern to show a *class adapter pattern* implementation.

When implementing MVP, I like to avoid having a circular dependency between the View and the Presenter. The Presenter deals with coupling to the Model which means it gets data from the Model to give to the View; but the View receives commands from the user that affects the Model. This can easily lead to a circular dependence—a cycle—between the Model and the View. This can lead to some maintenance and flexibility issues. To avoid this, the coupling from the View to the Presenter through Inversion of Control, implemented with the Observer Pattern—the View communicates with the Presenter via events that the Presenter subscribes to.

The first step to refactoring to MVP is making a clear separation between the Presenter's responsibilities and what are the View's responsibilities. As we've described the MVP pattern, the Presenter is responsible for direct access to the Domain (Model), decoupling the View from the Model, transforming data into something the view can consume directly, and translating input from the user into updates to the Model. The View is then responsible for displaying information to the user and accepting their input and passing it along to the Presenter.

Where we left off with `ViewInvoiceForm` was that we changed the constructor to accept a parameter for an `IInvoiceRepository` object (a Domain object) so that the form could get the data for the invoice based on an invoice ID. The form as it stands is coupled to two Domain concepts: the Invoice Repository and the Invoice. Although we're more loosely coupled to the Invoice Repository via `IInvoiceRepository`; we're too coupled for MVP.

To start, we need to decouple the View from the `Invoice` class. To do this, we'll create a **Data Transfer Object (DTO)** class `InvoiceLineItemDTO` to encapsulate line item data.



Data Transfer Object (DTO) is a design pattern to encapsulate data that needs to be transferred from one context to another and does not implement any behavior.

Our `InvoiceLineItemDTO` class is very similar to our `InvoiceLineItem` class; but lives in the same namespace as the View:

```
namespace InvoicingFrontEnd
{
    /// <summary>
    /// Data Transfer Object for InvoiceLineItem objects
    /// </summary>
    public class InvoiceLineItemDTO
    {
        public float Quantity { get; set; }
        public float Price { get; set; }
        public float Discount { get; set; }
        public string Description { get; set; }
        public ITaxRate TaxRate1 { get; set; }
        public ITaxRate TaxRate2 { get; set; }
        public InvoiceLineItemDTO(
            InvoiceLineItem invoiceLineItem)
        {
            Quantity = invoiceLineItem.Quantity;
            Price = invoiceLineItem.Price;
            Discount = invoiceLineItem.Discount;
            Description = invoiceLineItem.Description;
            TaxRate1 = invoiceLineItem.TaxRate1;
            TaxRate2 = invoiceLineItem.TaxRate2;
        }
        public InvoiceLineItemDTO()
        {
        }
        public InvoiceLineItem ToInvoiceLineItem()
        {
            return new InvoiceLineItem()
            {
                Description = Description,
                Discount = Discount,
                Price = Price,
                Quantity = Quantity,
            }
        }
    }
}
```

```

        TaxRate1 = TaxRate1,
        TaxRate2 = TaxRate2
    };
}
}
}

```

We then need to start providing an interface for the Presenter to be able to deal with the `Form` object, without having to deal with it as a `Form` object (that is, "decouple" it from `Form`). This is where we implement the adapter pattern. Our `ViewInvoiceForm` class derives from `Form`, but we want our Presenter to communicate with it through a different interface—which we'll call `IEditInvoiceView`:

```

public interface IEditInvoiceView
{
    ICollection<InvoiceLineItemDTO> InvoiceLineItemDTOs
    { get; }
    DateTime Date { get; set; }
    string Title { get; set; }
    event EventHandler<EventArgs> DataUpdated;
    event EventHandler<EventArgs> ShowPreview;
}

```

This interface uses nothing from our Model and uses nothing specific for a `Form` object decoupling the View from the Model and decoupling the Presenter from the implementation details of the View (the fact that it derives from `Form`). `InvoiceLineItemDTOs` has the line item data for the invoice, and the rest of the invoice data is stored in the `Date` and `Title` properties. The Presenter will be informed of changes that need to be persisted back to the Model via the `DataUpdated` event—at which point the event handler will query the `InvoiceLineItemDTO` data and `Date` and `Title`. The next step is to have `ViewInvoiceForm` implement `IEditInvoiceView`.

To support `IEditInvoiceView`, modify `Populate` to use `InvoiceLineItemDTOs` instead of the invoice field's `LineItems` property. Our current implementation of the `InvoicePreviewFormFactory` is coupled directly to some of our domain classes; so, we'll create a `ShowPreview` event that will be raised in the `previewButton_Click` method instead of calling `InvoicePreviewFormFactory.Create`. Now that the invoice field is no longer needed, we'll remove that. To complete the refactoring, add explicit processing of the **OK** button click, so that the `DataUpdated` event can be raised.

The results of our refactoring effort will look something similar to the following:

```

/// <summary>
/// Read-only view of a form.

```

```
/// </summary>
public partial class ViewInvoiceForm
    : Form, IEditInvoiceView
{
    public ViewInvoiceForm()
    {
        InitializeComponent();
    }

    private void Populate()
    {
        dateTimePicker.Value = Date;
        foreach (var item in InvoiceLineItemDTOs)
        {
            lineItemsListView.Items.Add(
                new ListViewItem(
                    new string[]
                    {
                        item.Price.ToString(),
                        item.Discount.ToString(),
                        item.Quantity.ToString(),
                        item.Description }));
        }
    }

    private void previewButton_Click(object sender,
        EventArgs e)
    {
        ShowPreview(this, EventArgs.Empty);
    }

    List<InvoiceLineItemDTO> invoiceLineItemDTOs =
        new List<InvoiceLineItemDTO>();

    public ICollection<InvoiceLineItemDTO> InvoiceLineItemDTOs
    {
        get { return invoiceLineItemDTOs; }
    }

    public DateTime Date { get; set; }
    public string Title { get; set; }

    public event EventHandler<EventArgs> DataUpdated;
    public event EventHandler<EventArgs> ShowPreview;

    private void okButton_Click(object sender, EventArgs e)
    {
        DataUpdated(this, EventArgs.Empty);
        DialogResult = System.Windows.Forms.DialogResult.OK;
    }
}
```

```
        Close();
    }

    private void ViewInvoiceForm_Load(object sender,
        EventArgs e)
    {
        Populate();
    }
}
```

This leaves us with creating a Presenter to access the Model and populate the View. We'll continue the refactoring by creating a Presenter class

EditInvoicePresenterClass:

```
/// <summary>
/// Presenter to encapsulate business logic
/// relating to editing/viewing invoices
/// </summary>
class EditInvoicePresenter
{
    private IEditInvoiceView view;

    public EditInvoicePresenter(IEditInvoiceView view)
    {
        this.view = view;
    }

    public void Start(Invoice invoice)
    {
        foreach (var invoiceLineItem in invoice.LineItems)
        {
            view.InvoiceLineItemDTOs.Add(
                new InvoiceLineItemDTO(invoiceLineItem));
        }
        view.Title = invoice.Title;
        view.Date = invoice.Date;
        view.DataUpdated += view_DataUpdated;
        view.ShowPreview += view_ShowPreview;
    }

    void view_ShowPreview(object sender, System.EventArgs e)
    {
        List<InvoiceLineItem> lineItems =
            new List<InvoiceLineItem>(
                view.InvoiceLineItemDTOs.Count);
        foreach (var invoiceLineItemDTO in
            view.InvoiceLineItemDTOs)
```



```
    {
        lineItems.Add(
            invoiceLineItemDTO.ToInvoiceLineItem());
    }

    using (Form form = InvoicePreviewFormFactory.Create(
        view.Date, view.Title, lineItems))
    {
        form.ShowDialog((IWin32Window)view);
    }
}

public bool IsDirty { get; private set; }

void view_DataUpdated(object sender, System.EventArgs e)
{
    IsDirty = true;
}

public Invoice GetInvoice()
{
    List<InvoiceLineItem> invoiceLineItems =
        new List<InvoiceLineItem>();
    foreach (InvoiceLineItemDTO invoiceLineItemDTO in
        view.InvoiceLineItemDTOs)
    {
        invoiceLineItems.Add(
            invoiceLineItemDTO.ToInvoiceLineItem());
    }
    return new Invoice(view.Title, invoiceLineItems,
        view.Date);
}
}
```

In this class we have a constructor that initializes the presenter based on the View. The `Start` method, which is called when we're about to start the process of presenting the View, initializes the View and wires up the event handlers. The `ShowPreview` handler instantiates an `InvoicePreviewForm` object with the appropriate data and shows the form. The `DataUpdated` event handler updates the `IsDirty` flag. And the `GetInvoice` method is a convenient method to create a Domain `Invoice` object based on the View's data.

This leaves us with replacing the current instantiation and use of the `ViewInvoiceForm` class to include the Presenter. This can be done as follows:

```
using (var form = new ViewInvoiceForm())
{
    var presenter = new EditInvoicePresenter(form);
```

```
presenter.Start(invoice);  
form.ShowDialog(this);  
}
```

Façade pattern

The adapter pattern adapts the interface of one class into a more acceptable interface. Besides an interface that makes more sense in a specific concept, the adapter pattern can decouple unrelated classes from one another, provide a cohesive encapsulation, and make otherwise implicit concepts explicit. The façade pattern is very similar to the adapter pattern; but creates a new interface to wrap several interfaces. A façade pattern implementation has very little logic of its own; it simply manages several other objects and group specific interactions with those objects into methods whose name matches the intention or final goal of that interaction and generally contains no state of its own. Façade pattern implementations are often implementing as Singletons or used as if they were Singletons.

The façade pattern specifically makes particular interactions explicit (that may otherwise be implicit as blocks of code within another class) through their grouping within a method of the façade pattern implementation.

Domain-driven design includes the concept of Services. Services wrap operations that are important to a software system but don't naturally fit within the responsibilities of a specific class within the Domain. They may or may not include business logic or rules. Services that don't include business logic or rules are generally considered infrastructure services or application services. Services that do contain business logic or rules can be thought of as domain services. Services generally have no real state of their own and manage collaboration of other types. Services are excellent examples of façade pattern implementations.

The façade pattern is very useful in implementing an external interface for external clients to use. Generally, external interfaces make use of a particular protocol or framework to manage the communication between the client and the supplier. These protocols and frameworks have specific requirements for that communication. Examples of these restrictions may include things like type of data or format of the data.

Detecting the need for façade

Implementing the façade pattern can improve maintainability of code by making the concepts explicit. Code that's difficult to maintain because understanding it depends on reading inline comments can often be improved by implementing a façade around that code. The code in question usually requires many comments because it doesn't really apply to the class it's contained within.

Often, improving maintainability of code of this nature can simply be made explicit methods of the class it currently resides within. This is perfectly acceptable in many circumstances – after all, we're still taking functioning code and refactoring it. So, maintainability unto itself may not be a sufficient motivation to use a façade implementation.

Refactoring to the façade pattern

Unfortunately, the evolution of a code-base doesn't necessarily evolve from one particular pattern or principle to another. As a result, we'll return to a concept that we have introduced previously: the Data Access Layer. Generally, a Data Access Layer is a façade over use of data access subsystems. Although the Data Access Layer that we introduced in a prior chapter already effectively implemented a façade, we'll see how to refactor potential code to that data access class.

Another common starting point of refactoring to a Data Access Layer (and definitely one of the drawbacks of not having a Data Access Layer) is to have data access code to persist an entity within the entity itself. There are varying degrees of data access coupling within entities. I'll detail the worst-case scenario of having all the entity-specific data access code within the entity mostly for readability rather than an industry commentary (although, it does make refactoring to a Data Access Layer very persuasive).

```
public void Load(string connectionString)
{
    using (SqlConnection connection =
        new SqlConnection(connectionString))
    {
        connection.Open();
        using (DataSet invoiceDataSet = new DataSet("Invoice"))
        {
            using (SqlCommand command =
                new SqlCommand("SELECT Id, Date, Title," +
                    " Status FROM Invoice WHERE (Id = @ID)",
                    connection))
            {
                command.Parameters.Add("@ID",
                    SqlDbType.UniqueIdentifier);
                command.Parameters["@ID"].Value = Id;
                using (SqlDataAdapter invoiceDataAdapter =
                    new SqlDataAdapter())
                {
                    invoiceDataAdapter.TableMappings.Add(
                        "Table", "Invoices");
                }
            }
        }
    }
}
```

```
        command.CommandType = CommandType.Text;
        invoiceDataAdapter.SelectCommand = command;
        invoiceDataAdapter.Fill(invoiceDataSet);
    }
}
DataRow dataRow =
    invoiceDataSet.Tables["Invoices"].Rows[0];
Title = (String)dataRow["Title"];
}
using (DataSet lineItemDataSet =
    new DataSet("LineItems"))
{
    using (SqlCommand command =
        new SqlCommand("SELECT InvoiceId, Price, " +
            "Discount,Quantity, Description, " +
            "TaxRate1, TaxRate2 FROM LineItem " +
            "WHERE (InvoiceId = @ID)",
            connection))
    {
        command.Parameters.Add("@ID",
            SqlDbType.UniqueIdentifier);
        command.Parameters["@ID"].Value = Id;
        using (SqlDataAdapter lineItemDataAdapter
            = new SqlDataAdapter())
        {
            lineItemDataAdapter.TableMappings.Add(
                "Table", "LineItems");

            command.CommandType = CommandType.Text;
            lineItemDataAdapter.SelectCommand = command;
            lineItemDataAdapter.Fill(lineItemDataSet);
        }
    }
}
List<InvoiceLineItem> lineItems =
    new List<InvoiceLineItem>(
        lineItemDataSet.Tables["LineItems"]
            .Rows.Count);
foreach (DataRow row in
    lineItemDataSet.Tables["LineItems"].Rows)
{
    InvoiceLineItem invoiceLineItem =
        new InvoiceLineItem()
        {
            Description = row["Description"] as String,
```

```
        Discount = (float)(double)row["Discount"],
        Price = (float)(Decimal)row["Price"],
        Quantity = (int)row["Quantity"],
    };
    if (row["TaxRate1"] != DBNull.Value)
    {
        if ((String)row["TaxRate1"] == "GST")
        {
            invoiceLineItem.TaxRate1 =new FederalGST();
        }
        else if ((String)row["TaxRate1"] == "PST")
        {
            invoiceLineItem.TaxRate1 =new OntarioPST();
        }
    }
    if (row["TaxRate2"] != DBNull.Value)
    {
        if ((String)row["TaxRate2"] == "GST")
        {
            invoiceLineItem.TaxRate2 =new FederalGST();
        }
        else if ((String)row["TaxRate2"] == "PST")
        {
            invoiceLineItem.TaxRate2 =new OntarioPST();
        }
    }
    lineItems.Add(invoiceLineItem);
}
}
}
```

This single Load method in the Invoice class populates the Invoice class's Title, and LineItems properties based on data in the database. It's missing error and missing data checking – which I'll leave as an exercise for the reader so we can focus more on the actual refactoring process. It's expected to be used like this:

```
Invoice invoice = new Invoice(guid);
invoice.Load(Properties.Settings.Default.ConnectionString);
```

The module-level design details of a Data Access Layer are generally project or team specific. I generally tend towards putting a Data Access Layer into its own assembly to make enforcing and detecting rules like no cycles and separation by interfaces easier to enforce and detect.

Visual Studio 2010 Layer Diagrams



In Visual Studio 2010 Premium and Ultimate editions, there is a new feature called Layer Diagrams. You can declare your architectural intentions with a layer diagram by defining what assemblies constitute layers and the intended dependencies between them. You can then use **Validate Architecture** to ensure that the dependencies in code don't violate the expected dependencies between layers.

So, starting this refactoring requires creating a new class library project in our solution to house our Data Access Layer. In our example, we'll call it `Invoicing.Data` so our default namespace will be `Invoicing.Data`.

Once we have the project for our Data Access Layer, we can begin adding code to it. We want to keep our Data Layer separated from the rest of our system through an abstraction. We'll choose to implement this abstraction through an interface. So, the next step is to create our interface. Our interface will be named, creatively, `IDataAccess`. To start with, it will contain `LoadInvoice` and `LoadInvoiceLineItems` method. Following is the definition of our interface:

```
namespace Invoicing.Data
{
    public interface IDataAccess
    {
        DataSet LoadInvoice(Guid invoiceId);
        DataSet LoadInvoiceLineItems(Guid invoiceId);
    }
}
```

Our data access class will be named `DataAccess`; so, we'll add a new class to our `Invoicing.Data` project with this name, derive it from `IDataAccess`, and use the **Implement Interface** menu item to add stub methods for `IDataAccess`. This results in the following:

```
namespace Invoicing.Data
{
    public class DataAccess : IDataAccess
    {
        public DataSet LoadInvoice(Guid invoiceId)
        {
            throw new NotImplementedException();
        }

        public DataSet LoadInvoiceLineItems(Guid invoiceId)
        {
```

```
        throw new NotImplementedException();
    }
}
}
```

We'll effectively perform an Extract Method then a Move Method refactoring to move code from the `Invoice.Load` method to each of the two `DataAccess` methods, resulting in the following:

```
public class DataAccess : IDataAccess
{
    public DataSet LoadInvoiceLineItems(Guid invoiceId)
    {
        using (SqlConnection connection =
            new SqlConnection(connectionString))
        {
            connection.Open();
            DataSet lineItemDataSet = new DataSet("LineItems");
            using (SqlCommand command =
                new SqlCommand("SELECT InvoiceId, Price, " +
                    "Discount, Quantity, Description, " +
                    "TaxRate1, TaxRate2 FROM LineItem " +
                    "WHERE (InvoiceId = @ID)", connection))
            {
                command.Parameters.Add("@ID",
                    SqlDbType.UniqueIdentifier);
                command.Parameters["@ID"].Value = invoiceId;
                using (SqlDataAdapter lineItemDataAdapter
                    = new SqlDataAdapter())
                {
                    lineItemDataAdapter.TableMappings.Add(
                        "Table", "LineItems");

                    command.CommandType = CommandType.Text;
                    lineItemDataAdapter.SelectCommand = command;

                    lineItemDataAdapter.Fill(lineItemDataSet);
                }
            }
            return lineItemDataSet;
        }
    }

    public DataSet LoadInvoice(Guid invoiceId)
    {
        using (SqlConnection connection =
            new SqlConnection(connectionString))
```

```

    {
        connection.Open();
        DataSet invoiceDataSet = new DataSet("Invoice");
        using (SqlCommand command =
            new SqlCommand("SELECT Id, Date, Title," +
                " Status FROM Invoice WHERE (Id = @ID)",
                connection))
        {
            command.Parameters.Add("@ID",
                SqlDbType.UniqueIdentifier);
            command.Parameters["@ID"].Value = invoiceId;
            using (SqlDataAdapter invoiceDataAdapter =
                new SqlDataAdapter())
            {
                invoiceDataAdapter.TableMappings.Add(
                    "Table", "Invoices");
                command.CommandType = CommandType.Text;
                invoiceDataAdapter.SelectCommand = command;
                invoiceDataAdapter.Fill(invoiceDataSet);
            }
        }
        return invoiceDataSet;
    }
}

```

We'll also separate the connection string into a field that get's initialized in the constructor, so we don't have to pass it along every time we want to load data:

```

private string connectionString;
public DataAccess(string connectionString)
{
    this.connectionString = connectionString;
}

```

Code that invoked the Load method changes to the following:

```

IDataAccess dataAccess = new DataAccess(Properties.Settings.Default.
    ConnectionString);
DataSet lineItemDataSet = dataAccess.LoadInvoiceLineItems(invoiceId);
List<InvoiceLineItem> lineItems =
    new List<InvoiceLineItem>(
        lineItemDataSet.Tables["LineItems"]
        .Rows.Count);
foreach (DataRow row in

```



```
lineItemDataSet.Tables["LineItems"].Rows)
{
    InvoiceLineItem invoiceLineItem =
        new InvoiceLineItem()
        {
            Description = row["Description"] as String,
            Discount = (float)(double)row["Discount"],
            Price = (float)(Decimal)row["Price"],
            Quantity = (int)row["Quantity"],
        };
    if (row["TaxRate1"] != DBNull.Value)
    {
        if ((String)row["TaxRate1"] == "GST")
        {
            invoiceLineItem.TaxRate1 =new FederalGST();
        }
        else if ((String)row["TaxRate1"] == "PST")
        {
            invoiceLineItem.TaxRate1 =new OntarioPST();
        }
    }
    if (row["TaxRate2"] != DBNull.Value)
    {
        if ((String)row["TaxRate2"] == "GST")
        {
            invoiceLineItem.TaxRate2 =new FederalGST();
        }
        else if ((String)row["TaxRate2"] == "PST")
        {
            invoiceLineItem.TaxRate2 =new OntarioPST();
        }
    }
    lineItems.Add(invoiceLineItem);
}

DataSet invoiceDataSet = dataAccess.LoadInvoice(invoiceId);
DataRow dataRow =
    invoiceDataSet.Tables["Invoices"].Rows[0];
Invoice invoice = new Invoice(
    (String)dataRow["Title"],
    lineItems,
    (DateTime)dataRow["Date"])
{
    Id = (Guid)dataRow["Id"]
};
```

This completes this particular Refactor to Façade. This particular refactoring often includes a refactoring to a Repository implementation to move the code that processes the `DataSet` object into a Repository implementation to avoid having to repeat it throughout the system. See Chapter 8 for details on refactoring to a Repository implementation.

This Data Access façade implementation makes use of several classes from a couple of subsystems: `System.Data.SqlClient.SqlConnection`, `System.Data.DataSet`, `System.Data.SqlClient.SqlCommand`, `System.Data.SqlClient.SqlDataAdapter`, and various other types indirectly from these subsystems. Conceptually, what we want to be able to do (load invoice data) is simple; but we've got many lines of code to use and manage types from these subsystems. This is much more complexity than we want stuck in our Invoice class. Our `DataAccess` façade class hides all that complexity within it. It's much easier to read with domain-specific terminology like "LoadInvoice", plus it reduces the dependencies without our Invoice class. This makes our Invoice class much more flexible.

Proxy pattern

The proxy pattern is defined as having the following intent: "Provide a surrogate or placeholder for another object to control access to it". We've seen the Adapter and how it can be used to manage access to another. It may seem that the proxy pattern is a type of Adapter Pattern; but the proxy pattern controls access to a particular object while providing an identical interface to that object. A proxy pattern must also provide additional functionality; but effectively can be used in place of the other — that is, it doesn't "adapt" the interface. Depending on the type of the proxy pattern implementation, it may actually also be an adapter or façade pattern implementation in the strictest sense because it adapts interfaces that would otherwise not be available to the calling code or composes several objects to implement the interface.

There are generally four main types of proxy pattern implementations within .NET. The first is the remote proxy. The remote proxy implementation is a local depiction of an object that executes in a remote address space.

The next is the virtual proxy. A virtual proxy implementation attempts to implement a placeholder for an expensive creation object so that the creation of the object needs to occur until its first use (otherwise, until it's really needed). This is an example of Lazy Initialization.

Another implementation type is a protection proxy. This implementation controls access to the contained object based on permissions or access rights. This control could work in two ways. One would be that calling code that should not have access to the contained object could be denied. Or, the proxy could encapsulate elevation of permissions to effectively grant access to an object or resource that the calling code would otherwise not have access to.

The final implementation type is the smart reference. The smart reference is often a wrapper for a pointer in some native languages; but is basically a proxy that adds functionality. Types of the smart reference proxies include smart pointers and decorators.

Detecting need for proxy

From our descriptions of the types of Proxy Pattern implementations, you've likely thought of some places in code you've worked on where the Proxy Pattern would fit quite nicely. We'll detail some ways of detecting where using Proxy Pattern may be appropriate.

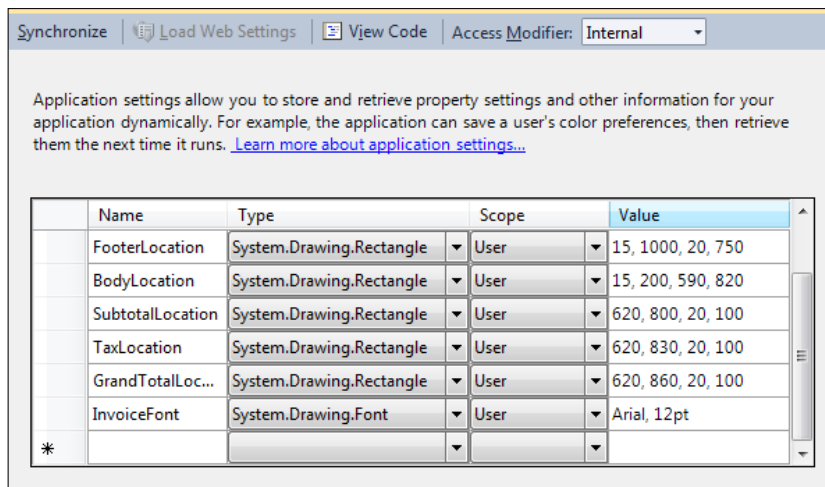
Although the proxy pattern covers circumstances where you want to wrap another object that takes a long time to perform operations, it has been my experience that dealing with objects or resources whose operations have the potential to be lengthy should not use the proxy pattern. Using the proxy pattern to encapsulate lengthy operations just delays the lengthy operation—you still end up blocking the current thread waiting for the lengthy operation. In cases where you want to encapsulate an object whose operations have the potential to be lengthy, the Task Parallel Library in .NET 4.0 provides much better support for parallelization of operations.

When you're looking to add functionality to an existing object without having to modify the object, prefer the decorator pattern over the proxy pattern.

Refactoring to proxy

I generally try to use Interface-based design when implementing the proxy pattern. This abstracts the type of the proxy from the use of the proxy. This helps with testability, maintainability, and so on; but also means the type of proxy can be determined at runtime (that is, through a Dependency Injection container). Using interface-based design also means that implementing a proxy is much easier.

For our example, let's show implementing a Virtual Proxy. Let's say that our `RenderReadableInvoiceService` loaded default coordinates and the font from user preferences. Our `Settings.settings` in our Visual Studio project may look something like the following:



With this particular implementation, the construction of the `InvoiceRenderingService` would be like the following:

```
public InvoiceRenderingService ()
{
    HeaderLocation =
        Properties.Settings.Default.InvoiceHeaderLocation;
    FooterLocation =
        Properties.Settings.Default.FooterLocation;
    BodyLocation = Properties.Settings.Default.BodyLocation;
    SubtotalLocation =
        Properties.Settings.Default.SubtotalLocation;
    TaxLocation = Properties.Settings.Default.TaxLocation;
    GrandTotalLocation =
        Properties.Settings.Default.GrandTotalLocation;
    Font = Properties.Settings.Default.InvoiceFont;
}
```

With this particular implementation, we incur "expensive" calls to `Properties.Settings.Default` that may result in accessing the local hard drive or a remote share (depending on how Windows was deployed). If we wanted to defer that until it was absolutely necessary (thus potentially avoiding it altogether) we could implement a virtual proxy invoice rendering service and implement Lazy Initialization.

Since we're dealing with a class that already uses Interface-Based Design, we don't need to create a new interface because `InvoiceRenderingService` implements `IInvoiceRenderingService`. If we were refactoring a class that didn't already implement an applicable interface, we would start with an Extract Interface refactoring.

To begin refactoring to virtual proxy with `InvoiceRenderingService`, we'll create a new class named `InvoiceRenderingServiceVirtualProxy`. We'll derive it from `IInvoiceRenderingService` and use **Implement Interface/Implement Interface** to have Visual Studio create the methods and properties for us (in this case, just one method `RenderReadableInvoice`). Since we're implementing a proxy, we need a field to store a reference to the object we're delegating to, so we'll add a field of type `IInvoiceRenderingService` named `realSubject`. To continue the refactoring, we'll implement the `RenderReadableInvoice` so that if `realSubject` is null, it assigns a new instance of `InvoiceRenderingService` to it before calling `RenderReadableInvoice` on the `realSubject` object. That implementation of `InvoiceRenderingServiceVirtualProxy` might look something like the following:

```
public class InvoiceRenderingServiceVirtualProxy
    : IInvoiceRenderingService
{
    private IInvoiceRenderingService realSubject;
    private Action<Invoice, Graphics>
        RenderReadableInvoiceProxy;
    public InvoiceRenderingServiceVirtualProxy()
    {
        RenderReadableInvoiceProxy = RenderReadableInvoiceImpl;
    }
    private void RenderReadableInvoiceImpl(Invoice invoice,
        Graphics graphics)
    {
        realSubject = new InvoiceRenderingService();
        RenderReadableInvoiceProxy =
            realSubject.RenderReadableInvoice;
        RenderReadableInvoiceProxy(invoice, graphics);
    }
    public void RenderReadableInvoice(Invoice invoice,
        Graphics graphics)
    {
        RenderReadableInvoiceProxy(invoice, graphics);
    }
}
```

With this particular implementation, I've chosen to use delegates to implement the detection of `realSubject` being null. Initialization of `InvoiceRenderingServiceProxy` initializes the delegate `ReaderReadableInvoiceProxy` to a method that instantiates an `InvoiceRenderingService` object and assigns it to `realSubject`, assigns the `ReaderReadableInvoice` method of that new instance to the `RenderReadableInvoiceProxy` delegate, then re-invokes the delegate (and thus the `realSubject.RenderReadlbeInvoice` method). Future invocations will simply immediately invoke the `realSubject.RenderReadlbeInvoice` method. This saves us from having to test the `realSubject` object for null on every invocation of `ReaderReadableInvoiceProxy.RenderReadlbeInvoice`.

At this point, we have a class that can be used in place of `InvoiceRenderingService`. Depending on how `InvoiceRenderingService` is used determines how we complete the refactoring. If `InvoiceRenderingService` is instantiated directly, we need to replace those instantiations with an instantiation of `InvoiceRenderingServiceVirtualProxy`. This could be as simple as changing the following:

```
IInvoiceRenderingService invoiceRenderingService =
    new InvoiceRenderingService();
```

to this:

```
IInvoiceRenderingService invoiceRenderingService =
    new InvoiceRenderingServiceVirtualProxy();
```

If that original use of the `InvoiceRenderingService` isn't used as an `IInvoiceRenderingService` reference, then use of `InvoiceRenderingService` references will have to be changed to `IInvoiceRenderingService` references.

If instantiation of the `InvoiceRenderingService` object is done through a Dependency Injection Container then completing the refactoring is just a matter of configuring the container to map `InvoiceRenderingServiceVirtualProxy` to `IInvoiceRenderingService` instead of `InvoiceRenderingService`. With Unity `app.config` configuration, this means changing something like the following:

```
<type type="IInvoiceRenderingService"
      mapTo="InvoiceRenderingService"/>
```

to this:

```
<type type="IInvoiceRenderingService"
      mapTo="InvoiceRenderingServiceVirtualProxy"/>
```

After creating an alias for `InvoiceRenderServiceVirtualProxy` in the `<typeAliases>` element:

```
<typeAlias alias="InvoiceRenderingServiceVirtualProxy"
  type="InvoicingFrontEnd.InvoiceRenderingServiceVirtualProxy
  , InvoicingFrontEnd"/>
```

Object/Relational Mapping

Object/Relation Mapping (ORM) is the use of an ORM framework to map objects (or more specifically, classes) to entities in a relational database (table, view, stored procedure, and so on). More often than not, this is a class-to-table mapping; but this could also be a mapping to views, functions, stored procedures, queries, and so on.

An ORM framework frees you from the need to design, write, test, and evolve code that performs direct data-access to an RDBMS. One important side-effect of this is the decoupling of the application from a specific relational database implementation. Assuming the ORM framework supports more than one database brand, making use of an ORM framework means moving from one database to another is that much easier.

As we've seen earlier in this chapter, one possible implementation to load data from the database is just a matter of populating `DataSet` objects and processing those `DataSet` objects. There's no domain-specific value in this code. It's conceptually simple code, but lengthy, full of SQL, hard to maintain, and fragile. It's basically the same code with different SQL queries or statements – the only difference being how many columns to process and their names. This code is tedious and time-consuming to write and not really the value proposition of our software system.

When data access code is originally written, the database structure is usually simple. Concepts are simple and there's usually a one-to-one relationship of properties to columns and classes to tables. But, object-oriented design is based on software engineering techniques and relational databases are based on mathematical techniques like set theory. As a database grows, performance and storage can become bottlenecks. Addressing these issues in a relational database is approached in ways that don't match well with how we design object-oriented software systems. Normalization in a database, for example, is about maximizing resources in the database, not for making the data conceptually easier to read. Objects, on the other hand, are an attempt to model real-world concepts. This disparity in the two ways of modeling and processing data is called the **Object-Relational Impedance Mismatch** or **Impedance Mismatch** for short. For example, a `Manager` class may derive from a `Person` class. This relationship can't be modeled explicitly in a relational database. It's this mismatch that takes time to alleviate and causes flexibility, maintainability, and robustness issues.



Object-Relational Impedance Mismatch is a term to describe the divergence between the relational set theory used in RDBMSs and object-oriented design. It's an attempt to recognize the side-effects of using RDBMSs to store and load data from and to object-oriented code.

Object/Relational Mappers free you from having to write this code. Instead of writing lower-level code that actually uses database-specific classes, you define which of your classes map to entities in your database (or databases).

I've often heard the argument that projects rarely change database brands and making use of an ORM framework simply adds complexity that isn't needed. Refactoring to an ORM framework requires a very specific need. The fact that the database *might* change or even that the database *needs* to change isn't always impetus enough to refactor to use an ORM framework. I've worked on several projects where the database brand or the database version needed to change and that caused undue modifications to data access code. This could have been avoided if an ORM was used.

One less-than-obscure way of dealing with data access is to generate the data access code from the database schema. This code generation (or codegen) technique does not address the Impedance Mismatch—it simply mirrors the database structure with Value Objects and provides code to populate their fields and store the values of the fields in a database. The developer still has to manually deal with areas of mismatch.

Problems with code generation

If you're working in a software system that is primarily a **CRUD Interface**, code generation of a data-access layer is more than likely perfectly acceptable. For the most part, the complexity in complex software systems comes from being business-logic heavy. The need to refactor stems from that complexity, and the expectation of this book is that readers are looking at ways of managing that business logic and are likely to encounter the problems with code generation of a data access layer.



CRUD Interface is a software system that provides little or no business logic and primarily simply provides a way to Create, Remove, Update, and Delete data within the system. A software system of this nature would normally contain no business logic or domain layer and only user interface and data access layers.

In a software system heavy with business logic, generation of business logic code is impossible based upon database schema. The resulting lightweight classes that do get generated are obviously missing business logic the system needs. Code generated from a database schema that can (and will) change will have any manual changes made to it lost when it is re-generated. This means the business logic must reside somewhere else. This leads either to putting the business logic effectively into the "user interface layer" (which may not be a tangible layer but the de facto "other" layer) or creating a Domain Layer that is responsible for encapsulating access to these generated classes. So, the problem is that the code exists and is easy to edit and cause problems.

Code Generation has the implicit promise of being easy. In complex systems that rely on much business logic, that appearance of ease might make generating business logic persuasive. Business logic is application-specific. It's really hard to make general-purpose code generation that also generates correct business logic. This means that when a code generator does generate business logic there's lots of testing, tweaking, and maintenance that goes along with it. The value you're attempting to add is the software system, not the code generator. Hand-coded code generators often result in a lot of work to perfect; work that could be better spent on direct value to the end-user.

Detecting need to refactor to ORM

Refactoring to an ORM framework requires a bit of infrastructure in order to happen correctly. It's helpful to have some level of abstraction of the data access. Having a full-blown data-access layer is often very helpful in the process of refactoring to an ORM framework; but it's more important that the data-access layer be abstract. If the domain code is designed around some sort of relational logic or relational data-access, then refactoring the data-access layer at any level will be difficult.

An ORM framework effectively takes the place of your data-access layer. You may still want to have a data-access layer that is effectively a wrapper around use of the ORM framework.

As I pointed out earlier, the fact that database brand might, or even, will change, may not be a persuasive reason to refactor to using an ORM. If you had used an ORM, that transition may have been easier; but that's more a side-effect than a reason.

One area of value that an ORM brings is its ability to decouple the necessary SQL statements from your code which makes supporting large quantities of objects or database entities much easier. You must know the SQL statement, table name, or stored procedure name in order to map the data from the database to your business logic regardless of whether you're using an ORM, writing your data access code from scratch, or using code generation (that is, you have to choose what the generator uses). Code generation effectively decouples the SQL from *your* code; but embeds it into code within the software system. With an ORM if the SQL needs to change (for example, from a statement to a stored procedure) you simply change the configuration and not your code.

Refactoring to ORM

So, with the requirement that the software system needs to add persistence for many more entities, let's look at how we can refactor what we've done so far to use an ORM. So far, we have a distinct Data Access Layer to encapsulate our data access code, a distinct Domain Layer to encapsulate our business logic entities and their direct interaction with the Data Access Layer, and a distinct User Interface Layer. Direct interaction with the ORM (beyond configuration) will be encapsulated in the Domain Layer. Use of an ORM should not affect how the high-level layers like the User Interface Layer interact with the Domain Layer (after all, we're still going to load and store our domain entities). Use of the ORM simply becomes an implementation detail of the Domain Layer. We're effectively off-loading the data access work to the ORM—it becomes our Data Access Layer.

For this particular refactoring example, we'll show using NHibernate. There isn't one ORM that is used by everyone; but NHibernate has been around in the .NET community for the longest and has a good community around it. This book isn't about choosing an ORM and it isn't specifically about how to refactor to *every* ORM. So, we're picking one ORM and using it in the refactoring; the semantics of using an ORM should be similar, the specific APIs will change.

We'll work with NHibernate 2.1.2—as of writing it's the current version of NHibernate. We'll start with the assumption that it has been downloaded and "installed". The first step is to add references to our Visual Studio project to the NHibernate assemblies. The application assembly needs references to `NHibernate.dll`, `LinFu.DynamicProxy.dll`, `NHibernate.ByteCode.LinFu.dll` and our Domain Layer class library needs to reference `NHibernate.dll`. All these dlls can be found in the directory where NHibernate was installed. The `LinFu` dlls will be in a **Required_For_LazyLoading** directory.

The next step is to configure NHibernate. There are multiple ways of doing this. One way is via the `app.config` file. We've chosen the `app.config` configuration method for various other things prior to this, so we'll continue with this method. We first need to add to `app.config` a configuration section handler to the `app.config configSections` element:

```
<section name="hibernate-configuration"
  type="NHibernate.Cfg.ConfigurationSectionHandler,
  NHibernate"/>
```

We then need to add the section that this configuration section handler will handle to `app.config`:

```
<hibernate-configuration
  xmlns="urn:nhibernate-configuration-2.2">
  <session-factory>
    <property name="connection.provider">
      NHibernate.Connection.DriverConnectionProvider
    </property>
    <property name="connection.driver_class">
      NHibernate.Driver.SqlClientDriver
    </property>
    <property name="connection.connection_string">
      Data Source=(local)\sqlexpress;
      Initial Catalog=Invoices;
      Integrated Security=True;
      Pooling=False
    </property>
    <property name="dialect">
      NHibernate.Dialect.MsSql2008Dialect
    </property>
    <property name="show_sql">
      false
    </property>
    <property name="proxyfactory.factory_class">
      NHibernate.ByteCode.LinFu.ProxyFactoryFactory,
      NHibernate.ByteCode.LinFu
    </property>
  </session-factory>
</hibernate-configuration>
```

This section configures multiple things. First, it configures the connection provider. Here we're using the `NHibernate.Connection.DriverConnectionHandler` – which is the built-in class that handles creating and disposing of database connections (`IDbConnection` objects). Next, it configures the NHibernate database driver. Here we're using the built-in `SqlClientDriver` because we're using a SQL Server Express database. Next, it configures how we're talking with SQL server – the dialect. Here we're using `SQL Server 2008 Dialect` because we're using SQL Server 2008 Express Edition. Next, it configures to not trace the executed SQL statements. Finally, it configures what proxy factory to use. Here, we're configuring the `LinFu` dynamic proxy. The dynamic proxy implements the proxy pattern to deal with lazy loading of data. As part of the configuration of NHibernate we need to tell it how our domain entities map to data in the database. This is done through NHibernate mapping files.

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2"
    assembly="Invoicing.Domain"
    namespace="Invoicing.Domain">
  <class name="Invoice" lazy="false" table="Invoice">
    <id name="Id">
      <generator class="guid" />
    </id>
    <property name="Title" />
    <property name="Date" />
  </class>
</hibernate-mapping>
```

Let's look at individual parts of the mapping file to see what is being configured:

```
<class name="Invoice" lazy="false" table="Invoice">
```

This tells NHibernate that the name of our domain entity class is "Invoice" and that direct access to the data is done through the "Invoice" table. Data won't be lazy-loaded:

```
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2"
    assembly="Invoicing.Domain"
    namespace="Invoicing.Domain">
```

This tells NHibernate that the Invoice class is in the `Invoicing.Domain` namespace in the `Invoicing.Domain` assembly:

```
<id name="Id">
  <generator class="guid" />
</id>
```

This tells NHibernate that the identity of the `Invoice` object is stored in the `Id` property and that the identity of the invoice data is in the "Id" column. It also tells NHibernate to automatically generate new identities through its GUID generator:

```
<property name="Title" />
```

This tells NHibernate that the `Invoice` property `Title` is loaded and stored from/to the database through the "Title" column:

```
<property name="Date" />
```

This tells NHibernate that the `Invoice` property `Date` is loaded and stored from/to the database through the "Date" column.

The properties on our `Invoice` class match the corresponding column names in our `Invoice` table. This convention makes it easier to read code, mappings, and SQL queries. This isn't always the case (or even the policy). If class name and table name don't match (for example, the table is named "Invoices") the `class` element would look like this:

```
<class name="Invoice" lazy="false" table="Invoice">
```

If the `Invoice.Date` property name didn't match the corresponding column name (it's named "InvoiceDate" for example) then the `Date` property attribute would look like this:

```
<property name="Date" column="InvoiceDate" />
```

There are various other options available to most ORMs, which include defining what types to use for each property/column, whether to use specific SQL statements or stored procedures, and so on. This is by no means a tutorial on NHibernate, so we'll stick to the basics.

To continue the refactoring we need to initialize NHibernate. We're going to do this in our `Program.Main` method. Depending on your circumstances, you may need to do this elsewhere:

```
var configuration = new NHibernate.Cfg.Configuration();
configuration.Configure();
configuration.AddAssembly(
    typeof(Invoicing.Domain.Invoice).Assembly);
```

This code creates an NHibernate `Configuration` object, requests that it load the configuration (in our case, from `app.config`) with a call to `Configure()`, then we tell NHibernate what assembly contains our business entities (which is the assembly where `Invoice` is located).

Next, we need to update the `InvoiceRepository` class to use NHibernate instead of Data Access Layer. The way NHibernate works is that requests to load or save data are done through an `ISession` object. Our `InvoiceRepository` needs a reference to an `ISession` implementation. We'll choose to use dependency injection and change the constructor to initialize a new `ISession` field (`session`) based on a constructor argument. Next, we'll update the `Load` method to make use of the `session` field. To load an object from the database we simply tell the `ISession` reference what type to load and its ID. In this case, we use generics to tell it what type we're dealing with in the call to `ISession.Load<T>(Object id)`:

```
public class InvoiceRepository : IInvoiceRepository
{
    NHibernate.ISession session;

    public InvoiceRepository(NHibernate.ISession session)
    {
        this.session = session;
    }

    public Invoice Load(Guid invoiceId)
    {
        return session.Load<Invoice>(invoiceId);
    }
}
```

This code is much simpler than the code we had before. Conceptually, to support loading any new entity requires a repository class as simple as this.

To finalize the refactoring we need to instantiate an `ISession` reference and get it into our repository object when it is instantiated. Since we're now using a Dependency Injection container, we'll configure our container with an `ISession` object for it to use where we initialize our container:

```
using (IUnityContainer container =
    new UnityContainer())
{
    container.RegisterInstance(
        configuration.BuildSessionFactory().OpenSession());
}
```

Here we're asking the NHibernate configuration to build a `ISessionFactory` reference, then call `ISessionFactory.OpenSession()` to open a session to our configured database. The resulting `ISession` reference is registered with our Unity container so that whenever an object that depends on an `ISession` reference is created, it should always use this reference.

ORM sessions

The `ISession` manages a particular connection to a database. In RDBMS parlance this is similar to a transaction. The `ISession` deals in a single unit-of-work context or a single transaction. If two database operations needed to be within the same transaction (for example, one is dependent on the success of the other) then we'd need those operations to be performed on the same `ISession` reference.

NHibernate's use of `ISession` is not a unique NHibernate concept. Other ORMs have similar concepts; with Microsoft Entity Framework 4, it's an `ObjectContext` object and with LINQ to SQL, it's a `DataContext` object.

Summary

We've seen how structural patterns allow us to compose new objects from existing objects. This allows us to encapsulate functional within these new objects to make more abstract objects. These more abstract objects are easier to decouple the composing objects away from the rest of the system to make changes to them or the rest of the system easier to realize.

ORM Frameworks allow us to make the database more abstract. We've seen how using an ORM can make integration of the database easier, more flexible, and more decoupled. Having a more abstract data-access implementation makes a more cohesive application by making sure the nuances of a particular database brand are unlikely to influence the design and the semantics of our software system.

In the next chapter, we'll review the importance of unit-testing. We'll discuss how writing automated tests before, during, and after refactoring helps us understand the requirements imposed on our software system. It will also allow us to verify that those requirements are being met and validate that the parts of the system that have been refactored are still functioning as expected.

11

Ensuring Quality with Unit Testing

In the previous chapter, we discussed techniques for refactoring architecture structure. We discussed how we change the architectural structure to promote decoupling and decouple our code from external or legacy code.

In this chapter, we'll see how we can support the refactoring effort through the use of unit testing to validate that the refactored code has not changed external behavior.

We'll discuss various aspects of unit testing, as well as how to approach unit testing with Visual Studio®. We'll cover the following topics:

- Automated testing
- Unit tests
- Mocking
- Mocking frameworks
- Unit test frameworks
- Unit testing legacy code
- Test-driven development
- Third party refactoring tools

Change is not always good

Any change to existing code means it has the potential to change the external behavior of the system. When we refactor code, we explicitly intend not to change the external behavior of system. But how do we perform our refactorings while being reasonably comfortable that we haven't changed external behavior?

The first step to validating that external behavior hasn't been affected is to define the criteria by which we can validate that the external behavior hasn't changed.

Automated testing

Every developer does unit testing. Some developers write a bit of test code, maybe an independent project that uses the code to verify it in some way then promptly forgets about the project. Or even worse, they throw away that project. For the purposes of this text, when I use the term "testing", I mean "automated testing".

Test automation is the practice of using a testing framework to facilitate and execute tests. A test automation framework promotes the automatic execution of multiple tests. Generally these frameworks include some sort of Graphical User Interface that helps manage tests and their execution. Passing tests are "Green" and failing tests are "Red", which is where the "Red, Green, Refactor" mantra comes from.

Unit tests

If we're refactoring, there's a chance that what we want to refactor isn't currently under test. This means that if we do perform refactoring on the code, we'll have to manually test the system through the established user interfaces to verify that the code works. Realistically, this doesn't verify the *code*; this verifies that the *external behavior* hasn't changed. There could very well be a hidden problem in the code that won't manifest itself until the external behavior has been modified – distancing detection of the defect from when it was created. Our goal is to not affect external behavior when refactoring, so verification through the graphical user interface doesn't fully verify our changes and is time consuming and more prone to human error.

What we really want to do is unit test the code. The term "unit test" has become overloaded over the years. MSDN describes unit testing as taking:

...the smallest piece of testable software in the application, [isolating] from the remainder of the code, and [determining] whether it behaves exactly as [expected].

This smallest piece of software is generally at the method level – unit testing is effectively about ensuring each method behaves as expected. Originally, it meant to test an individual unit of code. "Unit test" has evolved to mean any sort of code-based automated test, tests that developers write and execute within the development process. With various available frameworks, the process of testing the graphical user interface can also be automated in a code-based test, but we won't focus on that.

It's not unusual for some software projects to have hundreds and thousands of individual unit tests. Given the granularity of some of the tests, it's also not unusual for the lines of code in the unit tests to outnumber the actual production code. This is expected.

At the lowest level, we want to perform true "unit-testing", we want to test individual units of code, independently, to verify that unit of code functions as expected – especially in the presence of refactoring. To independently test these units of code we often have to separate them from their dependant code. For example, if I want to verify the business logic to uniquely generate an entity ID, there's no real need for me to access the database to verify that code. That code to generate a unique ID may depend on a collection of IDs to fully verify the algorithm to generate a unique ID – but that collection of IDs, for the purposes of verification, doesn't need to come from a database. So, we want to separate out use of some dependencies like the database from some of our tests.

As we've seen in previous chapters, techniques for loosely-coupled design like Dependency Inversion and Dependency Injection allow for a composable design. This composable design aids in the flexibility and agility of our software system, but it also aids in unit testing.

Other testing


Useful and thorough information about all types of testing could easily reach enough information to take up several tomes. We're focusing on the developer task of refactoring, so we're limiting our coverage of testing to absolute essential developer testing: unit testing.

The fact that we're focusing on unit tests with regard to refactoring doesn't mean that other types of testing is neither useful nor needed. The fact that developers are performing unit tests doesn't preclude that they also need to perform a certain level of integration testing and the QA personnel are performing other levels of integration testing, user interface testing, user acceptance testing, system testing, and so on.

Integration testing is combining distinct modules in the system to verify that they interoperate exactly as expected. User interface testing is testing that the user interface is behaving exactly as expected. User acceptance testing is verifying that specific user requirements are being met – which could involve unit testing, integration testing, user interface testing, verifying non-functional requirements, and so on.

Mocking

Mocking is a general term that usually refers the substitution of **Test Doubles** for dependencies within a system under test that aren't the focus of the test. "Mocking" generally encompasses all types of test doubles, not just **Mock** test doubles.

 **Test Double** is any object that takes the place of a production object for testing purposes.

Mock is a type of **Test Double** that stands in for a production object whose behavior or attributes are directly used within the code under test and within the verification.

Test Doubles allow an automated test to gather the criteria by which the code is verified. Test Doubles allow isolation of the code under test. There are several different types of Test Doubles: **Mock**, **Dummy**, **Stub**, **Fake**, and **Spy**.

- **Dummy** is a type of Test Double that is only passed around within the test but not directly used by the test. "null" is an example of a dummy – use of "null" satisfies the code, but may not be necessary for verification.
- **Stub** is a type of Test Double that provides inputs to the test and may accept inputs from the test but does not use them. The inputs a Stub provides to the test are generally "canned".
- **Fake** is a type of Test Double that is used to substitute a production component for a test component. A Fake generally provides an alternate implementation of that production component that isn't suitable for production but useful for verification. Fakes are generally used for components with heavy integration dependencies that would otherwise make the test slow or heavily reliant on configuration.
- **Spy** is a type of Test Double that effectively records the actions performed on it. The recorded actions can then be used for verification. This is often used in behavioral-based – rather than state-based – testing.

Test doubles can be created manually, or they can be created automatically through the use of mocking frameworks. Frameworks like Rhino Mocks provide the ability to automatically create test doubles. Mocking framework generally rely on a loosely-coupled design so that the generated test doubles can be substituted for other objects based upon an interface.

Let's look at an example of writing a unit test in involving mocking. If we return to one of our decoupling examples – `InvoiceRepository` – we can now test the internals of `InvoiceRepository` without testing our **Data Access Layer (DAL)**. We would start by creating a test for the `InvoiceRepository.Load` method:

```
[TestClass()]
public class InvoiceRepositoryTest
{
    [TestMethod()]
    public void LoadTest()
    {
        DateTime expectedDate = DateTime.Now;
        IDataAccess dataAccess =
            new InvoiceRepositoryDataAccessStub(expectedDate);
        InvoiceRepository target = new
            InvoiceRepository(dataAccess);
        Guid invoiceId = Guid.NewGuid();

        Invoice actualInvoice = target.Load(invoiceId);

        Assert.AreEqual(expectedDate, actualInvoice.Date);
        Assert.AreEqual(invoiceId, actualInvoice.Id);
        Assert.AreEqual("Test", actualInvoice.Title);
        Assert.AreEqual(InvoiceStatus.Posted,
            actualInvoice.Status);
        Assert.AreEqual(1, actualInvoice.LineItems.Count());
        InvoiceLineItem actualLineItem =
            actualInvoice.LineItems.First();
        Assert.AreEqual("Description",
            actualLineItem.Description);
        Assert.AreEqual(1F, actualLineItem.Discount);
        Assert.AreEqual(2F, actualLineItem.Price);
        Assert.AreEqual(3F, actualLineItem.Quantity);
    }
}
```

Here, we're creating an instance of our repository passing it a **Stub** `IDataAccess` class. We then invoke the `Load` method and verify the various attributes of the resulting `Invoice` object. We, of course, don't have a class named `InvoiceRepositoryDataAccessStub`, so we'll have to create one. This class, for the purposes of this test, will look like this.

```
class InvoiceRepositoryDataAccessStub : IDataAccess
{
    private DateTime constantDate;

    public InvoiceRepositoryDataAccessStub(DateTime date)
    {
        constantDate = date;
    }

    public System.Data.DataSet LoadInvoice(Guid invoiceId)
    {
        DataSet invoiceDataSet = new DataSet("Invoice");
        DataTable invoiceTable =
            invoiceDataSet.Tables.Add("Invoices");
        DataColumn column = new DataColumn("Id",
            typeof(Guid));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Date", typeof(DateTime));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Title", typeof(String));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Status", typeof(int));
        invoiceTable.Columns.Add(column);
        DataRow invoiceRow =
            invoiceTable.NewRow();
        invoiceRow["Id"] = invoiceId;
        invoiceRow["Date"] = constantDate;
        invoiceRow["Status"] = InvoiceStatus.Posted;
        invoiceRow["Title"] = "Test";
        invoiceTable.Rows.Add(invoiceRow);
        return invoiceDataSet;
    }

    public System.Data.DataSet LoadInvoiceLineItems(
        Guid invoiceId)
    {
        DataSet lineItemDataSet = new DataSet("LineItem");
        DataTable lineItemTable =
            lineItemDataSet.Tables.Add("LineItems");
        DataColumn column =
```

```
        new DataColumn("InvoiceId", typeof(Guid));
lineItemTable.Columns.Add(column);
column = new DataColumn("Price", typeof(Decimal));
lineItemTable.Columns.Add(column);
column = new DataColumn("Quantity", typeof(int));
lineItemTable.Columns.Add(column);
column = new DataColumn("Discount", typeof(double));
lineItemTable.Columns.Add(column);
column = new DataColumn("Description", typeof(String));
lineItemTable.Columns.Add(column);
column = new DataColumn("TaxRate1", typeof(String));
lineItemTable.Columns.Add(column);
column = new DataColumn("TaxRate2", typeof(String));
lineItemTable.Columns.Add(column);

DataRow lineItemRow =
    lineItemDataSet.Tables["LineItems"].NewRow();

lineItemRow["InvoiceId"] = invoiceId;
lineItemRow["Discount"] = 1F;
lineItemRow["Price"] = 2F;
lineItemRow["Quantity"] = 3;
lineItemRow["Description"] = "Description";

lineItemTable.Rows.Add(lineItemRow);

return lineItemDataSet;
}

public void SaveInvoice(System.Data.DataSet dataSet)
{
    throw new NotImplementedException();
}
}
```

Here, we're manually creating `DataSet` object and populating rows with canned data that we're specifically checking for in the validation code within the test. It's worth noting that we haven't implemented `SaveInvoice` in this class. This is mostly because we haven't implemented this in the production code yet; but, in the case of testing `Load`, an exception would be thrown should it call `SaveInvoice`—adding more depth to the validation of the `Load` method, since it shouldn't be using `SaveInvoice` to load data.

In the `InvoiceRepositoryTest.LoadTest` method, we're specifically using the `InvoiceRepositoryDataAccessStub`. `InvoiceRepositoryDataAccessStub` is a Stub of and `IDataAccess` specifically for use with `InvoiceRepository`. If you recall, a Stub is a Test Double that substitutes for a production component but inputs canned data into the system under test. In our test, we're just checking for that canned data to verify that the `InvoiceRepository` called our `InvoiceRepositoryDataAccessStub` instance in the correct way.

Priorities

In a project with little or no unit tests, it can be overwhelming to begin refactoring the code. There can be the tendency to want to first establish unit tests for the entire code base before refactoring starts. This, of course, is linear thinking. An established code base has been verified to a certain extent. If it's been deployed, the code effectively "works". Attempting to unit test every line of code isn't going to change that fact.

It's when we start to change code that we want to verify that our change doesn't have an unwanted side-effect. To this effect, we want to prioritize unit-testing to avoid having unit-testing become the sole focus of the team. I find that the unit-testing priorities when starting out with unit-testing are the same as when a system has had unit tests for some time. The focus should be that any new code should have as much unit-testing code coverage as realistically possible and any code that needs to be refactored should have code coverage as high as realistically possible.

The priority here is to ensure that any new code is tested and verified, and accept the fact that existing code has been verified in its own way. If we're not planning on immediately changing certain parts of code, they don't need unit-tests and should be of lower priority.

Code coverage

Something that often goes hand-in-hand with unit testing is **Code Coverage**. The goal of code coverage is to get as close to 100% coverage as reasonably possible.



Code Coverage is the measure of the percentage of code that is executed (covered) by automated tests.

Code coverage is a metric generally used in teams that are performing unit tests on a good portion of their code. Just starting out with unit testing, code coverage is effectively anecdotal. It doesn't tell you much more than you are doing some unit tests.

One trap to get into as teams start approaching majority code coverage is to strive for 100% code coverage. This is both problematic and counterproductive. There is some code that is difficult to test and even harder to verify. The work involved to test this code is simply to increase code coverage percentages.

I prefer to view the code coverage delta over time. In other words, I concentrate on how the code coverage percentage changes (or doesn't). I want to ensure that it's not going down. If the code coverage percentage is really low (say 25%) then I may want to see it increasing, but not at the risk of supplanting other work.

Mocking frameworks

As we've seen so far, we've manually created test doubles in order to isolate the system under test. There are frameworks collectively called "Mocking Frameworks" that facilitate automatic creation of Stubs, Fakes, Mocks, and Dummies. We have a cursory look at a common subset of those frameworks now.

Rhino Mocks

Rhino Mocks is an open source mocking framework written by Ayende Rahien. To use Rhino Mocks with your test project, simply add a reference to `Rhino.Mocks.dll`.

```
[TestClass]
public class InvoiceRepositoryTest
{
    /// <summary>
    /// Creates canned invoice data
    /// </summary>
    /// <param name="invoiceId"></param>
    /// <param name="constantDate"></param>
    /// <returns>DataSet of invoice data</returns>
    private DataSet CreateInvoiceDataSet(Guid invoiceId,
        DateTime constantDate)
    {
        DataSet invoiceDataSet = new DataSet("Invoice");
        DataTable invoiceTable =
            invoiceDataSet.Tables.Add("Invoices");
        DataColumn column = new DataColumn("Id",
            typeof(Guid));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Date", typeof(DateTime));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Title", typeof(String));
```



```
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Status", typeof(int));
        invoiceTable.Columns.Add(column);
        DataRow invoiceRow =
            invoiceTable.NewRow();
        invoiceRow["Id"] = invoiceId;
        invoiceRow["Date"] = constantDate;
        invoiceRow["Status"] = InvoiceStatus.Posted;
        invoiceRow["Title"] = "Test";

        invoiceTable.Rows.Add(invoiceRow);
        return invoiceDataSet;
    }

    /// <summary>
    /// Creates canned line item data
    /// </summary>
    /// <param name="invoiceId"></param>
    /// <returns>DataSet of line item data</returns>
    private DataSet CreateLineItemDataSet(Guid invoiceId)
    {
        DataSet lineItemDataSet = new DataSet("LineItem");
        DataTable lineItemTable =
            lineItemDataSet.Tables.Add("LineItems");
        DataColumn column =
            new DataColumn("InvoiceId", typeof(Guid));
        lineItemTable.Columns.Add(column);
        column = new DataColumn("Price", typeof(Decimal));
        lineItemTable.Columns.Add(column);
        column = new DataColumn("Quantity", typeof(int));
        lineItemTable.Columns.Add(column);
        column = new DataColumn("Discount", typeof(double));
        lineItemTable.Columns.Add(column);
        column = new DataColumn("Description", typeof(String));
        lineItemTable.Columns.Add(column);
        column = new DataColumn("TaxRate1", typeof(String));
        lineItemTable.Columns.Add(column);
        column = new DataColumn("TaxRate2", typeof(String));
        lineItemTable.Columns.Add(column);

        DataRow lineItemRow =
            lineItemDataSet.Tables["LineItems"].NewRow();

        lineItemRow["InvoiceId"] = invoiceId;
        lineItemRow["Discount"] = 1F;
        lineItemRow["Price"] = 2F;
        lineItemRow["Quantity"] = 3;
    }
}
```

```
        lineItemRow["Description"] = "Description";
        lineItemTable.Rows.Add(lineItemRow);
        return lineItemDataSet;
    }
    /// <summary>
    /// Test Load Method
    /// </summary>
    [TestMethod]
    public void LoadTest()
    {
        DateTime expectedDate = DateTime.Now;
        Guid invoiceId = Guid.NewGuid();
        var mocker = new Rhino.Mocks.MockRepository();
        IDataAccess dataAccess = mocker.Stub<IDataAccess>();
        using (mocker.Record())
        {
            Rhino.Mocks.SetupResult.For(
                dataAccess.LoadInvoice(invoiceId))
                .Return(CreateInvoiceDataSet(invoiceId,
                    expectedDate));
            Rhino.Mocks.SetupResult.For(
                dataAccess.LoadInvoiceLineItems(invoiceId))
                .Return(CreateLineItemDataSet(invoiceId));
        }
        InvoiceRepository target =
            new InvoiceRepository(dataAccess);
        Invoice actualInvoice = target.Load(invoiceId);
        Assert.AreEqual(expectedDate, actualInvoice.Date);
        Assert.AreEqual(invoiceId, actualInvoice.Id);
        Assert.AreEqual("Test", actualInvoice.Title);
        Assert.AreEqual(InvoiceStatus.Posted,
            actualInvoice.Status);
        Assert.AreEqual(1, actualInvoice.LineItems.Count());
        InvoiceLineItem actualLineItem =
            actualInvoice.LineItems.First();
        Assert.AreEqual("Description",
            actualLineItem.Description);
        Assert.AreEqual(1F, actualLineItem.Discount);
        Assert.AreEqual(2F, actualLineItem.Price);
        Assert.AreEqual(3F, actualLineItem.Quantity);
    }
}
```

Rhino Mocks works with the record/playback paradigm. In our case, we first create a `MockRepository` that will be used to "load" mocks (and stubs, in our case). We then ask Rhino Mocks to create a stub of our `IDataAccess`. We then ask Rhino Mocks that we want the result of `CreateInvoiceDataSet` to be returned to `IDataAccess.LoadInvoice` when given the value stored in `invoiceId`. The same for `LoadInvoiceLineItems` and `CreateLineItemDataSet`. Rhino Mocks "records" these actions. If we want to also validate behavior, (that is, that certain methods are called and they're called in a certain order) we can use the "playback" feature. We're concentrating on state-based tests in our examples, so we don't have a corresponding playback call. We then proceed with the test using our stubbed `IDataAccess` and verifying that `Load` returns the data that we expect it to for the given inputs.

Moq

Moq (pronounced "Mock") is an open source mocking framework. To use Moq with your test project, simply add a reference to `Moq.dll` after installation.

Moq uses lambda expressions to configure each method in the test double. Our test re-written to support Moq would look like the following:

```
/// <summary>
/// Test Load Method
/// </summary>
[TestMethod]
public void LoadTest()
{
    DateTime expectedDate = DateTime.Now;
    Guid invoiceId = Guid.NewGuid();
    var mock = new Moq.Mock<IDataAccess>();
    mock.Setup(da => da.LoadInvoice(invoiceId))
        .Returns(CreateInvoiceDataSet(invoiceId, expectedDate));
    mock.Setup(da => da.LoadInvoiceLineItems(invoiceId))
        .Returns(CreateLineItemDataSet(invoiceId));
    IDataAccess dataAccess = mock.Object;

    InvoiceRepository target = new
        InvoiceRepository(dataAccess);

    Invoice actualInvoice = target.Load(invoiceId);
    Assert.AreEqual(expectedDate, actualInvoice.Date);
    Assert.AreEqual(invoiceId, actualInvoice.Id);
    Assert.AreEqual("Test", actualInvoice.Title);
    Assert.AreEqual(InvoiceStatus.Posted,
        actualInvoice.Status);
    Assert.AreEqual(1, actualInvoice.LineItems.Count());
}
```

```
InvoiceLineItem actualLineItem =
    actualInvoice.LineItems.First();
Assert.AreEqual("Description", actualLineItem.Description);
Assert.AreEqual(1F, actualLineItem.Discount);
Assert.AreEqual(2F, actualLineItem.Price);
Assert.AreEqual(3F, actualLineItem.Quantity);
}
```

With this method, we start by creating a mock of our interface. Once we have a Mock object we then configure what return values we would like our methods to have. With Moq, the lambda expression tells Moq what method we want to configure. `Mock<T>.Setup` accepts the lambda statement then parses that statement to find what method we are referring to "`da => da.LoadInvoice(invoiceId)`" is code to execute the `IDataAccess.LoadInvoice` method passing in a specific Guid value. Moq parses that and allows you to configure what return value will be returned for that specific Guid value. In our case, when `IData.LoadInvoice` is invoked with the value in `invoiceId`, we return the results of the `CreateInvoiceDataSet()` method. We configure `IDataAccess.LoadInvoiceLineItems` similarly. We then access the `Mock<T>.Object` property to get the actual test double object that is used in the rest of the test.

Unit testing existing legacy code

If we already have unit tests for a particular portion of code, then we can reasonably be sure that we can detect whether changes to this code has caused any changes in external behavior.

As we detailed in a previous chapter, legacy code is code that has no automated tests associated with it. There's two realistic ways to deal with legacy code that we detailed in the previous chapter: to go ahead and evolve it, or to decouple it from its dependants and treat it like a third party library that we don't have control over.

When you're refactoring code, that refactoring is often to redesign the code to follow principles like loosely-coupled. True unit-testing is possible only if the code is loosely-coupled. This is a Catch-22. Clearly, we can't unit-test until we have a loosely-coupled design where we want to unit test, but we can't verify refactoring to loosely-coupled without unit-testing.

So, how do we deal with this paradox? Clearly, we can write succinct and efficient unit tests to validate our existing legacy code before refactoring. But, we can begin refactoring our legacy code and the code that uses it. Some fairly benign refactorings can be applied to decouple the legacy code from the code that uses it. The Extract Interface refactoring is excellent for this. A particular legacy class can have the Extract Interface refactoring performed on it to create a new interface that matches the public interface of the class that it is performed upon. This refactoring doesn't replace use of variables that use the type with the new interface so you'll have to manually do that for each case where you need it. This works out well because you're going to focus on individual uses of the legacy code because you're testing the code that uses the legacy code, not the legacy code.

If we look at a previous implementation of `DataAccess`, it originally did not implement an `IDataAccess` interface. This class – which we'll use as an example of a legacy component – may have been used like the following:

```
    DataAccess dataAccess = new DataAccess(connectionString);
    InvoiceRepository repository =
        new InvoiceRepository(dataAccess);
```

Where `InvoiceRepository` is implemented, partially, as follows:

```
public class InvoiceRepository : IInvoiceRepository
{
    DataAccess dataAccess;
    public InvoiceRepository(DataAccess dataAccess)
    {
        this.dataAccess = dataAccess;
    }
    //...
}
```

If we performed the Extract Interface on `DataAccess` to generate an `IDataAccess` interface, we'd refactor our `InvoiceRepository` class to the following:

```
public class InvoiceRepository : IInvoiceRepository
{
    IDataAccess dataAccess;
    public InvoiceRepository(IDataAccess dataAccess)
    {
        this.dataAccess = dataAccess;
    }
    //...
}
```

Both the Extract Interface refactoring and the refactoring of `InvoiceRepository` are fairly benign. We've now decoupled `InvoiceRepository` from directly using `DataAccess`, but haven't changed any logic (just variable types). With little change to `DataAccess` (what we've deemed as the "legacy" component), the `InvoiceRepository` (the system under test) is now free to use a Test Double as a substitute for `DataAccess` in order to isolate the code in `InvoiceRepository` to fully validate. This affords us the ability to test our repository without having to deploy, install, populate, and configure a database.

TypeMock isolator

We've seen how facilitating unit tests sometimes requires changing existing code in order to realistically test certain code. Mocking usually requires a certain level of decoupled design in order to substitute product components with test doubles. In addition to being a mocking framework, TypeMock isolator offers the ability to stub out instance methods of class instances to replace their return value with canned values.

For example, if we return to our `DataAccess` class example before it was refactored for:

```
[TestClass]
public class InvoiceRepositoryTest
{
    private DataSet CreateInvoiceDataSet(Guid invoiceId,
        DateTime constantDate)
    {
        DataSet invoiceDataSet = new DataSet("Invoice");
        DataTable invoiceTable =
            invoiceDataSet.Tables.Add("Invoices");
        DataColumn column = new DataColumn("Id",
            typeof(Guid));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Date", typeof(DateTime));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Title", typeof(String));
        invoiceTable.Columns.Add(column);
        column = new DataColumn("Status", typeof(int));
        invoiceTable.Columns.Add(column);
        DataRow invoiceRow =
            invoiceTable.NewRow();
        invoiceRow["Id"] = invoiceId;
        invoiceRow["Date"] = constantDate;
        invoiceRow["Status"] = InvoiceStatus.Posted;
    }
}
```

```
        invoiceRow["Title"] = "Test";
        invoiceTable.Rows.Add(invoiceRow);
        return invoiceDataSet;
    }

private DataSet CreateLineItemDataSet(Guid invoiceId)
{
    DataSet lineItemDataSet = new DataSet("LineItem");
    DataTable lineItemTable =
        lineItemDataSet.Tables.Add("LineItems");
    DataColumn column =
        new DataColumn("InvoiceId", typeof(Guid));
    lineItemTable.Columns.Add(column);
    column = new DataColumn("Price", typeof(Decimal));
    lineItemTable.Columns.Add(column);
    column = new DataColumn("Quantity", typeof(int));
    lineItemTable.Columns.Add(column);
    column = new DataColumn("Discount", typeof(double));
    lineItemTable.Columns.Add(column);
    column = new DataColumn("Description", typeof(String));
    lineItemTable.Columns.Add(column);
    column = new DataColumn("TaxRate1", typeof(String));
    lineItemTable.Columns.Add(column);
    column = new DataColumn("TaxRate2", typeof(String));
    lineItemTable.Columns.Add(column);

    DataRow lineItemRow =
        lineItemDataSet.Tables["LineItems"].NewRow();

    lineItemRow["InvoiceId"] = invoiceId;
    lineItemRow["Discount"] = 1F;
    lineItemRow["Price"] = 2F;
    lineItemRow["Quantity"] = 3;
    lineItemRow["Description"] = "Description";

    lineItemTable.Rows.Add(lineItemRow);

    return lineItemDataSet;
}

[TestMethod]
public void LoadTest()
{
    DateTime expectedDate = DateTime.Now;
    Guid invoiceId = Guid.NewGuid();
    String connectionString = "";
    DataAccess dataAccess =
        new DataAccess(connectionString);
```

```

Isolate.NonPublic.WhenCalled(dataAccess,
    "LoadInvoice").
    WillReturn(CreateInvoiceDataSet (invoiceId,
        expectedDate));
Isolate.NonPublic.WhenCalled(dataAccess,
    "LoadInvoiceLineItems").
    WillReturn(CreateLineItemDataSet (invoiceId));
InvoiceRepository target =
    new InvoiceRepository(dataAccess);
Invoice actualInvoice = target.Load(invoiceId);
Assert.AreEqual(expectedDate, actualInvoice.Date);
Assert.AreEqual(invoiceId, actualInvoice.Id);
Assert.AreEqual("Test", actualInvoice.Title);
Assert.AreEqual(InvoiceStatus.Posted,
    actualInvoice.Status);
Assert.AreEqual(1, actualInvoice.LineItems.Count());
InvoiceLineItem actualLineItem =
    actualInvoice.LineItems.First();
Assert.AreEqual("Description",
    actualLineItem.Description);
Assert.AreEqual(1F, actualLineItem.Discount);
Assert.AreEqual(2F, actualLineItem.Price);
Assert.AreEqual(3F, actualLineItem.Quantity);
    }
}

```

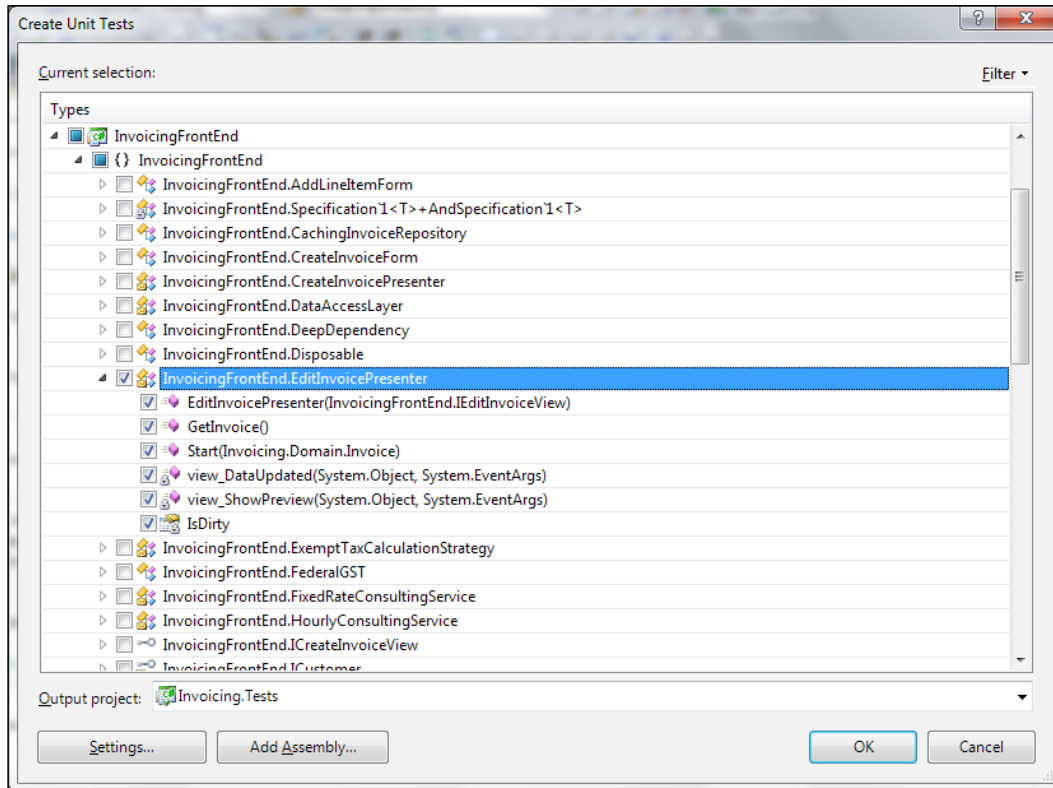
In the `LoadTest` method, we create a `DataAccess` object passing the constructor a dummy connection string. We then ask `TypeMock` isolator to return the result of `CreateInvoiceDataSet()` when `DataAccess.LoadInvoice()` is called, and to return the result of `CreateLineItemDataSet()` when `DataAccess.LoadInvoiceLineItem()` is called. `CreateInvoiceDataSet()` mimics the result of accessing the database by returning a canned `DataSet` containing invoice data.

`CreateInvoiceLineItemDataSet()` does something similar by returning a canned `DataSet` containing related invoice line item data. We complete the unit test by performing the same asserts we did in a previous example.

`TypeMock` isolator is useful for situations where you don't have control of the source code for the components you want to mock.

Unit testing in Visual Studio®

Unit testing is supported directly in Visual Studio® 2010 editions Professional and above. Visual Studio® supports automatically creating unit tests based on your source code. You can right click a method or a class and selecting **Create Unit Tests...** For example, if we right clicked on our `EditInvoicePresenter` class and chose **Create Unit Tests...**, we'd be presented with the following form:



The default action for a class is to select all methods within the class to be tested. If we accept the defaults for this class and press OK, code similar to the following will be generated:

```
/// <summary>
///This is a test class for CreateInvoicePresenterTest and is
/// intended to contain all CreateInvoicePresenterTest Unit
///Tests
///</summary>
```

```
/// <summary>
///A test for CreateInvoicePresenter Constructor
///</summary>
[TestMethod()]
public void CreateInvoicePresenterConstructorTest()
{
    ICreateInvoiceView view = null; // TODO: Initialize to
                                   // an appropriate value
    CreateInvoicePresenter target =
        new CreateInvoicePresenter(view);
    Assert.Inconclusive(
        "TODO: Implement code to verify target");
}

/// <summary>
///A test for GetInvoice
///</summary>
[TestMethod()]
public void GetInvoiceTest()
{
    ICreateInvoiceView view = null; // TODO: Initialize to
                                   // an appropriate value
    CreateInvoicePresenter target =
        new CreateInvoicePresenter(view); // TODO: Initialize
                                   // to an
                                   // appropriate
                                   // value
    Invoice expected = null; // TODO: Initialize to an
                            // appropriate value

    Invoice actual;
    actual = target.GetInvoice();
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive(
        "Verify the correctness of this test method.");
}

/// <summary>
///A test for Start
///</summary>
[TestMethod()]
public void StartTest()
{
    ICreateInvoiceView view = null; // TODO: Initialize to
                                   // an appropriate value
    CreateInvoicePresenter target =
```

```
        new CreateInvoicePresenter(view); // TODO: Initialize
                                           // to an
                                           // appropriate
                                           // value
    target.Start();
    Assert.Inconclusive("A method that does not return a value cannot
        be verified.");
}
}
```

The unit test creator in Visual Studio® analyzes the signature of each method and automatically generates unit test code that performs a test with dummy data. You simply have to fill in the blanks with the data you actually want to test with and remove the `Assert.Inconclusive` call. Our first unit test is a modified version of the above generated code.

Test driven development

As you start writing new code, you don't want to fall into the same trap as the legacy code you have to deal with. You want this new code to have tests in order to verify changes to it—either as the next step in your refactoring or in some future refactoring. With **Test-Driven Development (TDD)** you actually make sure you write the tests before you write the code. Let's have another look at our Model View Presenter (MVP) refactoring.

When separating out business logic code from the GUI (Graphical User Interface) we're now able to write tests to verify that business logic independent of the user interfaces—tests that can be automated much easier. Before we refactored `CreateInvoiceForm`, we knew we needed the presenter to create an invoice based on information in the view as well as construct a presenter based on a specific view. In this case, we want two tests: one to test that the presenter is created correctly with the correct view reference, and one to verify that the presenter uses the correct data from the view to create an `Invoice` object. In TDD we'd create these tests before we'd create the classes to support them. For example:

```
/// <summary>
///A test for CreateInvoicePresenter Constructor
///</summary>
[TestMethod()]
public void CreateInvoicePresenterConstructorTest()
{
    ICreateInvoiceView view = new DummyCreateInvoiceView();
    CreateInvoicePresenter target =
        new CreateInvoicePresenter(view);
}
```

```
Assert.AreSame(view, target.View, "CreateInvoicePresenter.View had
unexpected value after construction.");
}
/// <summary>
/// A test for GetInvoice
/// </summary>
[TestMethod()]
public void GetInvoiceTest()
{
    string expectedTitle = "Test";
    DateTime expectedDate = DateTime.Now;
    List<InvoiceLineItemDTO> invoiceLineItemDTOs =
        new List<InvoiceLineItemDTO>() {
            new InvoiceLineItemDTO()
            {
                Quantity = 1,
                Description = "description",
                Discount = 0F,
                Price = 42F
            }
        };
    ICreateInvoiceView view =
        new MockCreateInvoiceView()
        {
            Title = expectedTitle,
            Date = expectedDate,
            InvoiceLineItemDTOs = invoiceLineItemDTOs
        };
    CreateInvoicePresenter target =
        new CreateInvoicePresenter(view);
    Invoice actual = target.GetInvoice();
    Assert.AreEqual(expectedTitle, actual.Title);
    Assert.AreEqual(expectedDate, actual.Date);
    InvoiceLineItem actualInvoiceLineItem =
        actual.LineItems.First();
    Assert.AreEqual(invoiceLineItemDTOs[0].Description,
        actualInvoiceLineItem.Description);
    Assert.AreEqual(invoiceLineItemDTOs[0].Discount,
        actualInvoiceLineItem.Discount);
    Assert.AreEqual(invoiceLineItemDTOs[0].Price,
        actualInvoiceLineItem.Price);
    Assert.AreEqual(invoiceLineItemDTOs[0].Quantity,
        actualInvoiceLineItem.Quantity);
}
```

In `CreateInvoicePresenterConstructorTest` we create **Dummy Test Double** to pass to the `CreateInvoicePresenter` constructor that will only be used to verify, is used for the `CreateInvoicePresenter.View` property. The test verifies that the constructor is working correctly by verifying that `View` property contains the same reference that was passed into `CreateInvoicePresenter` constructor through the use of `Assert.AreSame()`.

In the `GetInvoiceTest` method, we create all of the inputs to the test (`expectedTitle`, `expectedDate`, and `invoiceLineItemDTOs`). We then use those inputs to create **Mock Test Double** of the view when creating the presenter object. The call to `CreateInvoicePresenter.GetInvoice()` will then use the attributes of our **Mock** object to create an `Invoice` object. We then verify that what is returned from `GetInvoice` is correct (and thus verify that `GetInvoice` is functioning properly) through multiple calls to `Assert.AreEqual` on what we expect compared to attributes of the `Invoice` object.

Had our `Invoice` object overridden the `Equals` method, we could have possibly improved this test as follows:

```
public void GetInvoiceTest()
{
    List<InvoiceLineItemDTO> invoiceLineItemDTOs =
        new List<InvoiceLineItemDTO>() {
            new InvoiceLineItemDTO()
            {
                Quantity = 1,
                Description = "description",
                Discount = 0F,
                Price = 42F
            }
        };

    Invoice expectedInvoice = new Invoice("Title",
        new List<InvoiceLineItem>() {
            invoiceLineItemDTOs.First().ToInvoiceLineItem()
        },
        DateTime.Now);

    ICreateInvoiceView view =
        new MockCreateInvoiceView()
        {
            Title = expectedInvoice.Title,
            Date = expectedInvoice.Date,
            InvoiceLineItemDTOs = invoiceLineItemDTOs
        };

    CreateInvoicePresenter target =
```

```
        new CreateInvoicePresenter(view);
        Invoice actual = target.GetInvoice();
        Assert.AreEqual(expectedInvoice, actual, "Actual invoice returned
from GetInvoice was not as expected.");
    }
```

This simplifies our verification by process to a single call to `Assert.AreEqual`; but we lose granularity in the detail of what part of the Invoice doesn't match. We also employ `InvoiceLineItemDTO.ToInvoiceLineItem`, making the test dependant on that working and effectively dependant on tests to verify `ToInvoiceLineItem` be run and pass before this test. I'll leave it as an exercise of the reader to decide which benefits and drawbacks are important.

This of course, results in compile errors if we try to build our test project. We then create enough of the classes and code to get a clean compile. But, when we run the two tests they will both fail because the code hasn't been written yet. The following is just enough code to compile and fail the tests:

```
class DummyCreateInvoiceView : ICreateInvoiceView
{
    public ICollection<InvoiceLineItemDTO> InvoiceLineItemDTOS
    {
        get { throw new NotImplementedException(); }
    }
    public DateTime Date
    {
        get
        {
            throw new NotImplementedException();
        }
        set
        {
            throw new NotImplementedException();
        }
    }
    public string Title
    {
        get
        {
            throw new NotImplementedException();
        }
        set
        {
            throw new NotImplementedException();
        }
    }
}
```

```
    }
  }
}

class CreateInvoicePresenter
{
  public CreateInvoicePresenter(ICreateInvoiceView view)
  {
  }

  public ICreateInvoiceView View { get; private set; }

  public void Start()
  {
    // TODO: wire-up other data, subscribe to events, etc.
  }

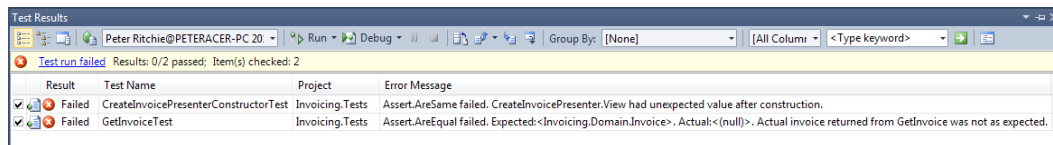
  public Invoice GetInvoice()
  {
    return null;
  }
}

class MockCreateInvoiceView : ICreateInvoiceView
{
  public ICollection<InvoiceLineItemDTO> InvoiceLineItemDTOs {
    get;
    set;
  }

  public DateTime Date { get; set; }


  public string Title { get; set; }
}
```

Running the tests in the Visual Studio® 2010 test runner show that the tests fail:



The screenshot shows the Test Results window in Visual Studio 2010. The status bar indicates 'Test run failed' with 'Results: 0/2 passed; Item(s) checked: 2'. The table below lists the failed tests:

Result	Test Name	Project	Error Message
Failed	CreateInvoicePresenterConstructorTest	Invoicing.Tests	Assert.AreSame failed. CreateInvoicePresenter.View had unexpected value after construction.
Failed	GetInvoiceTest	Invoicing.Tests	Assert.AreEqual failed. Expected:<Invoicing.Domain.Invoice>. Actual:<(null)>. Actual invoice returned from GetInvoice was not as expected.

 Failing tests first? This may seem counterproductive, but doing this has one striking benefit: it tests that unexpected things don't pass the test, that is, the defaults don't cause the test to pass and that actual written code is required to pass the test.

Unit testing frameworks

Visual Studio® 2010 Professional Edition and better includes unit-testing. Visual Studio® 2010 Premium includes additional testing tools like Code Coverage. Visual Studio® 2010 Ultimate Edition includes extra features related to managing the testing effort like Test Case Management. There are third party unit testing frameworks that are available if you're using Visual Studio® 2010 Express or Standard. We'll look at a common subset of these unit testing frameworks now.

NUnit

NUnit was one of the first unit testing frameworks available for .NET. It's the defacto standard for automated testing on .NET. The original author of NUnit is actually working for Microsoft now. Even if you find you're going to use the built-in Visual Studio® unit testing, you may encounter unit tests written for NUnit so it may be worthwhile understanding how NUnit works.

To write unit tests for NUnit, simply add a reference to the `nunit.framework.dll` after installation.

NUnit differs from VS unit testing firstly in the names of the attributes that declare a test class and a test method. NUnit uses `TestFixtureAttribute` instead of `TestClassAttribute` and `TestAttribute` instead of `TestMethod` attribute. Many of the Assert methods are the same. The following is an example of our test modified to work with NUnit:

```
[TestFixture]
public class InvoiceRepositoryTest
{
    [Test]
    public void LoadTest()
    {
        DateTime expectedDate = DateTime.Now;
        IDataAccess dataAccess =
            new InvoiceRepositoryDataAccessStub(expectedDate);
        InvoiceRepository target = new
            InvoiceRepository(dataAccess);
        Guid invoiceId = Guid.NewGuid();

        Invoice actualInvoice = target.Load(invoiceId);
        Assert.AreEqual(expectedDate, actualInvoice.Date);
        Assert.AreEqual(invoiceId, actualInvoice.Id);
        Assert.AreEqual("Test", actualInvoice.Title);
        Assert.AreEqual(InvoiceStatus.New,
            actualInvoice.Status);
    }
}
```



```
Assert.AreEqual(1, actualInvoice.LineItems.Count());
InvoiceLineItem actualLineItem =
    actualInvoice.LineItems.First();
Assert.AreEqual("Description",
    actualLineItem.Description);
Assert.AreEqual(1F, actualLineItem.Discount);
Assert.AreEqual(2F, actualLineItem.Price);
Assert.AreEqual(3F, actualLineItem.Quantity);
    }
}
```

XUnit

XUnit is an open source unit-testing framework originating out of Microsoft. Unfortunately, before going to press, XUnit didn't support .NET 4.0. So, to use XUnit you have to change the target framework to something other than .NET 4.0 for your test project and any project that your test project references (usually every other project).

To write tests for XUnit, simply add a reference to the `XUnit.dll` after installation.

XUnit's semantics is a little different than other testing frameworks. The first thing to notice is you don't have to attribute the test class at all. XUnit finds and executes tests simply by the method attribute. In XUnit the method attribute is `FactAttribute`. XUnit also doesn't use the same method names in the `Assert` class. Our test modified to work with XUnit would look like the following:

```
public class InvoiceRepositoryTest
{
    [Fact]
    public void LoadTest()
    {
        DateTime expectedDate = DateTime.Now;
        IDataAccess dataAccess =
            new InvoiceRepositoryDataAccessStub(expectedDate);
        InvoiceRepository target = new
            InvoiceRepository(dataAccess);
        Guid invoiceId = Guid.NewGuid();

        Invoice actualInvoice = target.Load(invoiceId);

        Assert.Equal(expectedDate, actualInvoice.Date);
        Assert.Equal(invoiceId, actualInvoice.Id);
        Assert.Equal("Test", actualInvoice.Title);
        Assert.Equal(InvoiceStatus.Posted,
            actualInvoice.Status);
    }
}
```

```
Assert.Equal(1,
    actualInvoice.LineItems.Count());
InvoiceLineItem actualLineItem =
    actualInvoice.LineItems.First();
Assert.Equal("Description",
    actualLineItem.Description);
Assert.Equal(1F, actualLineItem.Discount);
Assert.Equal(2F, actualLineItem.Price);
Assert.Equal(3F, actualLineItem.Quantity);
}
}
```

Automating unit-testing

So far we've detailed coding unit test and the automatic executing through some sort of unit test framework. But, the executing of the tests has been a manual process – pressing "Run" in order to execute the tests. This assumes a developer is modifying code and running the tests to verify their changes. In an ideal world, this would be enough to verify our software system. But, various aspects of software development mean this isn't enough. Team software development, for example, means that someone else on the team could have made a modification that causes a test to fail. This failure may not be visible to them while they are both developing. While both team members may execute all the tests and they all pass, the act of both of them committing their changes to source control could introduce a failure that they wouldn't notice. This failure could impact other team members causing them decreased productivity or reduced quality.

Automated unit-testing is when the unit tests are run atomically. They may simply be periodically run, or they may be run in response to a particular action (like committing changes), or it may be a combination of both.

Some source control systems recognize that the code that is being committed may pass unit tests locally but may not pass unit test once committed. These types of source control systems have check-in policies that perform certain checks before fully committing the revision. In terms of unit-testing, there is a policy that can be enabled and/or configured that automatically executes the unit tests (or a subset thereof) and will not allow the change set to be committed unless all the unit tests pass. This is the most proactive way of ensuring unit tests are periodically executed so that changes are least likely to affect other members of the team.

An alternative to using the revision control system to proactively deny commits that don't pass all unit tests, unit tests can be periodically automatically executed and can inform team members of the status. This can simply be scheduled as part of the build environment, like unit tests automatically being executed every day at 2 a.m. This could also be in response to check-ins: where unit tests are executed on a specific server whenever a check-in is performed, and any errors are sent to team members so they know as soon as possible that there's a problem.

Continuous Integration

There's a category of tools and features that implement a feature called Continuous Integration. Continuous Integration promotes integration of the software system under development and the work of development team members continuously. Continuous Integration tools essentially facilitate the automation of the build process. This sounds very simple, but there are lots of details to a full build process. There's the integration with a source code control system, the extraction of the latest source code from the source code control system, the compilation of the source code, the execution of unit tests, compilation of installs, and communication of build status with the team. This process can also include many other details like automatic deployment to a test environment. Continuous Integration tools facilitate these tasks. There are several tools available for Continuous Integration from open source to commercial. Visual Studio® Team System (VSTS) includes **Team Foundation Server (TFS)** that facilitates Continuous Integration. CruiseControl.NET is an open source Continuous Integration tool. There are a few commercial Continuous Integration tools; one of the more popular ones (other than TFS) is TeamCity.

Continuous Integration frees the development team from a manual build process that someone has to be trained on and perform. Continuous Integration usually includes the process of running the unit tests, so there's an up-to-date status on the quality of the source code. The Continuous Integration tool can be configured to periodically start the build process, and can often be configured to automatically detect change set commits to the source code control system and automatically (and thus continuously) execute the build process.

Continuous Integration means that the work of development team members is integrated as quickly as possible. It's important that problems with the work product of people are found as close to when they perform that work as possible. The longer it takes to detect problems, the greater the amount of work involved is required to fix the problem. The more work involved to fix a problem, the more likely that quality will be affected. These tools mean problems are detected as closely as possible when they're created.

Continuous Integration leads to more rapid development of software, finding failures and integration problems as quickly as possible, and reducing the work involved in resolving integration and quality issues.


Continuous Integration relies heavily on a source code control system. If you're unfamiliar with source code control systems, please refer to Chapter 2 for more detail on source code control systems and the principle of check-in-often.

Third party refactoring tools

There's third party refactoring add-ons for Visual Studio® that increases the six built-in refactorings into dozens of refactorings. A couple of those add-ins are detailed below.

Resharper


JetBrains has a product called Resharper. Resharper (version 5) includes 40 refactorings. Resharper includes many productivity utilities to aid in the development of software, especially with regard to Test Driven Development. Resharper also includes a unit test runner that works with Visual Studio® 2010 Standard – that doesn't include a built-in unit test runner.

[ More information about Resharper can be found at <http://www.jetbrains.com/resharper/index.html>]

Refactor! Pro

DevExpress has a product called Refactor! Pro. Refactor! Pro has over 150 refactorings. As with Resharper, Refactor! Pro includes many productivity features that when coupled with DevExpress' CodeRush, productivity gains are increased even more.

I highly recommend using one of these two productivity tools. These tools offer features and extensions to Visual Studio® that better match the way people code and make coding that much more productive. After getting to know how to use either one of these tools, you'll find developing software in just Visual Studio® much less productive.

[ More information about Refactor Pro! can be found at http://www.devexpress.com/Products/Visual_Studio_Add-in/Refactoring/]

Summary

In this chapter, we discussed that to support the refactoring effort some level of automated unit testing is required to maintain the quality of the software. We detailed some testing frameworks to facilitate writing and executing those tests. We detailed other frameworks that aid in writing unit test faster, like mocking frameworks. We also detailed how we might approach promoting unit testing for new code through practices like test-driven development.

This ends our refactoring journey, so far. I hope this book shows you what you gain from approaching refactoring as a succinct and independent task. It allows you to evolve source code systematically while minimizing quality issues. Refactoring is a recognition that software needs to change to suit the needs of its users over time and the software projects need to be agile to support their users' changing needs.

Index

A

- abstract factory**
 - about 201
 - refactoring 202
- abstractions 191**
- accumulative construction 53**
- acyclic dependency principle 186**
- adapter pattern**
 - about 283
 - Data Transfer Object (DTO) class 285
 - DataUpdated event 287, 290
 - Form object 287
 - GetInvoice method 290
 - IEditInvoiceView 287
 - IInvoiceRespository object 285
 - invoice field 287
 - InvoiceLineItem class 286
 - InvoiceLineItemDTO class 286
 - InvoiceLineItemDTOs 287
 - InvoicePreviewForm object 290
 - LineItems property 287
 - Model View Presenter (MVP) pattern, implementing in WinForms 284
 - need, detecting 283, 284
 - previewButton_Click method 287
 - refactoring to 284-291
 - ShowPreview handler 290
 - Test Double 285
 - ViewInvoiceForm 285
 - ViewInvoiceForm class 287, 290
- AddContact 131**
- afferent coupling 207**
- aggregate types 148**
- And method 268, 271**

- AndSpecification<T> class 268**
- application layer 248**
- assembly cohesion**
 - about 173
 - refactoring 174
- Assert.AreEqual 335**
- atomic operation 149**
- automated testing 314**
- automated unit testing 69, 71, 72**

B

- Barbara Liskov 122**
- behavioral patterns**
 - about 252
 - class patterns 252
 - Don't Repeat Yourself (DRY) 253
 - object patterns 252
 - observer, need detecting for 275
 - observer pattern 274, 275
 - observer pattern, refactoring to 275-278
 - publish/subscribe paradigm 273, 274
 - refactoring to, need for 252
 - specification pattern 259, 260
 - specification pattern, need detecting 261
 - specification pattern, refactoring to 261-273
 - strategy pattern 253, 254
 - strategy pattern, need detecting for 254, 255
 - strategy pattern, refactoring to 255-259
 - types 252
- Big Design Up Front (BDUF) 11**
- business logic layer 218, 219, 238**
- business rules**
 - modeling 152, 153
- Button.Click event 275**

C

CA1051

URL 41

CA1502 Avoid Excessive Complexity 27

CalculateArea() attribute 127

CalculateInvoiceGrandTotal 78

CalculateInvoiceGrandTotal method 75

CalculateInvoiceTotalTax method 73

CalculateLineItemSubTotal, CalculateInvoiceTotalTax 78

CalculateTaxReceivable method 257

CalculateTotals 168

check-in often 65

Circular dependency. *See* dependency cycles

class cohesion

about 156-159

detecting, with low-cohesion 163, 164

refactoring, with low-cohesion 160-163

single responsibility principle (SRP) 160

Class Coupling 206

class diagram, IDE navigation

about 114

bottom class details pane 114

creating 115

rename refactor 115

top design surface pane 114

classes

Shape class 82

class patterns 252

class, rename identifier refactoring

renaming 37

class view, IDE navigation

about 108

lower Members pane 108

Move To Namespace refactoring 108

nested namespace declaration 109

upper Objects pane 108

Clone Detective

about 42

URL 42

code

changes, tracking 64

performance 24

unused references 65

unused references, removing 65

code comments smell

about 56

example 57

magic number, refactoring 59

code coverage 320

code editor, IDE navigation

about 116

Navigate Backward command button 117

Navigate Forward command button 117

text editor area 116

code maintainability

about 67-69

automated unit testing 69-72

contrived complexity 90-93

Don't Repeat Yourself (DRY) 84

feature envy code smell 81

Feature Envy code smell 81, 82

inappropriate intimacy code smell 84-86

issues, detecting 95, 96

lazy class code smell 87, 88

maintainability issues, detecting 96, 97

object-model usability 88, 89

code metrics 27

code, navigating in Visual Studio® 2010

class diagram 114, 115

class view 108, 110

code editor 116-118

search 106, 107

solution explorer 111-114

code smells

about 23, 41, 68

code method smell 56-59

duplicate code smells 41, 42

long method smell 56

cohesion

about 155, 156

assembly cohesion 173

class cohesion 156-159

method cohesion 164, 165

namespace cohesion 171

collapse hierarchy refactoring 87

Color attribute 127

Combine method 271

command 170

Comments 143

communication hosts 215

complexity

- reducing 23
- composition over inheritance**
 - about 134
 - exceptions 140
 - foreach loop 136
 - IDisposable 138
 - IEnumerable<InvoiceLineItem> 136
 - inheritance, replacing with delegation
 - refactoring 140, 142
 - Invoice class 136
 - InvoiceLineItem objects 136
 - InvoiceLineItems 134
 - List<T> 134
 - List<T> constructor 136
 - multiple-inheritance 136
 - object behavior 142, 144
 - object-orientation, refactoring to 144-148
 - object-oriented design 142, 144
 - superclass 137
 - virtual methods, refactoring to events
 - 138-140
- consistency, Convention over Configuration (CoC) pattern 101, 102**
- ContactInformation class 42**
- convention, Convention over Configuration (CoC) pattern 102**
- Convention over Configuration (CoC) pattern**
 - about 101
 - consistency 101, 102
 - conventions 102
 - scoping types, with namespaces 104, 105
- conventions, Convention over Configuration (CoC) pattern**
 - naming 102-104
- Convert to Sibling refactoring 126, 127**
- coupling**
 - about 174, 175, 190
 - content coupling 175
 - control coupling 175
 - data coupling 175
 - dependency cycles 186
 - external coupling 175
 - inward coupling 175
 - message coupling 175
 - outward coupling 175
 - proper dependency design 187
 - refactoring content coupling 176
 - refactoring external coupling 183, 185
 - refactoring subclass coupling 175
 - types 207
 - subclass coupling 175
- coupling, types**
 - afferent coupling 207
 - efferent coupling 208
- CreateCustomer method 89**
- CreateInvoiceDataSet 324**
- CreateInvoiceDataSet() method 325, 329**
- CreateInvoiceForm class 199, 240, 242**
- CreateInvoiceForm.okButton_Click method 201**
- CreateInvoiceForm refactoring 200**
- CreateInvoiceLineItemDataSet() 329**
- CreateInvoice method 205**
- CreateInvoicePresenter class 245**
- CreateInvoicePresenter constructor 334**
- CreateInvoicePresenterConstructorTest 334**
- CreateInvoicePresenter.View property 334**
- CreateLineItemDataSet() 329**
- CRUD Interface 305**
- Customer class 85, 132**
- Customer object 131**
- Customer parameter 165**
- Cyclomatic Complexity (CC) 96, 207**
- Cyclomatic Complexity software metric 27**

D

- data access layer**
 - about 218, 221
 - domain data access, refactoring 233-236
 - UI data access, refactoring 221-233
- Data Access Layer (DAL) 317**
- DataAccess.LoadInvoice() 329**
- DataAccess.LoadInvoiceLineItem() 329**
- DataAccess methods 296**
- DataAccess object 329**
- data-driven classes**
 - designing 144
- data-driven design 142-144**
- DataSet object 319**
- DataSet objects 304**
- Data Transfer Object (DTO) 144 242**
- DataUpdated event 277, 287, 290**

- dead code** 60
- decorator pattern** 203, 206
- dependency** 190
- dependency cycles** 186
- dependency injection**
 - about 193-195
 - InvoiceLineItem objects 195
 - InvoiceRenderingService class 195
 - InvoiceRenderingService class
 - constructor 195
 - InvoiceRenderingService constructor 195
 - refactoring, performing 195
- dependency inversion principle**
 - about 118, 119, 191
 - abstractions 191
 - details 191
 - high-level modules 191
 - low-level modules 191
- design patterns** 118
- details** 191
- detecting highly-coupled** 190
- DeviceInterface** 102
- Disposed event handler** 140
- domain data access**
 - refactoring, to data access layer 233-236
- domain-driven design** 219
- Domain Layer** 220, 307
- Don't Repeat Yourself (DRY)** 41, 84, 253
- Duck Typing** 274, 275
- dummy, test double** 316
- duplicate code**
 - in class 42-45
 - in construction 53
 - in multiple classes 47-53
- Duplicate Code Detector** 42
- duplicate code smells**
 - advanced duplicate code refactoring 54-56
 - duplicate code, in class 42-46
 - duplicate code, in construction 53
 - duplicate code, in multiple classes 47-51

E

- EditInvoicePresenter class** 247, 248
- efferent coupling** 208
- encapsulate field refactoring**
 - about 39

- reference changes, previewing 40
- encapsulate field, simple refactoring** 13
- events**
 - virtual methods, refactoring 138-40
- Execute method** 63
- ExemptTaxCalculationStrategy** 257
- ExemptTaxCalculationStrategy class** 257
- Expression<> object** 265
- ExpressionParameterExchangerVisitor class** 269
- Expression<T> object** 264
- extract interface, simple refactoring** 13
- extract method refactoring**
 - about 37-39, 168
 - CalculateInvoiceGrandTotal 169
 - CalculateInvoiceTotalTax 169
- extract method, simple refactoring** 12
- Extreme Programming.** *See* XP

F

- façade pattern**
 - about 291
 - DataAccess methods 296
 - Invoice.Load method 296
 - Invoicing.Data 295
 - LineItems properties 294
 - LoadInvoiceLineItems method 295
 - Load method 294, 297
 - need, detecting 291
 - refactoring to 292-299
- factory pattern** 200
- fake, test double** 316
- Fallacy of Reuse** 83
- feature envy code smell** 81, 82
- field design guidelines**
 - URL 41
- field, rename identifier refactoring**
 - New name 35
 - Preview Code Changes region 34
 - Preview reference changes checkbox 35
 - Refactor\Rename.... 34
 - Rename form 34
 - renaming 33, 34
- FindBySpecification method** 265
- foreach loop** 136
- FxCop**

about 26
URL 26

G

GenerateReadableInvoice 72
GetInvoice method 290
GivenName assignment 151
Graphics class 72
Graphics method 94
Graphics object 72
GuidinvoiceId parameter 231

H

higher-level layers 218
high-level modules 191
highly-coupled
 detecting 206, 207
Hollywood Principle. *See* IoC

I

ICreateInvoiceView interface 245
IDataAccess 324
IDataAccess class 318
IDataAccess.LoadInvoice method 325
IDataAccess object 231
IData.LoadInvoice 325
IDE navigation
 class diagram 114, 115
 class view 108-110
 code editor 116, 117
 search 106, 107
 solution explorer 111-114
IEditInvoiceView 287
IInterfaceC 100
IInvoiceFactory object 205
IInvoiceGrandTotalStrategy parameter 199
Invoice interface 195
InvoiceRenderingService 302, 303
InvoiceRepository object 233
InvoiceRepository parameter 233
InvoiceRepository interface 231
InvoiceRepository object 285
Impedance Mismatch 304
InactiveCustomer 131

inappropriate intimacy code smell

Customer class 85
example 84
List<PhoneNumber> object 85
List<PhoneNumber> type 86
PhoneNumbers property 84

infrastructure layer 248

inheritance

about 100
replacing, with delegation refactoring
 140-142

initialization

moving, to declaration 148

InjectionConstructor object 198

Integrated Development Environment (IDE) 9

intention-revealing design 60

Intention Revealing Naming 79

interface-driven design 100

interface segregation principle

about 208
example 208, 210
refactoring 211, 212

InvalidOperationException 131

Inversion of Control. *See* IoC

Inversion of Control Container (IoC container) 119

Invoice attribute 225

Invoice class 78, 156, 159, 236, 310

Invoice classes 114

Invoice.Date property 310

InvoiceFactory object 204

invoice field 287

invoiceGrandTotalStrategy 92

InvoiceGrandTotalStrategy 92

invoiceGrandTotalStrategy field 92

InvoiceGrandTotalStrategy object 199

InvoiceGrandTotalStrategy parameter 199

Invoice instance 260

InvoiceLineItem class 221

InvoiceLineItemDTOs 287

InvoiceLineItemDTO.ToInvoiceLineItem 335

Invoice.Load method 237, 296

Invoice object 236

InvoiceRenderingService 301-303

InvoiceRenderingService class 255, 258
InvoiceRenderingService class constructor 195
InvoiceRenderingService constructor 195
InvoiceRenderingService object 196, 303
InvoiceRenderingServiceProxy 303
InvoiceRenderingService.Render
 ReadableInvoice method 258
InvoiceRenderingServiceVirtualProxy 302, 303
InvoiceRenderingServiceVirtualProxy class 302
InvoiceRepository class 231, 276, 311, 326, 330
InvoiceRepository classes 237
InvoiceRepositoryDataAccessStub 320
InvoiceRepositoryDataAccessStub instance 320
InvoiceRepository.FindBySpecification method 265
InvoiceRepository.Load method 317
InvoiceRepositoryTest.LoadTest method 320
Invoicing.Data 295
Invoicing.Domain directive 114
Invoicing.Domain namespace 172, 309
IoC
 about 192, 193
 example 192
IoC containers
 about 193, 196
 IInvoiceGrandTotalStrategy interface 198
 InjectionConstructor object 198
 integrating, into code 197
 MemoryStream object 197
 selecting, criteria 197
 tightly-coupled code, example 199, 200
IQueryable<Invoice> instance 265
IQueryable<T> parameter 263
ISessionFactory.OpenSession() 311
ISession object 311
ISession reference 311
IsSatisfiedBy() method 265, 271
IsSatisfiedBy overload 264, 265
IsSatisfied method 261
ITaxCalculationStrategy field 259
ITaxCalculation implementation 257
ITaxCalculation object 258
ITaxCalculationStrategy implementation 258
ITaxCalculationStrategy object 258
ITaxCalculationStrategy property 259
ITaxCalculationStrategy variable 256
ITaxCalculationStrategy implementation 256
ITaxRate objects 256
J
JetBrains
 Resharper 341
just-in-time (JIT) compiler 104
K
Keep It Simple Stupid. *See* **KISS principle**
kernel 24
KISS principle 62, 64
L
Lack of Cohesion of Methods (LCOM) 163, 164
Larry Constantine
 about 155
 URL 155
layers
 about 217
 application layer 248
 business logic layer 218, 219
 business logic layer, used by UI layer 219
 data access layer 218, 219
 examples 218
 higher-level layers 218
 infrastructure layer 248
 logical layers 218
 lower-level layers 218
 presentation layer 248
 service layer 248
 user interface layer 218
lazy class code smell 87
legacy code
 unit testing 325, 326
LineItems properties 294
LineItems property 287

Lines of Code (LOC) 97
Liskov substitution principle
 about 122
 Convert to Sibling refactoring 126, 127
 detecting 122
 example 123
 quintessential example 122
 Rectangle class 123
 Rectangle object 125
 Rectangle/Square scenario 124
 refactoring to single class 128, 130
 Square class 122
 square/rectangle scenario 126
 violating 124
List<InvoiceLineItem> 140
List<PhoneNumber> object 85
List<PhoneNumber> type 86
List<T> 134
LoadInvoiceLineItems 324
LoadInvoiceLineItems method 226, 295
LoadInvoice method 226
Load method 231, 233, 294, 295, 297
LoadTest method 329
local variable, rename identifier refactoring
 renaming 36
LoggingInvoiceFactory 204
logical layers 218
long method smell 56
LookupCustomer 150
loosely-coupled
 about 190
 communication hosts 215
 drawbacks 214
 methods 214
 web services 215
loosely-coupled, methods
 communication hosts 215
 web services 215
lower-level layers 218
low-level modules 191
LSP. *See* Liskov substitution principle

M

magic number
 about 59
 refactoring 59

maintainability, code
 about 68, 69
 issues, detecting 95
MemoryStream object 197
method cohesion
 refactoring, with low-cohesion 165-171
method, rename identifier refactoring
 renaming 36
Mock 316
mocking 316
mocking frameworks
 about 321
 Moq 324
 Rhino Mocks 321-324
MockRepository 324
Mock<T>.Setup 325
model 238
Model View Controller/Model View Presenter patterns (MVC/MVP) 118
Model View Presenter (MVP) pattern
 about 238
 CreateInvoiceForm class 240, 242
 Data Transfer Object (DTO) 242
 line item DTO 242-245
 refactoring 240, 246
 refactoring 332
modifier 170
modularity 190
ModuleX 27
ModuleY 27
Moq
 about 324
 CreateInvoiceDataSet() method 325
 IData.LoadInvoice 325
 lambda expressions 324
 Mock<T>.Object property 325
 Mock<T>.Setup 325
move method refactoring 47
Move to Another Namespace refactoring
 172
Move To Namespace refactoring 108
mutable type
 about 148
 example 148

N

namespace cohesion

- about 171
- refactoring, with low-cohesion 171-173

namespaces, Convention over Configuration (CoC) pattern

- coping types with 104, 105

naming conventions, Convention over Configuration (CoC) pattern

- naming 102-104

Navigate Backward command button 117

Navigate Forward command button 117

NDepend 207

NHibernate.Connection.Driver ConnectionHandler 309

NormalTaxCalculationStrategy class definition 256

Not method 268, 272

NotSpecification<T> class 268

NUnit 337, 338

O

ObjectContext object 312

object-orientation

- refactoring to 144-148

object-oriented code

- navigating 100

Object-oriented Myopia 83

object patterns 252

Object-Relational Impedance Mismatch 304

Object/Relational Mappers (ORMs) 144

Object/Relation Mapping (ORM)

- about 304
- code generation, issues 305
- CRUD Interface 305
- DataSet objects 304
- Domain Layer 307
- Invoice class 310
- Invoice.Date property 310
- InvoiceRepository class 311
- Invoicing.Domain namespace 309
- ISessionFactory.OpenSession() 311
- Session object 311
- ISession reference 311
- NHibernate 307

NHibernate.Connection.DriverConnection- Handler 309

ObjectContext object 312

Program.Main method 310

refactoring, need detecting 306

refactoring to 307-311

sessions 312

User Interface Layer 307

observer pattern

about 274, 275

Button.Click event 275

DataUpdated event 277

need, detecting 275

refactoring to 275-278

observer patternSaveInvoice method 277

OnTimerTick method 192

Open/Closed Principle 279

Optical Character Recognition (OCR) 71

Or method 268, 271

OrSpecification<T> class 268

OverdueInvoiceSpecification 273

overdueInvoiceSpecification object 272

P

package 186

Partner sub-namespaces 114

PastDueInvoice 88

patterns

- refactoring to 22

PersonName class 152

PersonName instance 149

personName object 149

Plain Old CLR Objects (POCO) 236, 237

PointF object arguments 78

POJO 236

Populate method 225

Prefer Composition over Inheritance

- principle.. *See composition over inheritance*

presentation layer 248

presenter 239

previewButton_Click method 225, 287

principles

- refactoring to 22, 23

procedural code 100

Program.Main method 310

Programming by Coincidence 16

proper dependency design
 about 187
 data layer 187
 layers 187

Properties.Settings.Default 301

property, rename identifier refactoring
 renaming 35

proxy pattern
 about 299, 300
 IInvoiceRenderingService 302, 303
 InvoiceRenderingService 301-303
 InvoiceRenderingService object 303
 InvoiceRenderingServiceProxy 303
 InvoiceRenderingServiceVirtualProxy 302, 303
 InvoiceRenderingServiceVirtualProxy class 302
 need, detecting 300
 Properties.Settings.Default 301
 ReaderReadableInvoice 302
 ReaderReadableInvoice method 303
 ReaderReadableInvoiceProxy 303
 realSubject 302
 realSubject object 303
 realSubject.RenderReadlbeInvoice method 303
 refactoring to 300-303
 RenderReadableInvoiceService 300

publish/subscribe paradigm
 publish/subscribe paradigm about 273, 274

Pull Up Method refactoring 47

Push Down refactoring 81
 Circle class 83
 Circle.Diameter 83
 Diameter property 82
 Shape class 82
 Shape classes 83
 Shape.Diameter 83

Q

query 170

R

ReaderReadableInvoice 302
ReaderReadableInvoice method 303

ReaderReadableInvoiceProxy 303
realSubject 302
realSubject object 303
realSubject.RenderReadlbeInvoice method 303

Rectangle class 123
Rectangle object 125, 127
Rectangle/Square scenario 124
Refactor\Extract Method 72

refactoring
 about 8, 10
 code performance 24
 code smells 23
 complexity, reducing 23, 24
 Customer 132
 design methodologies 25
 highly-used code 25
 InactiveCustomer 132
 in software development trenches 15
 in Visual Studio 2010 26
 kernel 24
 need for 10, 11
 Programming by Coincidence 16
 refactoringbehavioral pattern 252
 refactoringMove To Namespace refactoring 108
 refactoringto behavioral pattern 252
 refactoringto observer pattern 275, 276
 refactoringto strategy pattern 255-259
 Replace Inheritance with Delegation 140, 142
 rewriting option 16-19
 simple refactoring 12
 technical debt 15
 to improved object-orientation 144-148
 to patterns 22
 to principles 22, 23
 to single class 128-131
 unit testing 11
 unused code 25
 virtual methods, to events 138-140
 working, into process 20

refactoring content coupling
 about 176
 delegates 179, 180
 events 180-183
 interface-based design 176, 178

- refactoring external coupling
 - about 183, 185
- refactoring, in Visual Studio 2010
 - about 26
 - code metrics 27
 - static code analysis 26
- refactoring subclass coupling 175
- refactorings, Visual Studio®
 - about 32
 - encapsulate field refactoring 39, 40
 - extract method refactoring 37, 38, 39
 - rename identifier refactoring 33
- Refactor! Pro 341
- Relational Database Management System (RDBMS) 221
- remove parameters, simple refactoring 14
- rename identifier refactoring
 - class, renaming 37
 - field, renaming 33-35
 - local variable, renaming 36
 - method, renaming 36
 - property, renaming 35
- rename method, simple refactoring 13
- rename variable, simple refactoring 12
- RenderReadableInvoice method 259
- RenderReadableInvoiceService 300
- reorder parameters, simple refactoring 13
- Replace Inheritance with Delegation 140, 142
- repository pattern 230
- Resharper 341
- Resolve method 198
- Rhino Mocks
 - about 321, 322
 - CreateInvoiceDataSet 324
 - CreateLineItemDataSet 324
 - IDataAccess 324
 - LoadInvoiceLineItems 324
 - MockRepository 324

S

- Sales namespace 114
- Sales sub-namespaces 114
- SatisfyingElementsFrom method 263
- SaveInvoice method 277
- scoping types, Convention over Configuration (CoC) pattern
 - with namespaces 104, 105
- search, IDE navigation
 - about 106
 - AddCustomerForm form 107
 - Find and Replace form 107
 - Find in Files mode 107
 - Match whole word options 107
 - whole word options 107
- Separate Query from Modifier refactoring 170
- Separation of Concerns (SoC) 75.
- service class 47
- service layer 248
- Service-oriented architectures (SOA) 144
- SetFaxNumber method 42
- SetPhoneNumber method 42
- ShapeDrawingService.Draw method 95
- ShapeDrawingService.Draw overloads 96
- Shape subclass 94
- Shape type 94
- simple refactoring, example
 - about 12
 - encapsulate field 13
 - extract interface 13
 - extract method 12
 - remove parameters 14
 - rename method 13
 - reorder parameters 13
- single class
 - Customer class 130
 - Customer/InactiveCustomer 132
 - Customer object 131
 - InactiveCustomer 131, 134
 - InactiveCustomer class 132
 - InvalidOperationException 131
 - refactoring to 128-131
- Single Responsibility principle,
 - Open-Closed principle, Liskov
 - Substitution principle, Interface
 - Segregation principle. *See* SOLID
- Single Responsibility Principle (SRP) 160
- SOLID 22

- SolidsDD**
 - about 42
 - URL 42
- solution explorer, IDE navigation**
 - about 111-113
 - FormTimer 113
 - Invoice classes 114
 - namespace hierarchy 113
 - Partner sub-namespaces 114
 - Sales namespace 114
 - SystemTimer 113
 - ThreadingTimer 113
- specification pattern**
 - about 259-261
 - And method 268, 271
 - AndSpecification<T> class 268
 - Combine method 271
 - example 259-261
 - Expression<> object 265
 - ExpressionParameterExchangerVisitor 269
 - ExpressionParameterExchangerVisitor class 269
 - FindBySpecification method 265
 - IEnumerable<T> implementation 262
 - Invoice instance 260
 - IQueryable<Invoice> instance 265
 - IQueryable<T> parameter 263
 - IsSatisfiedBy method 261
 - IsSatisfiedBy() method 265, 271
 - IsSatisfiedBy overload 264
 - need, detecting 261
 - Not method 268, 272
 - NotSpecification<T> class 268
 - Or method 268-272
 - OrSpecification<T> class 268
 - overdueInvoiceSpecification object 272
 - refactoring to 261-273
 - SatisfyingElementsFrom method 263
 - Specification<T> class 266
 - Specification<T> object 268
 - Specification<T> parameter 268
 - unpaidInvoiceSpecification 272
 - UnpaidInvoiceSpecification 265
 - UnpaidInvoiceSpecification class 262
 - UnpaidInvoiceSpecification instance 265
- Specification<T> class 266, 268**
- Specification<T> parameter 268**
- spy, test double 316**
- Square object 127**
- square/rectangle scenario 126**
- Stable Dependencies Principle (DSP) 187**
- Start method 248**
- static code analysis 26, 27**
- StatisticalInvoiceFactory object 205**
- strategy pattern**
 - about 253, 254
 - business logic 255
 - business logic, blocks 254
 - CalculateTaxReceivable method 257
 - example 254, 255
 - ExemptTaxCalculationStrategy 257
 - ExemptTaxCalculationStrategy class 257
 - InvoiceRenderingService class 258
 - InvoiceRenderingService.RenderReadableInvoice method 258
 - ITaxCalculation implementation 257
 - ITaxCalculation object 258
 - ITaxCalculationStrategy field 259
 - ITaxCalculationStrategy implementation 258
 - ITaxCalculationStrategy object 258
 - ITaxCalculationStrategy property 259
 - ITaxRate objects 257
 - need, detecting 254, 255
 - NormalTaxCalculationStrategy 257
 - refactoring to 255-259
 - RenderReadableInvoice method 259
 - subtype polymorphism 255
 - System.Console 254
 - System.Diagnostics.Debug 254
 - System.Trace 254
 - taxCalculationStrategy variable 259
 - TraceMethod variable traceMethod 254
 - usage criteria 255
- StreamWriter object 191**
- String.Empty 149**
- struct 150**
- structural patterns**
 - about 281
 - adapter pattern 283
 - façade pattern 291
 - legacy code 282
 - proxy pattern 299, 300
- stub, test double 316**

- subtype polymorphism 255
- System.Console 254
- System.Data.Linq namespace 174
- System.Data namespace 236
- System.DateTime 150
- System.Diagnostics.Debug 254
- System.Drawing namespace 78
- System.Threading.Timer object 192
- System.Trace 254

T

- taxCalculationStrategy variable 259
- TaxRate property 170
- test double
 - about 316
 - types 316
- test double, types
 - dummy 316
 - fake 316
 - mock 316
 - spy 316
 - stub 316
- Test-Driven Development (TDD)
 - about 332-334
 - CreateInvoiceForm 332
 - CreateInvoicePresenter constructor 334
 - CreateInvoicePresenterConstructorTest 334
 - CreateInvoicePresenter.View property 334
- Test-Driven-Development (TDD) process 8
- third party refactoring tools
 - about 341
 - Refactor! Pro 341
 - Resharper 341
- tightly-coupled
 - about 191
 - code, example 199, 200
 - example 191
- ToInvoiceLineItem 335
- TraceMethod variable traceMethod 254
- TypeMock isolator
 - about 327, 328
 - CreateInvoiceDataSet() 329
 - CreateInvoiceLineItemDataSet() 329
 - CreateLineItemDataSet() 329
 - DataAccess.LoadInvoice() 329
 - DataAccess.LoadInvoiceLineItem() 329

- DataAccess object 329
- LoadTest method 329

U

- Udi.dahan
 - Fallacy of Reuse 83
- UI data access
 - refactoring, to data access layer 221-233
- UI layer
- unit tests
 - about 11, 314, 315
 - business logic layer used 219
 - code coverage 320
 - in Visual Studio® 330-332
 - legacy code 325, 326
 - mocking 316
 - other testing 315
 - priorities 320
- UnpaidInvoiceSpecification class 262
- UnpaidInvoiceSpecification instance 265
- unused code 25
- unused references
 - removing 65
- Use() 150
- user interface layer 218, 237, 307
- using directives 65

V

- value types
 - about 150
 - immutable types 152
 - making immutable 152
 - refactoring to 150
- view 238
- ViewInvoiceForm 285
- ViewInvoiceForm class 231, 287, 290
- ViewInvoiceForm constructor 237
- virtual methods
 - refactoring, to events 138-140
- Visual Studio®
 - unit testing 330-332

W

- web services
 - about 215

implementing 215
WriteLine method 254

X

XP 8
XUnit 338, 339

Y

YAGNI 92. *See* You ain't gonna need it
You ain't gonna need it 61, 62



Thank you for buying **Refactoring with Microsoft Visual Studio 2010**

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

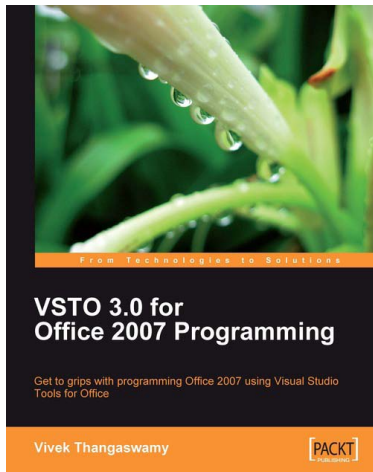
About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

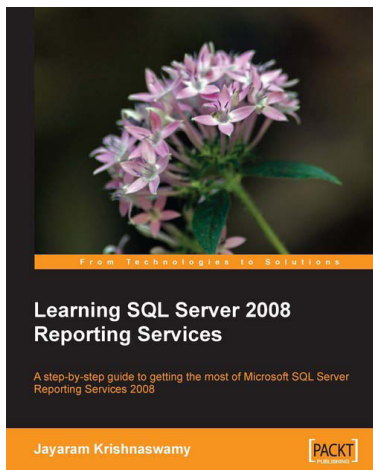


VSTO 3.0 for Office 2007 Programming

ISBN: 978-1-847197-52-8 Paperback: 260 pages

Get to grips with Programming Office 2007 using Visual Studio Tools for Office

1. A step-by-step guide for brand-new Office developers who want to explore programming with VSTO
2. Precise information on programming in Microsoft InfoPath, Word, Excel, PowerPoint, Outlook, Visio, and Project 2007 using VSTO
3. Create your own fully featured Office extensions



Learning SQL Server 2008 Reporting Services

ISBN: 978-1-847196-18-7 Paperback: 512 pages

A step-by-step guide to getting the most of Microsoft SQL Server Reporting Services 2008

1. Everything you need to create and deliver data-rich reports with SQL Server 2008 Reporting Services as quickly as possible
2. Packed with hands-on-examples to learn and improve your skills
3. Connect and report from databases, spreadsheets, XML Data, and more

Please check www.PacktPub.com for information on our titles