

Clean Code in C#

Refactor your legacy C# code base and improve application performance
by applying best practices



Packt>

www.packt.com

Jason Alls

Clean Code in C#

Refactor your legacy C# code base and improve application performance by applying best practices

Jason Alls



BIRMINGHAM - MUMBAI

Clean Code in C#

Copyright © 2020 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 or its dealers and distributors, will be held liable for any damages caused or alleged to have been 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.

Commissioning Editor: Richa Tripathi
Acquisition Editor: Alok Dhuri
Content Development Editor: Ruvika Rao
Senior Editor: Nitee Shetty
Technical Editor: Pradeep Sahu
Copy Editor: Safis Editing
Project Coordinator: Francy Puthiry
Proofreader: Safis Editing
Indexer: Manju Arasan
Production Designer: Nilesh Mohite

First published: July 2020

Production reference: 1170720

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83898-297-3

www.packt.com

To my parents, for supporting me throughout my life and career. To all the people in the world of software that have made my career possible, and who have employed me, trained me, and worked alongside me. You have been instrumental in helping me to get to where I am today.
I thank you all.



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Jason Alls has been programming for over 21 years using Microsoft technologies. Working with an Australasian company, he started his career developing call center management reporting software used by global clients including telecom providers, banks, airlines, and the police. He then moved on to develop GIS marketing applications and worked in the banking sector performing data migrations between Oracle and SQL Server. Certified as an MCAD in C# since 2005, he has been involved in the development of various desktop, web, and mobile applications.

Currently employed by a globally recognized leader in the educational software sector, he develops and supports dyslexia testing and assessment software written in ASP.NET, Angular, and C#.

I would like to thank my parents for always being there, supporting me throughout my life and career. Career-wise, I would like to thank all the people in the world of computing that have made my career possible. Especially those who have employed me, trained me, and worked alongside me. You have helped me to get to where I am today.

A special thank you to all the staff at Packt Publishing who provided me with the opportunity to write this book, and who assisted me in improving the content. It has been an eye-opening experience and a pleasant one. It is your hard work and dedication to the book-writing process that enables computer programmers like me to become accomplished authors. This book would not be what it is without your valuable input.

About the reviewer

Omprakash Pandey, a Microsoft 365 consultant, has been working with industry experts to understand project requirements and work on the implementation of projects for the last 20 years. He has trained more than 50,000 aspiring developers and has assisted in the development of more than 50 enterprise applications. He has offered innovative solutions on .NET development, Microsoft Azure, and other technologies. He has worked for multiple clients across various locations, including Hexaware, Accenture, Infosys, and many more. He has been a Microsoft Certified Trainer for more than 5 years.

I want to thank my parents, my colleagues, Ashish, and Francy, for their assistance and support.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Coding Standards and Principles in C#	7
Technical requirements	8
Good code versus bad code	9
Bad code	9
Improper indentation	10
Comments that state the obvious	10
Comments that excuse bad code	11
Commented-out lines of code	12
Improper organization of namespaces	12
Bad naming conventions	13
Classes that do multiple jobs	13
Methods that do many things	16
Methods with more than 10 lines of code	17
Methods with more than two parameters	18
Using exceptions to control program flow	18
Code that is difficult to read	19
Code that is tightly coupled	20
Low cohesion	21
Objects left hanging around	21
Use of the Finalize() method	22
Over-engineering	22
Learn to Keep It Simple, Stupid	22
Lack of regions in large classes	23
Lost-intention code	23
Directly exposing information	24
Good code	24
Proper indentation	25
Meaningful comments	25
API documentation comments	25
Proper organization using namespaces	26
Good naming conventions	26
Classes that only do one job	27
Methods that do one thing	27
Methods with less than 10 lines, and preferably no more than 4	27
Methods with no more than two parameters	28
Proper use of exceptions	28
Code that is readable	28
Code that is loosely coupled	29
High cohesion	29
Objects are cleanly disposed of	29
Avoiding the Finalize() method	30
The right level of abstraction	30

Using regions in large classes	31
The need for coding standards, principles, and methodologies	31
Coding standards	31
Coding principles	32
Coding methodologies	32
Coding conventions	33
Modularity	33
KISS	34
YAGNI	34
DRY	35
SOLID	35
Occam's Razor	36
Summary	36
Questions	37
Further reading	37
Chapter 2: Code Review – Process and Importance	38
The code review process	39
Preparing code for review	40
Leading a code review	41
Issuing a pull request	43
Responding to a pull request	46
Effects of feedback on reviewees	48
Knowing what to review	51
Company's coding guidelines and business requirement(s)	51
Naming conventions	51
Formatting	52
Testing	52
Architectural guidelines and design patterns	54
Performance and security	54
Knowing when to send code for review	55
Providing and responding to review feedback	57
Providing feedback as a reviewer	57
Responding to feedback as a reviewee	58
Summary	59
Questions	59
Further reading	60
Chapter 3: Classes, Objects, and Data Structures	61
Technical requirements	62
Organizing classes	62
A class should have only one responsibility	65
Commenting for documentation generation	67
Cohesion and coupling	70
An example of tight coupling	71

An example of low coupling	72
An example of low cohesion	73
An example of high cohesion	74
Design for change	75
Interface-oriented programming	76
Dependency injection and inversion of control	79
An example of DI	79
An example of IoC	81
The Law of Demeter	83
A good and a bad example (chaining) of the Law of Demeter	83
Immutable objects and data structures	85
An example of an immutable type	86
Objects should hide data and expose methods	87
An example of encapsulation	87
Data structures should expose data and have no methods	88
An example of data structure	89
Summary	89
Questions	90
Further reading	90
Chapter 4: Writing Clean Functions	91
Understanding functional programming	92
Keeping methods small	95
Indenting code	97
Avoiding duplication	99
Avoiding multiple parameters	100
Implementing SRP	102
Summary	107
Questions	107
Further reading	108
Chapter 5: Exception Handling	109
Checked and unchecked exceptions	110
Avoiding NullPointerExceptions	114
Business rule exceptions	117
Example 1 – handling conditions with business rule exceptions	120
Example 2 – handling conditions with normal program flow	121
Exceptions should provide meaningful information	123
Building your own custom exceptions	125
Summary	128
Questions	129
Further reading	129
Chapter 6: Unit Testing	130
Technical Requirements	131

Understanding the reasons for a good test	131
Understanding the testing tools	137
MSTest	137
NUnit	145
Moq	152
SpecFlow	157
TDD methodology practice – fail, pass, and refactor	162
Removing redundant tests, comments, and dead code	169
Summary	171
Questions	171
Further reading	172
Chapter 7: End-to-End System Testing	173
E2E testing	173
The login module (subsystem)	175
The admin module (subsystem)	179
The test module (subsystem)	181
Testing our three-module system using E2E	182
Factories	185
Dependency injection	194
Modularization	200
Summary	202
Questions	203
Further reading	203
Chapter 8: Threading and Concurrency	204
Understanding the thread life cycle	205
Adding thread parameters	207
Using a thread pool	208
Task Parallel Library	209
Parallel.Invoke()	209
Parallel.For()	210
ThreadPool.QueueUserWorkItem()	212
Using a mutex with synchronous threads	212
Working with parallel threads using semaphores	215
Limiting the number of processors and threads in the thread pool	217
Preventing deadlocks	219
Coding a deadlock example	220
Preventing race conditions	225
Understanding static constructors and methods	228
Adding static constructors to our sample code	229
Adding static methods to our sample code	230
Mutability, immutability, and thread safety	234
Writing code that is mutable and not thread-safe	234
Writing code that is immutable and thread-safe	236

Understanding thread safety	238
Synchronized method dependencies	242
Using the Interlocked class	243
General recommendations	247
Summary	248
Questions	249
Further reading	249
Chapter 9: Designing and Developing APIs	251
Technical requirements	252
What is an API?	252
API proxies	254
API design guidelines	256
Well-defined software boundaries	259
Understanding the importance of good quality API documentation	261
Swagger API development	262
Passing immutable structs instead of mutable objects	265
Testing third-party APIs	268
Testing your own APIs	269
API design using RAML	271
Installing Atom and API Workbench by MuleSoft	272
Creating the project	274
Generating our C# API from our agnostic RAML design specification	277
Summary	281
Questions	281
Further reading	282
Chapter 10: Securing APIs with API Keys and Azure Key Vault	283
Technical requirements	284
Undertaking the API project – dividend calendar	284
Accessing the Morningstar API	286
Storing the Morningstar API key in Azure Key Vault	286
Creating the dividend calendar ASP.NET Core web application in Azure	289
Publishing our web application	291
Using an API key to secure our dividend calendar API	297
Setting up the repository	297
Setting up authentication and authorization	300
Adding authentication	300
Adding authorization	304
Testing our API key security	308
Adding the dividend calendar code	311
Throttling our API	319
Summary	323
Questions	324

Further reading	324
Chapter 11: Addressing Cross-Cutting Concerns	326
Technical requirements	327
The decorator pattern	327
The proxy pattern	331
AOP with PostSharp	333
Extending the aspect framework	334
Developing our aspect	334
Injecting behaviors before and after the method execution	334
Extending the architectural framework	337
Project – cross-cutting concerns reusable library	338
Adding the caching concern	338
Adding file logging capabilities	340
Adding the logging concern	341
Adding the exception-handling concern	342
Adding the security concern	344
Adding the validation concern	348
Adding the transaction concern	352
Adding the resource pool concern	353
Adding the configuration settings concern	354
Adding the instrumentation concern	355
Summary	356
Questions	356
Further reading	356
Chapter 12: Using Tools to Improve Code Quality	357
Technical requirements	358
Defining good-quality code	358
Performing code cleanup and calculating code metrics	360
Performing code analysis	363
Using quick actions	366
Using the JetBrains dotTrace profiler	367
Using JetBrains ReSharper	372
Using Telerik JustDecompile	382
Summary	384
Questions	385
Further reading	385
Chapter 13: Refactoring C# Code – Identifying Code Smells	386
Technical requirements	387
Application-level code smells	387
Boolean blindness	387
Combinatorial explosion	389
Contrived complexity	390

Data clump	391
Deodorant comments	391
Duplicate code	392
Lost intent	392
The mutation of variables	393
The oddball solution	395
Shotgun surgery	397
Solution sprawl	399
Uncontrolled side effects	399
Class-level code smells	400
Cyclomatic complexity	400
Replacing switch statements with the factory pattern	400
Improving the readability of conditional checks within an if statement	403
Divergent change	404
Downcasting	405
Excessive literal use	405
Feature envy	405
Inappropriate intimacy	407
Indecent exposure	408
The large class (aka the God object)	408
The lazy class (aka the freeloader and the lazy object)	408
The middleman class	409
The orphan class of variables and constants	409
Primitive obsession	409
Refused bequest	410
Speculative generality	410
Tell, Don't Ask	410
Temporary fields	410
Method-level smells	411
The black sheep method	411
Cyclomatic complexity	411
Contrived complexity	411
Dead code	411
Excessive data return	412
Feature envy	412
Identifier size	412
Inappropriate intimacy	412
Long lines (aka God lines)	413
Lazy methods	413
Long methods (aka God methods)	413
Long parameter lists (aka too many parameters)	413
Message chains	413
The middleman method	414
Oddball solutions	414
Speculative generality	414
Summary	414

Questions	415
Further reading	416
Chapter 14: Refactoring C# Code – Implementing Design Patterns	417
Technical requirements	418
Implementing creational design patterns	418
Implementing the singleton pattern	419
Implementing the factory method pattern	420
Implementing the abstract factory pattern	422
Implementing the prototype pattern	425
Implementing the builder pattern	427
Implementing structural design patterns	433
Implementing the bridge pattern	434
Implementing the composite pattern	436
Implementing the façade pattern	439
Implementing the flyweight pattern	442
Overview of behavioral design patterns	445
Final thoughts	446
Summary	448
Questions	449
Further reading	450
Appendix A: Assessments	451
Chapter 1	451
Chapter 2	451
Chapter 3	452
Chapter 4	452
Chapter 5	453
Chapter 6	454
Chapter 7	455
Chapter 8	455
Chapter 9	456
Chapter 10	457
Chapter 11	457
Chapter 12	458
Chapter 13	458
Chapter 14	460
Other Books You May Enjoy	462
Index	465

Preface

Welcome to *Clean Code in C#*. You will learn how to identify problematic code that, while it compiles, does not lend itself to readability, maintainability, and extensibility. You will also learn about various tools and patterns, along with ways to refactor code to make it clean.

Who this book is for

This book is aimed at computer programmers with a good grasp of the C# programming language who would like guidance on identifying problematic code and writing clean code in C#. Primarily, the reader base will range from graduate to mid-level programmers, but even senior programmers may find this book valuable.

What this book covers

Chapter 1, *Coding Standards and Principles in C#*, looks at some good code contrasted with bad code. As you read through this chapter, you will come to understand why you need coding standards, principles, methodologies, and code conventions. You will learn about modularity and the design guidelines KISS, YAGNI, DRY, SOLID, and Occam's razor.

Chapter 2, *Code Review – Process and Importance*, takes you through the code review process and provides reasons for its importance. In this chapter, you are guided through the process of preparing code for review, leading a code review, knowing what to review, knowing when to send code for review, and how to provide and respond to review feedback.

Chapter 3, *Classes, Objects, and Data Structures*, covers the broad topics of class organization, documentation comments, cohesion, coupling, the Law of Demeter, and immutable objects and data structures. By the end of the chapter, you will be able to write code that is well organized and only has a single responsibility, provide users of the code with relevant documentation, and make code extensible.

Chapter 4, *Writing Clean Functions*, helps you to understand functional programming, how to keep methods small, and how to avoid code duplication and multiple parameters. By the time you finish this chapter, you will be able to describe functional programming, write functional code, avoid writing code with more than two parameters, write immutable data objects and structures, keep your methods small, and write code that adheres to the Single Responsibility Principle.

Chapter 5, *Exception Handling*, covers checked and unchecked exceptions, `NullPointerException`, and how to avoid them as well as covering, business rule exceptions, providing meaningful data, and building your own custom exceptions.

Chapter 6, *Unit Testing*, takes you through using the **Behavior-Driven Development (BDD)** software methodology using SpecFlow, and **Test-Driven Development (TDD)** using MSTest and NUnit. You will learn how to write mock (fake) objects using Moq, and how to use the TDD software methodology to write tests that fail, make the tests pass, and then refactor the code once it passes.

Chapter 7, *End-to-End System Testing*, guides you through the manual process of end-to-end testing using an example project. In this chapter, you will perform **End-to-End (E2E)** testing, code and test factories, code and test dependency injection, and test modularization. You will also learn how to utilize modularization.

Chapter 8, *Threading and Concurrency*, focuses on understanding the thread life cycle; adding parameters to threads; using `ThreadPool`, mutexes, and synchronous threads; working with parallel threads using semaphores; limiting the number of threads and processors used by `ThreadPool`; preventing deadlocks and race conditions; static methods and constructors; mutability and immutability; and thread-safety.

Chapter 9, *Designing and Developing APIs*, helps you to understand what an API is, API proxies, API design guidelines, API design using RAML, and Swagger API development. In this chapter, you will design a language-agnostic API in RAML and develop it in C#, and you will document your API using Swagger.

Chapter 10, *Securing APIs with API Keys and Azure Key Vault*, shows you how to obtain a third-party API key, store that key in Azure Key Vault, and retrieve it via an API that you will build and deploy to Azure. You will then implement API key authentication and authorization to secure your own API.

Chapter 11, *Addressing Cross-Cutting Concerns*, introduces you to using PostSharp to address cross-cutting concerns using aspects and attributes that form the basis of aspect-oriented development. You will also learn how to use proxies and decorators.

Chapter 12, *Using Tools to Improve Code Quality*, exposes you to various tools that will assist you in writing quality code and improving the quality of existing code. You'll gain exposure to code metrics and code analysis, quick actions, the JetBrains tools called dotTrace Profiler and Resharper, and Telerik JustDecompile.

Chapter 13, *Refactoring C# Code – Identifying Code Smells*, is the first of two chapters that take you through different types of problematic code and show you how to modify it to be clean code that is easy to read, maintain, and extend. Code problems are listed alphabetically through each chapter. Here, you will cover such topics as class dependencies, code that can't be modified, collections, and combinatorial explosion.

Chapter 14, *Refactoring C# Code – Implementing Design Patterns*, takes you through the implementation of creational and structural design patterns. Here, behavioral design patterns are briefly touched upon. You are then given some final thoughts on clean code and refactoring.

To get the most out of this book

The majority of the chapters can be read independently of each other and in any order. But to get the most out of this book, I recommend that you read the chapters in the order presented. As you work through the chapters, follow the instructions, and carry out the tasks presented. Then, when you reach the end of a chapter, answer the questions and carry out the recommended further reading to reinforce what you have learned. For maximum benefit when working through the contents of this book, it is recommended that you meet the following requirements:

Software/hardware covered in the book	Requirements
Visual Studio 2019	Windows 10, macOS
Atom	Windows 10, macOS, Linux: https://atom.io/
Azure resources	Azure subscription: https://azure.microsoft.com/en-gb/
Azure Key Vault	Azure subscription: https://azure.microsoft.com/en-gb/
The Morningstar API	Obtain your own API key from https://rapidapi.com/integraatio/api/morningstar1
Postman	Windows 10, macOS, Linux: https://www.postman.com/

It will be useful if you have these in place before you start reading and working your way through the chapters.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

You should have basic experience of using Visual Studio 2019 Community Edition or higher, and basic C# programming skills, including writing console applications. Many examples will be in the form of C# console applications. The main project will be using ASP.NET. It will help if you are capable of writing ASP.NET websites using the framework and core. But don't worry – you will be guided through the steps that you need to go through.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Clean-Code-in-C->. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781838982973_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `InMemoryRepository` class implements the `GetApiKey()` method of `IRepository`. This returns a dictionary of API keys. These keys will be stored in our `_apiKeys` dictionary member variable."

A block of code is set as follows:

```
using CH10_DividendCalendar.Security.Authentication;
using System.Threading.Tasks;

namespace CH10_DividendCalendar.Repository
{
    public interface IRepository
    {
        Task<ApiKey> GetApiKey(string providedApiKey);
    }
}
```

Any command-line input or output is written as follows:

```
az group create --name "<YourResourceGroupName>" --location "East US"
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "To create the app service, right-click the project you created and select **Publish** from the menu."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customer-care@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Coding Standards and Principles in C#

The primary goal of coding standards and principles in C# is for programmers to become better at their craft by programming code that is more performant and easier to maintain. In this chapter, we will look at some examples of good code contrasted with examples of bad code. This will lead nicely into discussing why we need coding standards, principles, and methodologies. We will then move on to consider conventions for naming, commenting, and formatting source code, including classes, methods, and variables.

A big program can be rather unwieldy to understand and maintain. For junior programmers, getting to know the code and what it does can be a daunting prospect. Teams can find it hard to work together on such projects. And from a testing viewpoint, it can make things rather difficult. Because of this, we will look at how you use modularity to break programs down into smaller modules that all work together to produce a fully functioning solution that is also fully testable, can be worked on by multiple teams simultaneously, and is much easier to read, understand, and document.

We will finish the chapter off by looking at some programming design guidelines, mainly, KISS, YAGNI, DRY, SOLID, and Occam's Razor.

The following topics will be covered in this chapter:

- The need for coding standards, principles, and methodologies
- Naming conventions and methods
- Comments and formatting
- Modularity
- KISS
- YAGNI
- DRY
- SOLID
- Occam's Razor

The learning objectives for this chapter are for you to do the following:

- Understand why bad code negatively impacts projects.
- Understand how good code positively impacts projects.
- Understand how coding standards improve code and how to enforce them.
- Understand how coding principles enhance software quality.
- Understand how methodologies aid the development of clean code.
- Implement coding standards.
- Choose solutions with the least assumptions.
- Reduce code duplication and write SOLID code.

Technical requirements

To work on the code in this chapter, you will need to download and install Visual Studio 2019 Community Edition or higher. This IDE can be downloaded from <https://visualstudio.microsoft.com/>.

You will find the code for this book located at <https://github.com/PacktPublishing/Clean-Code-in-C->. I have put them all under a single solution with each chapter as a solution folder. You will find the code for each chapter in the relevant chapter folder. If running a project, remember to assign it as the startup project.

Good code versus bad code

Both good code and bad code compile. That's the first thing to understand. The next thing to understand is that bad code is bad for a reason, and likewise, good code is good for a reason. Let's have a look at some of those reasons in the following comparison table:

Good Code	Bad Code
Proper indentation.	Improper indentation.
Meaningful comments.	Comments that state the obvious.
API documentation comments.	Comments that excuse bad code. Commented out lines of code.
Proper organization using namespaces.	An improper organization using namespaces.
Good naming conventions.	Bad naming conventions.
Classes that do one job.	Classes that do multiple jobs.
Methods that do one thing.	Methods that do many things.
Methods with less than 10 lines, and preferably no more than 4.	Methods with more than 10 lines of code.
Methods with no more than two parameters.	Methods with more than two parameters.
Proper use of exceptions.	Using exceptions to control program flow.
Code that is readable.	Code that is difficult to read.
Code that is loosely coupled.	Code that is tightly coupled.
High cohesion.	Low cohesion.
Objects are cleanly disposed of.	Objects left hanging around.
Avoidance of the <code>Finalize()</code> method.	Use of the <code>Finalize()</code> method.
The right level of abstraction.	Over-engineering.
Use of regions in large classes.	Lack of regions in large classes.
Encapsulation and information hiding.	Directly exposing information.
Object-oriented code.	Spaghetti code.
Design patterns.	Design anti-patterns.

That's quite an exhaustive list, isn't it? In the following sections, we will look at how these features and the differences between good and bad code impact the performance of your code.

Bad code

We will now take a brief look at each of the bad coding practices that we listed earlier, detailing specifically how that practice affects your code.

Improper indentation

Improper indentation can work toward making code really hard to read, especially if the methods are large. For code to be easy to read by humans, we need proper indentation. If code lacks proper indentation it can be very hard to see which part of the code belongs to which block.

By default, Visual Studio 2019 correctly formats and indents your code when parentheses and braces are closed. But sometimes, it incorrectly formats the code, to bring to your attention that the code you've written contains an exception. But if you are using a simple text editor, then you will have to do your formatting by hand.

Incorrectly indented code is also time-consuming to correct, and a frustrating waste of programming time when it could easily have been avoided. Let's look at a simple code example:

```
public void DoSomething()  
{  
    for (var i = 0; i < 1000; i++)  
    {  
        var productCode = $"PRC000{i}";  
        //...implementation  
    }  
}
```

The preceding code does not look all that nice, yet it is still readable. But the more lines of code you add, the harder the code becomes to read.

It is very easy to miss a closing bracket. If your code is not properly indented, then this can make finding the missing bracket that much harder, as you can not easily spot which code block is missing its closing bracket.

Comments that state the obvious

I've seen programmers get really upset at comments that state the obvious as they find them patronizing. In programming discussions that I have been part of, programmers have stated how they dislike comments, and how they believe the code should be self-documenting.

I can understand their sentiments. If you can read code without comments like you can read a book and understand it, then it is a really good piece of code. If you have a variable declared as a string, then why add a comment such as `// string`? Let's look at an example:

```
public int _value; // This is used for storing integer values.
```

We know here that the value holds an integer by its type of `int`. So there really is no need to state the obvious. All you're doing is wasting time and energy and cluttering up the code.

Comments that excuse bad code

You may have a tight deadline to meet, but comments such as `// I know this code sucks but hey at least it works!` are just awful. Don't do it. It shows a lack of professionalism and can really disgruntle fellow programmers.

If you really are pushed to get something working out the door, raise a refactor ticket and add it as part of a `TODO` comment such as `// TODO: PBI23154 Refactor Code to meet company coding practices`. Then you or the other developers who are assigned to work on technical debt can pick up the **Product Backlog Item (PBI)** and refactor the code.

Here's another example:

```
...
int value = GetDataValue(); // This sometimes causes a divide by zero
error. Don't know why!
...
```

This one is really bad. Okay, thank you for letting us know that divide-by-zero errors occur here. But have you raised a bug ticket? Have you tried to get to the bottom of it and fix it? If everybody who is actively working on the project does not touch that code, how will they know that buggy code is there?

At the very minimum, you should at least have a `// TODO: comment` in place. Then at least the comment will show up in the **Task List** so that developers can be notified and work on it.

Commented-out lines of code

If you comment out lines of code to try something, fine. But if you are going to use the replacement code instead of the commented-out code, then delete the commented-out code before you check it in. One or two commented outlines is not that bad. But when you have many lines of commented-out code, it becomes distracting and makes code hard to maintain; it can even lead to confusion:

```
/* No longer used as has been replaced by DoSomethinElse().
public void DoSomething()
{
    // ...implementation...
}
*/
```

Why? Just why? If it has been replaced and is no longer needed, then just delete it. If your code is in version control, and you need to get the method back, then you can always view the history of the file and get the method back.

Improper organization of namespaces

When using namespaces, do not include code that should be elsewhere. This can make finding the right code pretty hard or impossible, especially in large code bases. Let's look at this example:

```
namespace MyProject.TextFileMonitor
{
    + public class Program { ... }
    + public class DateTime { ... }
    + public class FileMonitorService { ... }
    + public class Cryptography { ... }
}
```

We can see that all classes in the preceding code are under one namespace. Yet, we have the opportunity to add three further namespaces to better organize this code:

- `MyProject.TextFileMonitor.Core`: Core classes that define commonly used members will be placed here, such as our `DateTime` class.
- `MyProject.TextFileMonitor.Services`: All classes that act as a service will be placed in this namespace, such as `FileMonitorService`.
- `MyProject.TextFileMonitor.Security`: All security-related classes will be placed in this namespace, including the `Cryptography` class in our example.

Bad naming conventions

In the days of Visual Basic 6 programming, we used to use Hungarian Notation. I remember using it when I first switched to Visual Basic 1.0. It is no longer necessary to use Hungarian Notation. Plus, it makes your code look ugly. So instead of using names such as `lblName`, `txtName`, or `btnSave`, the modern way is to use `NameLabel`, `NameTextBox`, and `SaveButton`, respectively

The use of cryptic names and names that don't seem to match the intention of the code can make reading code rather difficult. What does `ihridx` mean? It means **Human Resources Index** and is an *integer*. Really! Avoid using names such as `mystring`, `myint`, and `mymethod`. Such names really don't serve a purpose.

Don't use underscores between words in a name either, such as `Bad_Programmer`. This can cause visual stress for developers and can make the code hard to read. Simply remove the underscore.

Don't use the same code convention for variables at the class level and method level. This can make it difficult to establish the scope of a variable. A good convention for variable names is to use camel case for variable names such as `alienSpawn`, and Pascal case for method, class, struct, and interface names such as `EnemySpawnGenerator`.

Following the good variable name convention, you should distinguish between local variables (those contained within a constructor or method), and member variables (those placed at the top of the class outside of constructors and methods) by prefixing the member variables with an underscore. I have used this as a coding convention in the workplace, and it does work really well and programmers do seem to like this convention.

Classes that do multiple jobs

A good class should only do one job. Having a class that connects to a database, gets data, manipulates that data, loads a report, assigns the data to the report, displays the report, saves the report, prints the reports, and exports the report is doing too much. It needs to be refactored into smaller, better-organized classes. All-encompassing classes like this are a pain to read. I personally find them daunting. If you come across classes like this, organize the functionality into regions. Then move the code in those regions into new classes that perform one job.

Let's have a look at an example of a class that is doing multiple things:

```
public class DbAndFileManager
{
    #region Database Operations
```

```
public void OpenDatabaseConnection() { throw new
    NotImplementedException(); }
public void CloseDatabaseConnection() { throw new
    NotImplementedException(); }
public int ExecuteSql(string sql) { throw new
    NotImplementedException(); }
public SqlDataReader SelectSql(string sql) { throw new
    NotImplementedException(); }
public int UpdateSql(string sql) { throw new
    NotImplementedException(); }
public int DeleteSql(string sql) { throw new
    NotImplementedException(); }
public int InsertSql(string sql) { throw new
    NotImplementedException(); }

#endregion

#region File Operations

public string ReadText(string filename) { throw new
    NotImplementedException(); }
public void WriteText(string filename, string text) { throw new
    NotImplementedException(); }
public byte[] ReadFile(string filename) { throw new
    NotImplementedException(); }
public void WriteFile(string filename, byte[] binaryData) { throw new
    NotImplementedException(); }

#endregion
}
```

As you can see in the preceding code, the class does two main things: it performs database operations and it performs file operations. Now the code is neatly organized within correctly named regions used to logically separate code within a class. But the **Single Responsibility Principle (SRP)** is broken. We would need to begin by refactoring this code to separate out the database operations into a class of their own, called something like `DatabaseManager`.

Then, we would remove the database operations from the `DbAndFileManager` class, leaving only the file operations, and then rename the `DbAndFileManager` class to `FileManager`. We would also need to consider the namespace of each file, and whether it should be modified so that the `DatabaseManager` would be placed in the `Data` namespace and the `FileManager` would be placed in the `FileSystem` namespace, or their equivalents in your program.

The following code is the result of extracting the database code from the `DbAndFileManager` class into its own class and in the correct namespace:

```
using System;
using System.Data.SqlClient;

namespace CH01_CodingStandardsAndPrinciples.GoodCode.Data
{
    public class DatabaseManager
    {
        #region Database Operations

        public void OpenDatabaseConnection() { throw new
            NotImplementedException(); }
        public void CloseDatabaseConnection() { throw new
            NotImplementedException(); }
        public int ExecuteSql(string sql) { throw new
            NotImplementedException(); }
        public SqlDataReader SelectSql(string sql) { throw new
            NotImplementedException(); }
        public int UpdateSql(string sql) { throw new
            NotImplementedException(); }
        public int DeleteSql(string sql) { throw new
            NotImplementedException(); }
        public int InsertSql(string sql) { throw new
            NotImplementedException(); }

        #endregion
    }
}
```

The refactoring of the filesystem code results in the `FileManager` class in the `FileSystem` namespace, as shown in the following code:

```
using System;

namespace CH01_CodingStandardsAndPrinciples.GoodCode.FileSystem
{
    public class FileManager
    {
        #region File Operations

        public string ReadText(string filename) { throw new
            NotImplementedException(); }
        public void WriteText(string filename, string text) { throw new
            NotImplementedException(); }
        public byte[] ReadFile(string filename) { throw new
            NotImplementedException(); }

        #endregion
    }
}
```

```
        public void WriteFile(string filename, byte[] binaryData) { throw
            new NotImplementedException(); }

        #endregion
    }
}
```

We've seen how to identify classes that do too much, and how we can refactor them to do only a single thing. Now let's repeat the process as we look at methods that do many things.

Methods that do many things

I have found myself getting lost in methods with many, many levels of indentation doing many things in those various indentations. The permutations were mind-boggling. I wanted to refactor the code to make maintenance easier, but my senior prohibited it. I could clearly see how the method could have been smaller by farming out the code to different methods.

Time for an example. In this example, the method accepts a string. That string is then encrypted and decrypted. It is also long so that you can see why methods should be kept small:

```
public string security(string plainText)
{
    try
    {
        byte[] encrypted;
        using (AesManaged aes = new AesManaged())
        {
            ICryptoTransform encryptor = aes.CreateEncryptor(Key, IV);
            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor,
                    CryptoStreamMode.Write))
                {
                    using (StreamWriter sw = new StreamWriter(cs))
                    {
                        sw.Write(plainText);
                    }
                    encrypted = ms.ToArray();
                }
            }
        }
        Console.WriteLine($"Encrypted data:
            {System.Text.Encoding.UTF8.GetString(encrypted)}");
        using (AesManaged aesm = new AesManaged())
        {
            ICryptoTransform decryptor = aesm.CreateDecryptor(Key, IV);
            using (MemoryStream ms = new MemoryStream(encrypted))
            {

```

```
        using (CryptoStream cs = new CryptoStream(ms, decryptor,
            CryptoStreamMode.Read))
        {
            using (StreamReader reader = new StreamReader(cs))
                plainText = reader.ReadToEnd();
        }
    }
    Console.WriteLine($"Decrypted data: {plainText}");
}
catch (Exception exp)
{
    Console.WriteLine(exp.Message);
}
Console.ReadKey();
return plainText;
}
```

As you can see in the preceding method, it has 10 lines of code and is hard to read. Plus, it is doing more than one thing. This code can be broken down into two methods that each perform a single task. One method would encrypt a string, and the other method would decrypt the string. This leads us nicely into why methods should have no more than 10 lines of code.

Methods with more than 10 lines of code

Large methods are not nice to read and understand. They can also lead to very hard-to-find bugs. Another problem with large methods is they can lose sight of their original intent. It's even worse when you come across large methods that have sections separated by comments and code wrapped in regions.

If you have to scroll to read a method, then it is too long and can lead to programmer stress and misinterpretation. This in turn can lead to modifications that will break the code or the intent, or both. Methods should be as small as you can make them. But common sense does need to be exercised, as you can take the matter of small methods to the n^{th} degree to the point that it becomes excessive. The key to getting the right balance is to ensure the intent of the method is very clear and succinctly implemented.

The previous code is a good candidate for why you should keep methods small. Small methods are easy to read and understand. Normally, if your code drifts beyond 10 lines it may be doing more than it is intended to. Make sure your methods name their intentions, as in `OpenDatabaseConnection()` and `CloseDatabaseConnection()`, and that they stick to their intentions and do not deviate away from them.

We are now going to take a look at method parameters.

Methods with more than two parameters

Methods with many parameters tend to get a bit unwieldy. Apart from being hard to read, it is very easy to pass a value to the wrong parameter and break type safety.

Testing methods get increasingly more complex as the number of parameters increases, the main reason being that you have more permutations to apply to your test cases. It is possible that you will miss a use case that will cause issues in production.

Using exceptions to control program flow

Exceptions used to control program flow may hide the intention of the code. They can also lead to unexpected and unintended results. The very fact that your code has been programmed to expect one or more exceptions shows your design to be wrong. A typical scenario that is covered in more detail in [Chapter 5, Exception Handling](#).

A typical scenario is when a business uses **Business Rule Exceptions (BREs)**. A method will perform an action anticipating that an exception will be thrown. The program flow will be determined by whether the exception is thrown or not. A much better way is to use available language constructs to perform validation checks that return a Boolean value.

The following code shows the use of a BRE to control program flow:

```
public void BreFlowControlExample (BusinessRuleException bre)
{
    switch (bre.Message)
    {
        case "OutOfAcceptableRange":
            DoOutOfAcceptableRangeWork ();
            break;
        default:
            DoInAcceptableRangeWork ();
            break;
    }
}
```

The method accepts `BusinessRuleException`. Depending upon the message in the exception, `BreFlowControlExample()` either calls the `DoOutOfAcceptableRangeWork()` method or the `DoInAcceptableRangeWork()` method.

A much better way to control the flow is through Boolean logic. Let's look at the following `BetterFlowControlExample()` method:

```
public void BetterFlowControlExample(bool isInAcceptableRange)
{
    if (isInAcceptableRange)
        DoInAcceptableRangeWork();
    else
        DoOutOfAcceptableRangeWork();
}
```

In the `BetterFlowControlExample()` method, a Boolean value is passed into the method. The Boolean value is used to determine which path to execute. If the condition is in the acceptable range, then `DoInAcceptableRangeWork()` is called. Otherwise, the `DoOutOfAcceptableRangeWork()` method is called.

Next, we will consider code that is difficult to read.

Code that is difficult to read

Code such as lasagna and spaghetti code is really hard to read or follow. Badly named methods can also be a pain as they can obfuscate the intention of the method. Methods are further obfuscated if they are large and if linked methods are separated by a number of unrelated methods.

Lasagna code, also known more commonly as indirection, refers to layers of abstraction where something is referred to by name rather than by action. Layering is used extensively in **Object-Oriented Programming (OOP)** and to good effect. However, the more indirection is used, the more complex code can become. This can make it very hard for new programmers on a project to get up to speed with understanding the code. So there must be a balance struck between indirection and ease of understanding.

Spaghetti code refers to a tangled mess of tightly coupled code with low cohesion. Such code is very hard to maintain, refactor, extend, and redesign. Though on the plus side, it can be very easy to read and follow since it is more procedural in its programming. I remember working as a junior programmer on a VB6 GIS program that was sold to companies and used for marketing purposes. My technical director and his senior programmers had previously tried to redesign the software and failed. So they passed the gauntlet to me so that I would redesign the program. But not being skilled in software analysis and design at the time, I also failed.

The code was just too complex to follow and group into related items, and it was way too big. With hindsight, I would have been better off making a list of everything the program did, grouping the list by features, and then coming up with a list of requirements without even looking at the code.

So my lesson learned when redesigning software is to avoid looking at the code at all costs. Write down everything the program does, and what the new functionality is that it should include. Turn the list into a set of software requirements with associated tasks, tests, and acceptance criteria, and then program to the specifications.

Code that is tightly coupled

Code that is tightly coupled is hard to test and hard to extend or modify. It is also hard to reuse code that is dependent on other code within a system.

An example of tight coupling is when you reference a concrete class type in the parameter rather than referencing an interface. When referencing a concrete class, any changes to the concrete class directly affect the class that references it. So if you have a database connection class for a client that connects to SQL Server, and then takes on another customer that requires an Oracle database, then the concrete class would have to be modified for that specific customer and their Oracle database. That would lead to two versions of the code.

The more customers there are, the more versions of the code required. This soon becomes untenable and a right nightmare to maintain. Imagine that your database connection class has 100,000 different clients using 1 of 30 variations of the class, and they all have the same bug that has been identified and affects them all. That is 30 classes that have to have the same fix put in place, tested, packaged, and deployed. That's a lot of maintenance overhead, and very costly financially.

This particular scenario can be overcome by referencing an interface type and then using a database factory to build the required connection object. Then the connection string can be set in a configuration file by the customer and passed into the factory. The factory would then produce a concrete connection class that implements a connection interface for the specific type of database specified in the connection string.

Here is a bad example of tightly coupled code:

```
public class Database
{
    private SqlConnection _databaseConnection;

    public Database(SqlConnection databaseConnection)
```

```
{  
    _databaseConnection = databaseConnection;  
}
```

As you can see from the example, our database class is tied to using SQL Server and would require a hardcoded change to accept any other type of database. We will be covering refactoring of code in later chapters with actual code examples.

Low cohesion

Low cohesion consists of unrelated code that performs a variety of different tasks all grouped together. An example would be a utility class that contains a number of different utility methods for handling dates, text, numbers, doing file input and output, data validation, and encryption and decryption.

Objects left hanging around

When objects are left hanging around in memory, they can lead to memory leaks.

Static variables can lead to memory leaks in several ways. If you're not using `DependencyObject` or `INotifyPropertyChanged`, then you are effectively subscribing to events. The **Common Language Runtime (CLR)** creates a strong reference by using the `ValueChanged` event via the `PropertyDescriptor` `AddValueChanged` event, which results in the storage of `PropertyDescriptor` that references the object it is bound to.

Unless you unsubscribe your bindings, you will end up with a memory leak. You will also end up with memory leaks using static variables that reference objects that don't get released. Any object that is referenced by a static variable is marked as not to be collected by the garbage collector. This is because static variables that reference objects are **Garbage Collection (GC)** roots, and anything that is a GC root is marked by the garbage collector as *do not collect*.

When you use anonymous methods that capture class members, the instance of the class is referenced. This causes a reference to the class instance to remain alive while the anonymous methods stay alive.

When using **unmanaged code (COM)**, if you do not release any managed and unmanaged objects and explicitly deallocate any memory, then you will end up with memory leaks.

Code that caches indefinitely without using weak references, deleting unused cache, or limiting the cache size will eventually run out of memory.

You would also end up with a memory leak if you were to create object references in a thread that never terminates.

Event subscriptions that are not anonymous reference classes. While these events remain subscribed to, the objects will remain in memory. So unless you unsubscribe from events when they are not needed, it is likely you will end up with a memory leak.

Use of the Finalize() method

While finalizers can help free up resources from objects that have not been correctly disposed of and help to prevent memory leaks, they do have a number of drawbacks.

You do not know when finalizers will be called. They will be promoted by the garbage collector along with all dependants on the graph to the next generation, and will not be garbage-collected until the garbage collector decides to do so. This can mean that objects stay in memory for a long time. Out-of-memory exceptions could occur using finalizers as you can be creating objects faster than they are getting garbage-collected.

Over-engineering

Over-engineering can be an utter nightmare. The biggest reason for this is that as a mere human, wading through a massive system, trying to understand it, how you are to use it, and what goes where is a time-consuming process. All the more so when there is no documentation, you are new to the system, and even people who have been using it much longer than you are unable to answer your questions.

This can be a major cause of stress when you are expected to work on it with set deadlines.

Learn to Keep It Simple, Stupid

A good example of this is at one of the places I've worked. I had to write a test for a web app that accepted JSON from a service, allowed a child to do a test, and then passed the resulting scoring to another service. I did not use OOP, SOLID, or DRY, as I should have according to company policy. But I did get the work done by using KISS and procedural programming with events in a very small time frame. I was penalized for it and forced to rewrite it using their homegrown test player.

So I set about learning their test player. There was no documentation, it did not follow their DRY principles, and very few people if any really understood it. Instead of a few days, like my penalized system, my new version that had to use their system took weeks to build because it did not do what I needed it to do, and I was not allowed to modify it to do what I needed it to do. So I was slowed down while I waited for someone to do what was required.

My first solution satisfied the business requirements and was an independent piece of code that cared about nothing else. The second solution satisfied the development team's technical requirements. The project lasted longer than the deadline. Any project that overshoots its deadline costs the business more money than planned.

The other point I would like to make with my penalized system was that it was far simpler and easier to understand than the newer system that was rewritten to use the generic test player.

You don't always have to follow OOP, SOILD, and DRY. Sometimes it pays not to. After all, you can write the most beautiful OOP system. But under the hood, your code is converted to procedural code that is closer to what the computer understands!

Lack of regions in large classes

Large classes with lots of regions are very hard to read and follow, especially when related methods are not grouped together. Regions are very good for grouping similar to members within a large class. But they are no good if you don't use them!

Lost-intention code

If you are viewing a class and it is doing several things, then how do you know what its original intention was? If you are looking for a date method, for example, and you find it in a file class in the input/output namespace of your code, is the date method in the right location? No. Will it be hard for other developers who don't know your code to find that method? Of course it will. Take a look at this code:

```
public class MyClass
{
    public void MyMethod()
    {
        // ...implementation...
    }

    public DateTime AddDates(DateTime date1, DateTime date2)
```

```
    {  
        //...implementation...  
    }  
  
    public Product GetData(int id)  
    {  
        //...implementation...  
    }  
}
```

What is the purpose of the class? The name does not give any indication, and what does `MyMethod` do? The class also appears to be doing date manipulation and getting product data. The `AddDates` method should be in a class solely for managing dates. And the `GetData` method should be in the product's view model.

Directly exposing information

Classes that directly expose information are bad. Apart from producing tight coupling that can lead to bugs, if you want to change the information type, you have to change the type everywhere it is used. Also, what if you want to perform data validation before the assignment? Here's an example:

```
public class Product  
{  
    public int Id;  
    public int Name;  
    public int Description;  
    public string ProductCode;  
    public decimal Price;  
    public long UnitsInStock  
}
```

In the preceding code, if you wanted to change `UnitsInStock` from type `long` to type `int`, you would have to change the code *everywhere* it is referenced. You would have to do the same with `ProductCode`. If new product codes had to adhere to a strict format, you would not be able to validate product codes if the string could be directly assigned by the calling class.

Good code

Now that you know what not to do, it's time to look briefly at some good coding practices to be able to write pleasing, performant code.

Proper indentation

When you use proper indentation, it makes reading the code much easier. You can tell by the indentation where code blocks start and end, and what code belongs to those code blocks:

```
public void DoSomething()
{
    for (var i = 0; i < 1000; i++)
    {
        var productCode = $"PRC000{i}";
        //...implementation
    }
}
```

In the preceding simple example, the code looks nice and is readable. You can clearly see where each code block starts and finishes.

Meaningful comments

Meaningful comments are comments that express the programmer's intention. Such comments are useful when the code is correct but may not be easily understood by anyone new to the code, or even to the same programmer in a few week's time. Such comments can be really helpful.

API documentation comments

A good API is an API that has good documentation that is easy to follow. API comments are XML comments that can be used to generate HTML documentation. HTML documentation is important for developers wanting to use your API. The better the documentation, the more developers are likely to want to use your API. Here's an example:

```
/// <summary>
/// Create a new <see cref="KustoCode"/> instance from the text and
/// globals. Does not perform
/// semantic analysis.
/// </summary>
/// <param name="text">The code text</param>
/// <param name="globals">
///     The globals to use for parsing and semantic analysis. Defaults to
/// <see cref="GlobalState.Default"/>
/// </param>.
public static KustoCode Parse(string text, GlobalState globals = null) {
... }
```


This excerpt from the Kusto Query Language project is a good example of an API documentation comment.

Proper organization using namespaces

Code that is properly organized and placed in appropriate namespaces can save developers a good amount of time when looking for a particular piece of code. For instance, if you are looking for classes and methods to do with dates and times, it would be a good idea to have a namespace called `DateTime`, a class called `Time` for time-related methods, and a class called `Date` for date-related methods.

The following is an example of the proper organization of namespaces:

Name	Description
<code>CompanyName.IO.FileSystem</code>	The namespace contains classes that define file and directory operations.
<code>CompanyName.Converters</code>	The namespace contains classes for performing various conversion operations.
<code>CompanyName.IO.Streams</code>	The namespace contains types for managing stream input and output.

Good naming conventions

It is good to follow the Microsoft C# naming conventions. Use Pascal casing for namespaces, classes, interfaces, enums, and methods. Use camel case for variable names and argument names, and make sure to prefix member variables with an underscore.

Have a look at this example code:

```
using System;
using System.Text.RegularExpressions;

namespace CompanyName.ProductName.RegEx
{
    /// <summary>
    /// An extension class for providing regular expression extensions
    /// methods.
    /// </summary>
    public static class RegularExpressions
    {
        private static string _preprocessed;

        public static string RegularExpression { get; set; }
        public static bool IsValidEmail(this string email)
```

```

{
    // Email address: RFC 2822 Format.
    // Matches a normal email address. Does not check the
    // top-level domain.
    // Requires the "case insensitive" option to be ON.
    var exp = @"^A(?:[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\.
        [a-z0-9!#$%&'*/+=?^_`{|}~-]+)@(?:[a-z0-9](?:[a-z0-9-]
        [a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?)\Z";
    bool isEmail = Regex.IsMatch(email, exp, RegexOptions.IgnoreCase);
    return isEmail;
}

// ... rest of the implementation ...

}
}

```

It shows suitable examples of naming conventions for namespaces, classes, member variables, classes, parameters, and local variables.

Classes that only do one job

A good class is a class that does only one job. When you read the class, its intention is clear. Only the code that should be in that class is in that class and nothing else.

Methods that do one thing

Methods should only do one thing. You should not have a method that does more than one thing, such as decrypting a string and performing string replacement. A method's intent should be clear. Methods that do only one thing are more inclined to be small, readable, and intentional.

Methods with less than 10 lines, and preferably no more than 4

Ideally, you should have methods that are no longer than 4 lines of code. However, this is not always possible, so you should aim to have methods that are no more than 10 lines in length so that they are easy to read and maintain.

Methods with no more than two parameters

It is best to have methods with no parameters, but having one or two is okay. If you start having more than two parameters, you need to think about the responsibility of your class and methods: are they taking on too much? If you do need more than two parameters, then you are better placed to pass an object.

Any method with more than two parameters can become difficult to read and follow. Having no more than two parameters makes the code readable, and a single parameter that is an object is way more readable than a method with several parameters.

Proper use of exceptions

Never use exceptions to control program flow. Handle common conditions that might trigger exceptions in such a way that an exception will not be raised or thrown. A good class is designed in such a way that you can avoid exceptions.

Recover from exceptions and/or release resources by using `try/catch/finally` exceptions. When catching exceptions, use specific exceptions that may be thrown in your code, so that you have more detailed information to log or assist in handling the exception.

Sometimes, using the predefined .NET exception types is not always possible. In such cases, it will be necessary to produce your own custom exceptions. Suffix your custom exception classes with the word `Exception`, and make sure to include the following three constructors:

- `Exception()`: Uses default values
- `Exception(string)`: Accepts a string message
- `Exception(string, exception)`: Accepts a string message and an inner exception

If you have to throw exceptions, don't return error codes but return exceptions with meaningful information.

Code that is readable

The more readable the code is, the more developers will enjoy working with it. Such code is easier to learn and work with. As developers come and go on a project, newbies will be able to read, extend, and maintain the code with little effort. Readable code is also less inclined to be buggy and unsafe.

Code that is loosely coupled

Loosely coupled code is easier to test and refactor. You can also swap and change loosely coupled code more easily if you need to. Code reuse is another benefit of loosely coupled code.

Let's use our bad example of a database being passed a SQL Server connection. We could make that same class loosely coupled by referencing an interface instead of a concrete type. Let's have a look at a good example of the refactored bad example from earlier:

```
public class Database
{
    private IDatabaseConnection _databaseConnection;

    public Database(IDatabaseConnection databaseConnection)
    {
        _databaseConnection = databaseConnection;
    }
}
```

As you can see in this rather basic example, as long as the passed-in class implements the `IDatabaseConnection` interface, we can pass in any class for any kind of database connection. So if we find a bug in the SQL Server connection class, only SQL Server clients are affected. That means the clients with different databases will continue to work, and we only have to fix the code for SQL Server customers in the one class. This reduces the maintenance overhead and so reduces the overall cost of maintenance.

High cohesion

Common functionality that is correctly grouped together is known to be highly cohesive. Such code is easy to find. For example, if you look at the `Microsoft System.Diagnostics` namespace, you will find that it only contains code that pertains to diagnostics. It would not make sense to include collections and filesystem code in the `Diagnostics` namespace.

Objects are cleanly disposed of

When using disposable classes, you should always call the `Dispose()` method to cleanly dispose of any resources that are in use. This helps to negate the possibility of memory leaks.

There are times when you may need to set an object to `null` for it to go out of scope. An example would be a static variable that holds a reference to an object that you no longer require.

The `using` statement is also a good clean way to use disposable objects, as when the object is no longer in scope it is automatically disposed of, so you don't need to explicitly call the `Dispose()` method. Let's have a look at the code that follows:

```
using (var unitOfWork = new UnitOfWork())
{
    // Perform unit of work here.
}
// At this point the unit of work object has been disposed of.
```

The code defines a disposable object in the `using` statement and does what it needs to between the opening and closing curly braces. The object is automatically disposed of before the braces are exited. And so there is no need to manually call the `Dispose()` method, because it is called automatically.

Avoiding the `Finalize()` method

When using unmanaged resources, it is best to implement the `IDisposable` interface and avoid using the `Finalize()` method. There is no guarantee of when finalizers will run. They may not always run in the order you expect or when you expect them to run. Instead, it is better and more reliable to dispose of unmanaged resources in the `Dispose()` method.

The right level of abstraction

You have the right level of abstraction when you expose to the higher level only that which needs exposure, and you do not get lost in the implementation.

If you find that you are getting lost in the implementation details, then you have over-abstracted. If you find that multiple people have to work in the same class at the same time, then you have under-abstracted. In both cases, refactoring would be needed to get the abstraction to the right level.

Using regions in large classes

Regions are very useful for grouping items within a large class as they can be collapsed. It can be quite daunting reading through a large class and having to jump back and forth between methods, so grouping methods that call each other in the class is a good way to group them. The methods can then be collapsed and expanded as needed when working on a piece of code.

As you can see from what we have looked at so far, good coding practices make for code that is far more readable and easier to maintain. We will now take a look at the need for coding standards and principles along with some software methodologies such as SOLID and DRY.

The need for coding standards, principles, and methodologies

Most software today is written by multiple teams of programmers. As you know, we all have our own unique ways of coding, and we all have some form of programming ideology. You can easily find programming debates regarding various software development paradigms. But the consensus is that it does make our lives easier as programmers if we do all adhere to a given set of coding standards, principles, and methodologies.

Let's review what we mean by these in a little more detail.

Coding standards

Coding standards set out several dos and don'ts that must be adhered to. Such standards can be enforced through tools such as FxCop and manually via peer code reviews. All companies have their own coding standards that must be adhered to. But what you will find in the real world is that when the business expects a deadline to be met, those coding standards can go out of the window as the deadline can become more important than the actual code quality. This is usually rectified by adding any required refactoring to the bug list as technical debt to be addressed after the release.

Microsoft has its own coding standards, and the majority of the time these are the adopted standards that are modified to suit each business' needs. Here are some examples of coding standards found online:

- <https://www.c-sharpcorner.com/UploadFile/ankurmalik123/C-Sharp-coding-standards/>
- <https://www.dofactory.com/reference/csharp-coding-standards>
- <https://blog.submain.com/coding-standards-c-developers-need/>

When people across teams or within the same team adhere to coding standards, your code base becomes unified. A unified code base is much easier to read, extend, and maintain. It is also likely to be less error-prone. And if errors do exist, they are more likely to be found more easily, since the code follows a standard set of guidelines that all developers adhere to.

Coding principles

Coding principles are a set of guidelines for writing high-quality code, testing and debugging that code, and performing maintenance on the code. Principles can be different between programmers and programming teams.

Even if you are a lone programmer, you will do yourself an honorable service by defining your own coding principles and sticking to them. If you work in a team, then it is very beneficial to all to agree on a set of coding standards to make working on shared code easier.

Throughout this book, you will see examples of coding principles such as SOLID, YAGNI, KISS, and DRY, all of which will be explained in detail. But for now, **SOLID** stands for **Single Responsibility Principle, Open-Closed Principle, Liskov Substitution, Interface Segregation Principle, and Dependency Inversion Principle**. **YAGNI** stands for **You Ain't Gonna Need It**. **KISS** stands for **Keep It Simple, Stupid**, and **DRY** stands for **Don't Repeat Yourself**.

Coding methodologies

Coding methodologies break down the process of developing software into a number of predefined phases. Each phase will have a number of steps associated with it. Different developers and development teams will have their own coding methodologies that they follow. The main aim of coding methodologies is to streamline the process from the initial concept, through the coding phase, to the deployment and maintenance phases.

In this book, you will become accustomed to **Test-Driven Development (TDD)** and **Behavioral-Driven Development (BDD)** using SpecFlow, and **Aspect-Oriented Programming (AOP)** using PostSharp.

Coding conventions

It is best to implement the Microsoft C# coding conventions. You can review them at <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions>.

By adopting Microsoft's coding conventions, you are guaranteed to write code in a formally accepted and agreed-upon format. These C# coding conventions help people to focus on reading your code and spend less time focusing on the layout. Basically, Microsoft's coding standards promote best practices.

Modularity

Breaking large programs up into smaller modules makes a lot of sense. Small modules are easy to test, are more readily reused, and can be worked on independently from other modules. Small modules are also easier to extend and maintain.

A modular program can be divided into different assemblies and different namespaces within those assemblies. Modular programs are also much easier to work on in team environments as different modules can be worked on by different teams.

In the same project, code is modularized by adding folders that reflect namespaces. A namespace must only contain code that is related to its name. So, for instance, if you have a namespace called `FileSystem`, then types related to files and directories should be placed in that folder. Likewise, if you have a namespace called `Data`, then only types related to data and data sources should be located in that namespace.

Another beautiful aspect of correct modularization is that if you keep modules small and simple, they are easy to read. Most of a coder's life apart from coding is spent reading and understanding code. So the smaller and more correctly modularized the code is, then the more easier it is to read and understand the code. This leads to a greater understanding of the code and improves developer take-up and use of the code.

KISS

You may be the super genius of the computer programming world. You may be able to produce code that is so sexy that other programmers can only stare at it in awe and end up drooling on their keyboard. But do those other programmers know what the code is by just looking at it? If you found that code in 10 weeks' time when you head deep into a mountain of different code with deadlines to meet, would you be able to explain with absolute clarity what your code does and the rationale behind your choice of coding method? And have you considered that you may have to work on that code further down the road?

Have you ever programmed some code, gone away, and then looked at it more than a few days later and thought to yourself, *I didn't write this rubbish, did I? What was I thinking!?* I know I've been guilty of it and so have some of my ex-colleagues.

When programming code, it is essential to keep the code simple and in a human-readable format that even newbie junior programmers can understand. Often juniors are exposed to code to read, understand, and then maintain. The more complex the code, the longer it takes for juniors to get up to speed. Even seniors can struggle with complex systems to the point that they leave to find work elsewhere that's less taxing on the brain and their well-being.

For example, if you are working on a simple website, ask yourself a few questions. Does it really need to use microservices? Is the brownfield project you are working on really complicated? Is it possible to simplify it to make it easier to maintain? When developing a new system, what are the minimum number of moving parts you need to write a robust, maintainable, and scalable solution that performs well?

YAGNI

YAGNI is a discipline in the agile world of programming that stipulates that a programmer should not add any code until it is absolutely needed. An honest programmer will write failing tests based on a design, then write just enough production code for the tests to work, and finally, refactor the code to remove any duplication. Using the YAGNI software development methodology, you keep your classes, methods, and overall lines of code to an absolute minimum.

The primary goal of YAGNI is to prevent the over-engineering of software systems by computer programmers. Do not add complexity if it is not needed. You must remember to only write the code that you need. Don't write code that you don't need, and don't write code for the sake of experimentation and learning. Keep experimental and learning code in sandboxed projects specifically for those purposes.

DRY

I said *Don't Repeat Yourself!* If you find that you are writing the same code in multiple areas, then this is a definite candidate for refactoring. You should look at the code to see if it can be genericized and placed in a helper class for use throughout the system or in a library for use by other projects.

If you have the same piece of code in multiple locations, and you find the code has a fault and needs to be modified, you must then modify the code in other areas. In situations like this, it is very easy to overlook code that requires modification. The result is code that gets released with the problem fixed in some areas, but still existing in others.

That is why it is a good idea to remove duplicate code as soon as you encounter it, as it may cause more problems further down the road if you don't.

SOLID

SOLID is a set of five design principles that intend to make software easier to understand and maintain. Software code should be easy to read and extend without having to modify portions of the existing code. The five SOLID design principles are as follows:

- **Single Responsibility Principle:** Classes and methods should only perform a single responsibility. All the elements that form a single responsibility should be grouped together and encapsulated.
- **Open/Closed Principle:** Classes and methods should be open for extension and closed for modification. When a change to the software is required, you should be able to extend the software without modifying any of the code.
- **Liskov Substitution:** Your function has a pointer to a base class. It must be able to use any class derived from the base class without knowing it.
- **Interface Segregation Principle:** When you have large interfaces, the clients that use them may not need all the methods. So, using the **Interface Segregation Principle (ISP)**, you extract out methods to different interfaces. This means that instead of having one big interface, you have many small interfaces. Classes can then implement interfaces with only the methods they need.
- **Dependency Inversion Principle:** When you have a high-level module, it should not be dependent upon any low-level modules. You should be able to switch between low-level modules freely without affecting the high-level module that uses them. Both high-level and low-level modules should depend upon abstractions.

An abstraction should not depend upon details, but details should depend upon abstractions.

When you declare variables, you should always use static types such as an interface or abstract class. Concrete classes that implement the interface or inherit from the abstract class can then be assigned to the variable.

Occam's Razor

Occam's Razor states the following: *Entities should not be multiplied without necessity.* To paraphrase, this essentially means that *the simplest solution is most likely the correct one.* So, in software development, the breaking of the principle of Occam's Razor is accomplished by making unnecessary assumptions and employing the least simple solution to a software problem.

Software projects are usually founded upon a collection of facts and assumptions. Facts are easy to deal with but assumptions are something else. When coming up with a software project solution to a problem, you normally discuss the problem and potential solutions as a team. When choosing a solution, you should always choose the project with the least assumptions as this will be the most accurate choice to implement. If there are a few fair assumptions, the more assumptions you are having to make, the more likely it is that your design solution is flawed.

A project with less moving parts has less that can go wrong with it. So, by keeping projects small with as few entities as possible by not making assumptions unless they are necessary, and only dealing with facts, you adhere to the principle of Occam's Razor.

Summary

In this chapter, you have had an introduction to good code and bad code and, hopefully, you now understand why good code matters. You have also been provided with the link to the Microsoft C# coding conventions so that you can follow Microsoft best practices for coding (if you are not already doing so).

You have also briefly been introduced to various software methodologies including DRY, KISS, SOLID, YAGNI, and Occam's Razor.

Using modularity, you have seen the benefits of modularizing code using namespaces and assemblies. Such benefits include independent teams being able to work on independent modules, and code reusability and maintainability.

In the next chapter, we will be looking at peer code reviews. They can be unpleasant at times, but peer code reviews help to keep programmers in check by making sure they are adhering to the company coding procedure.

Questions

1. What are some of the outcomes of bad code?
2. What are some of the outcomes of good code?
3. What are some of the benefits of writing modular code?
4. What is DRY code?
5. Why should you KISS when writing code?
6. What does the acronym SOLID stand for?
7. Explain YAGNI.
8. What's Occam's Razor?

Further reading

- *Adaptive Code: Agile coding with design patterns and SOLID principles, Second Edition* by Gary McLean Hall.
- *Hands-On Design Patterns with C# and .NET Core* by Jeffrey Chilberto and Gaurav Arora.
- *Building Maintainable Software, C# Edition* by Rob van der Leek, Pascal van Eck, Gijs Wijnholds, Sylvain Rigal, and Joost Visser.
- Good information on software anti-patterns, including a long list of anti-patterns, can be found at https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Architecture/Anti-Patterns.
- Good information on design patterns, with a list of design patterns that links to diagrams and implementation source code, can be found at https://en.wikipedia.org/wiki/Software_design_pattern.

2

Code Review – Process and Importance

The primary motivation behind any code review is to improve the overall quality of the code. Code quality is very important. This almost goes without saying, especially if your code is part of a team project or is accessible to others, such as open source developers and customers through escrow agreements.

If every developer was free to code as they pleased, you would end up with the same kind of code written in so many different ways, and ultimately the code would become an unwieldy mess. That is why it is important to have a coding standards policy that outlines the company's coding practices and code review procedures that are to be followed.

When code reviews are carried out, colleagues will review the code of other colleagues. Colleagues will understand that it is only human to make mistakes. They will check the code for mistakes, coding that breaks the company's code of coding conduct, and any code that, while syntactically correct, can be improved upon to either make it more readable, more maintainable, or more performant.

Therefore, in this chapter, we will cover the following topics to understand the code review process in detail:

- Preparing code for review
- Leading a code review
- Knowing what to review
- Knowing when to send code for review
- Providing and responding to review feedback



Please note that for the *Preparing code for review* and *Knowing when to send code for review* sections, we will be talking from the point of view of the **programmer**. For the *Leading a code review* and *Knowing what to review* sections, we will be talking from the point of view of the **code reviewer**. However, as regards the *Providing and responding to review feedback* section, we will cover the viewpoints of both the **programmer** and the **code reviewer**.

The learning objectives for this chapter are for you to be able to do the following:

- Understand code reviews and why they are good
- Partake in code reviews
- Provide constructive criticism
- Respond positively to constructive criticism

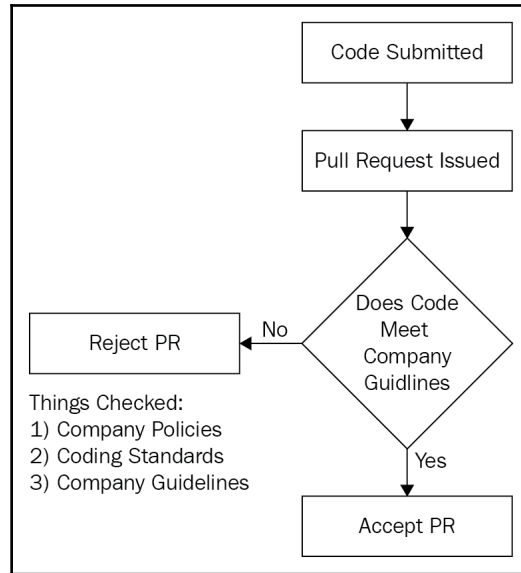
Before we dive deep into these topics, let's understand the general code review process.

The code review process

The normal procedure for carrying out a code review is to make sure your code compiles and meets the requirements set. It should also pass all unit tests and end-to-end tests. Once you are confident that you are able to compile, test, and run your code successfully, then it is checked in to the current working branch. Once checked in, you will then issue a pull request.

A peer reviewer will then review your code and share comments and feedback. If your code passes the code review, your code review is completed and you can then merge your working branch into the main trunk. Otherwise, the peer review will be rejected, and you will be required to review your work and address the issues raised in the comments provided by your reviewer.

The following diagram shows the peer code review process:



Preparing code for review

Preparing for a code review can be a right royal pain at times, but it does work for better overall quality of code that is easy to read and maintain. It is definitely a worthwhile practice that teams of developers should carry out as standard coding procedures. This is an important step in the code review process, as perfecting this step can save the reviewer considerable time and energy in performing the review.

Here are some standard points to keep in mind when preparing your code for review:

- **Always keep the code review in mind:** When beginning any programming, you should have the code review in mind. So keep your code small. If possible, limit your code to one feature.
- **Make sure that all your tests pass even if your code builds:** If your code builds but you have failing tests, then deal immediately with what's causing those tests to fail. Then, when the tests pass as expected, you can move on. It is important to make sure that all unit tests are passed, and that end-to-end testing passes all tests. It is important that all testing is complete and gets the green light, since releasing code that works but was a test fail could result in some very unhappy customers when the code goes to production.

- **Remember YAGNI:** As you code, make sure to only add code that is necessary to meet the requirement or feature you are working on. If you don't need it yet, then don't code it. Only add code when it is needed and not before.
- **Check for duplicate code:** If your code must be object-oriented and be DRY and SOLID, then review your own code to see whether it contains any procedural or duplicate code. Should it do so, take the time to refactor it so that it is object-oriented, DRY, and SOLID.
- **Use static analyzers:** Static code analyzers that have been configured to enforce your company's best practices will check your code and highlight any issues that are encountered. Make sure that you do not ignore information and warnings. These could cause you issues further down the line.



Most importantly, only check your code in when you are confident that your code satisfies business requirements, adheres to coding standards, and passes all tests. If you check your code in as part of a **Continuous Integration (CI)** pipeline, and your code fails the build, then you will need to address the areas of concern raised by the CI pipeline. When you are able to check in your code and the CI gives the green light, then you can issue a pull request.

Leading a code review

When leading code reviews, it is important to have the right people present. The people who will be in attendance at the peer code review will be agreed upon with the project manager. The programmer(s) responsible for submitting the code for review will be present at the code review unless they work remotely. In the case of remote working, the reviewer will review the code and either accept the pull request, decline the pull request, or send the developer some questions to be answered before taking any further action.

A suitable lead for a code review should possess the following skills and knowledge:

- **Be a technical authority:** The person leading the code review should be a technical authority that understands the company's coding guidelines and software development methodologies. It is also important that they have a good overall understanding of the software under review.
- **Have good soft skills:** As the leader of the code review, the person must be a warm and encouraging individual who is able to provide constructive feedback. It is important that the person reviewing the programmer's code has good soft skills so that there is no conflict between the reviewer and the person whose code is being reviewed.
- **Not be overly critical:** The leader of the code review must not be over-critical and must be able to explain their critique of the programmer's code. It is useful if the leader has been exposed to different programming styles, and can view the code objectively to ensure that it meets the project's requirements.

In my experience, peer code reviews are always carried out on pull requests in the version control tool being used by the team. A programmer will submit the code to version control and then issue a pull request. The peer code reviewer will then review the code in the pull request. Constructive feedback will be provided in the form of comments that will be attached to the pull request. If there are problems with the pull request, then the reviewer will reject the change request and comment on specific issues that need to be addressed by the programmer. If the code review is successful, then the reviewer may add a comment providing positive feedback, merge the pull request, and close it.

Programmers will need to note any comments made by the reviewer and take them on board. If the code needs to be resubmitted, then the programmer will need to ensure that all the reviewer's comments have been addressed prior to resubmitting.

It is a good idea to keep code reviews short, and to not review too many lines at any one time.

Since a code review normally starts with a pull request, we will look at issuing a pull request followed by responding to a pull request.

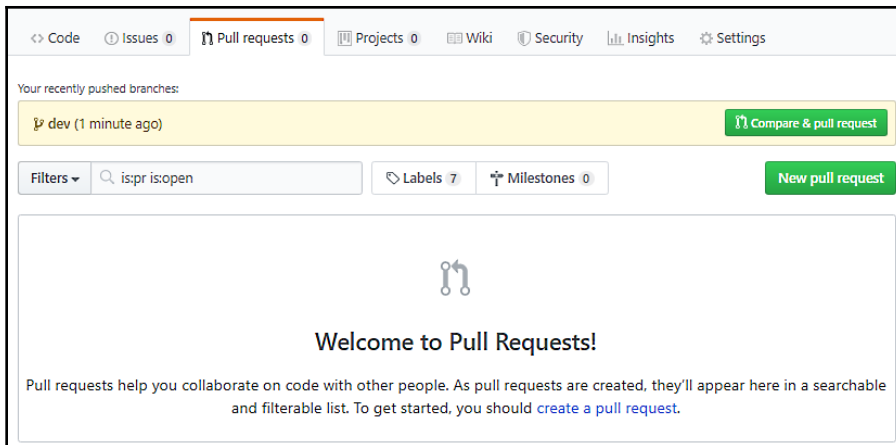
Issuing a pull request

When you have finished coding and you are confident in the quality of your code and that it builds, you are able to then push or check in your changes, depending on what source control system you use. When your code has been pushed, you can then issue a pull request. When a pull request is raised, other people that are interested in the code are notified and able to review your changes. These changes can then be discussed and comments made regarding any potential changes that you need to make. In essence, your pushing to your source control repository and issuing a pull request is what kick-starts the peer code review process.

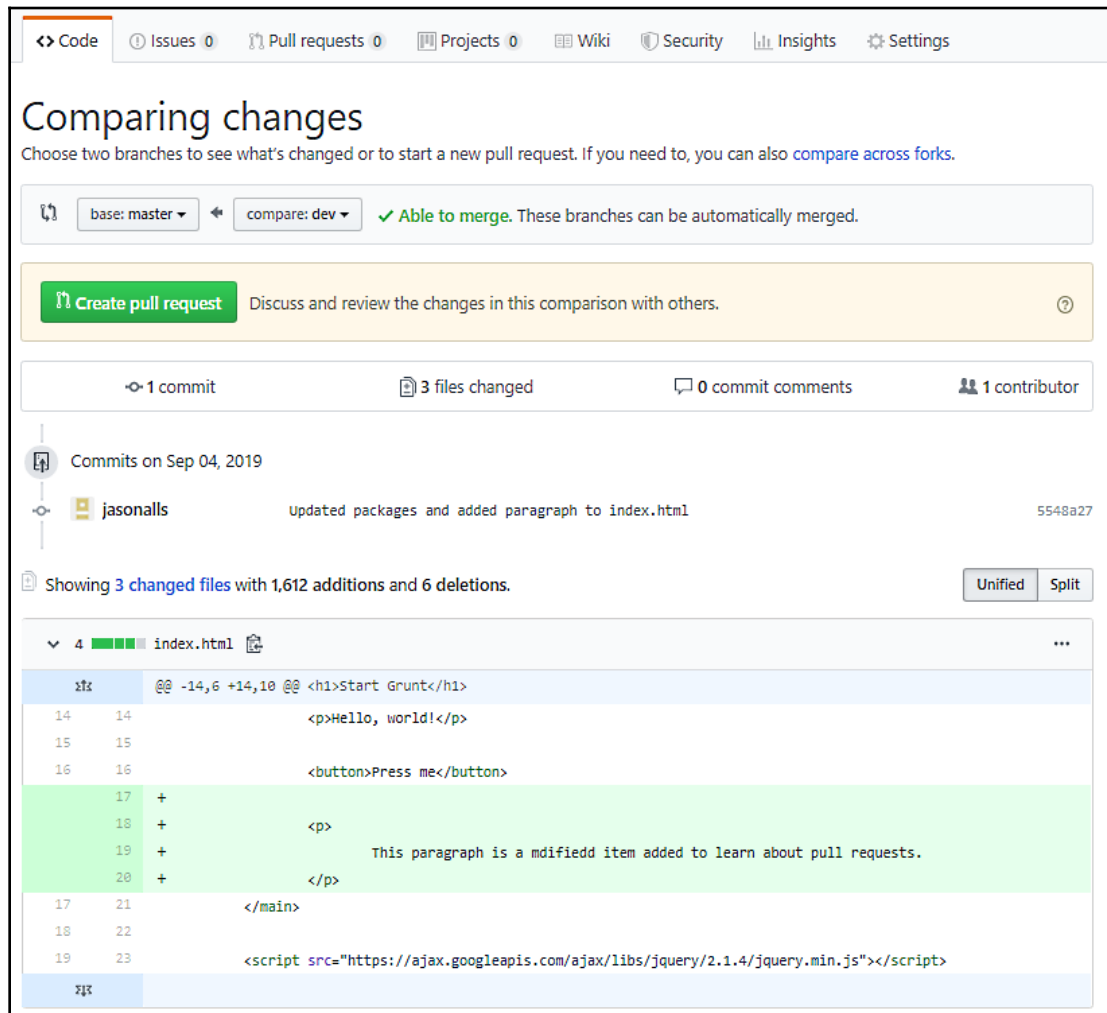
To issue a pull request, all you have to do (once you've checked your code in or pushed it) is click on the **Pull requests** tab of your version control. There will then be a button you can click on – **New pull request**. This will add your pull request to a queue to be picked up by the relevant reviewers.

In the following screenshots, we will see the process of requesting and fulfilling a pull request via GitHub:

1. On your GitHub project page, click on the **Pull requests** tab:



2. Then, click on the **New pull request** button. This will display the **Comparing changes** page:



3. If you are happy, then click on the **Create pull request** button to start the pull request. You will then be presented with the **Open a pull request** screen:

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base: master ← compare: dev ✓ **Able to merge.** These branches can be automatically merged.

Updated packages and added paragraph to index.html

Write Preview

Updated packages to the latest versions, and added a paragraph to index.html

Attach files by dragging & dropping, selecting or pasting them.

Create pull request

1 commit 3 files changed 0 commit comments 1 contributor

Commits on Sep 04, 2019

jasonalls Updated packages and added paragraph to index.html 5548a27

Showing 3 changed files with 1,612 additions and 6 deletions.

Unified Split

4 index.html

```

@@ -14,6 +14,10 @@ <h1>Start Grunt</h1>
14 14         <p>Hello, world!</p>
15 15
16 16         <button>Press me</button>
17 +
18 +         <p>
19 +             This paragraph is a mdifiedd item added to learn about pull requests.
20 +         </p>
17 21     </main>

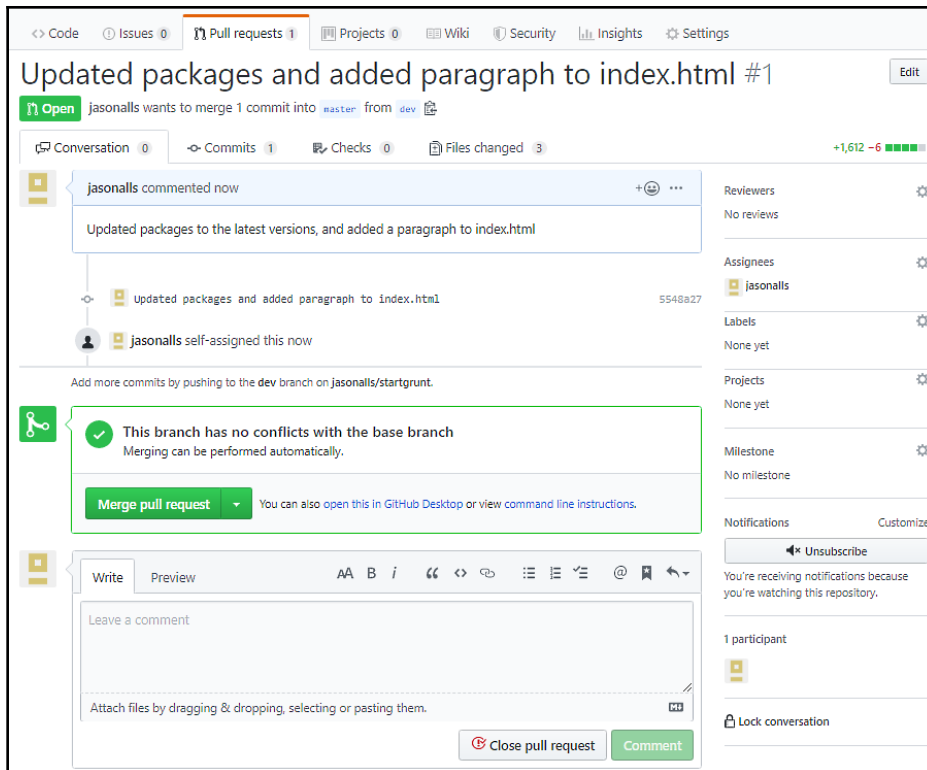
```

- Write your comment regarding the pull request. Provide all the necessary information for the code reviewer, but keep it brief and to the point. Useful comments include identification of what changes have been made. Modify the **Reviewers**, **Assignees**, **Labels**, **Projects**, and **Milestones** fields as necessary. Then, once you are happy with the pull request details, click on the **Create pull request** button to create the pull request. Your code will now be ready to be reviewed by your peers.

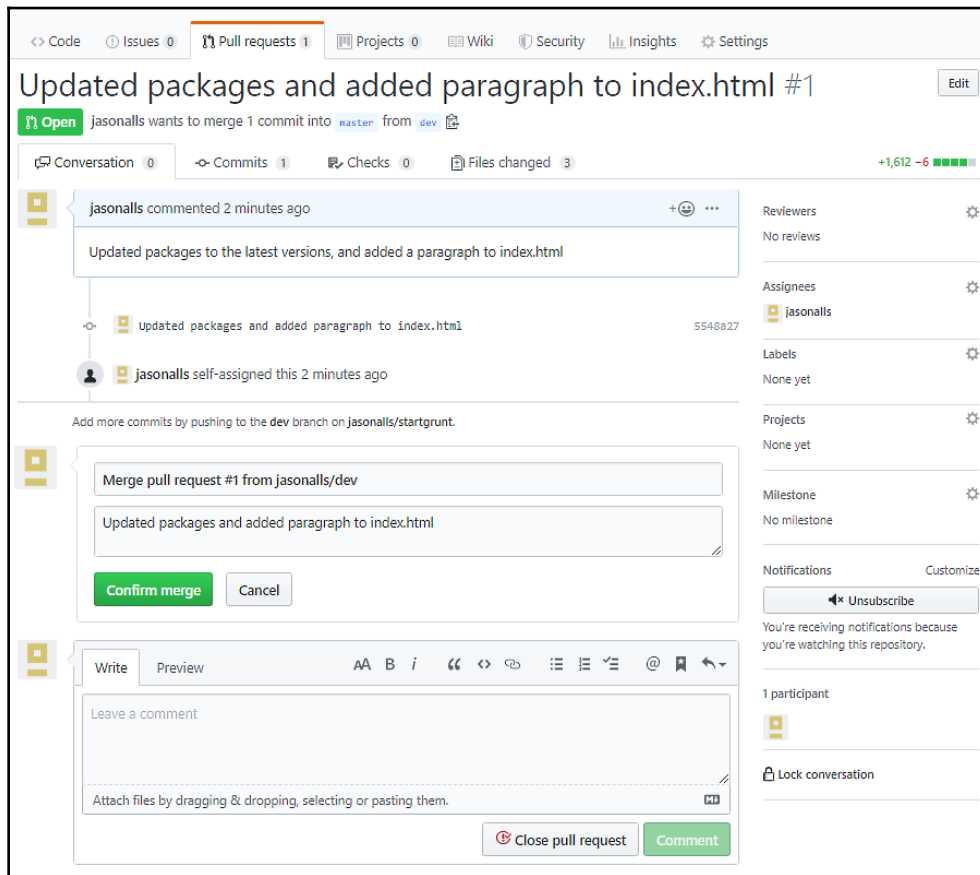
Responding to a pull request

Since the reviewer is responsible for reviewing pull requests prior to merges of branches, we would do well to look at responding to pull requests:

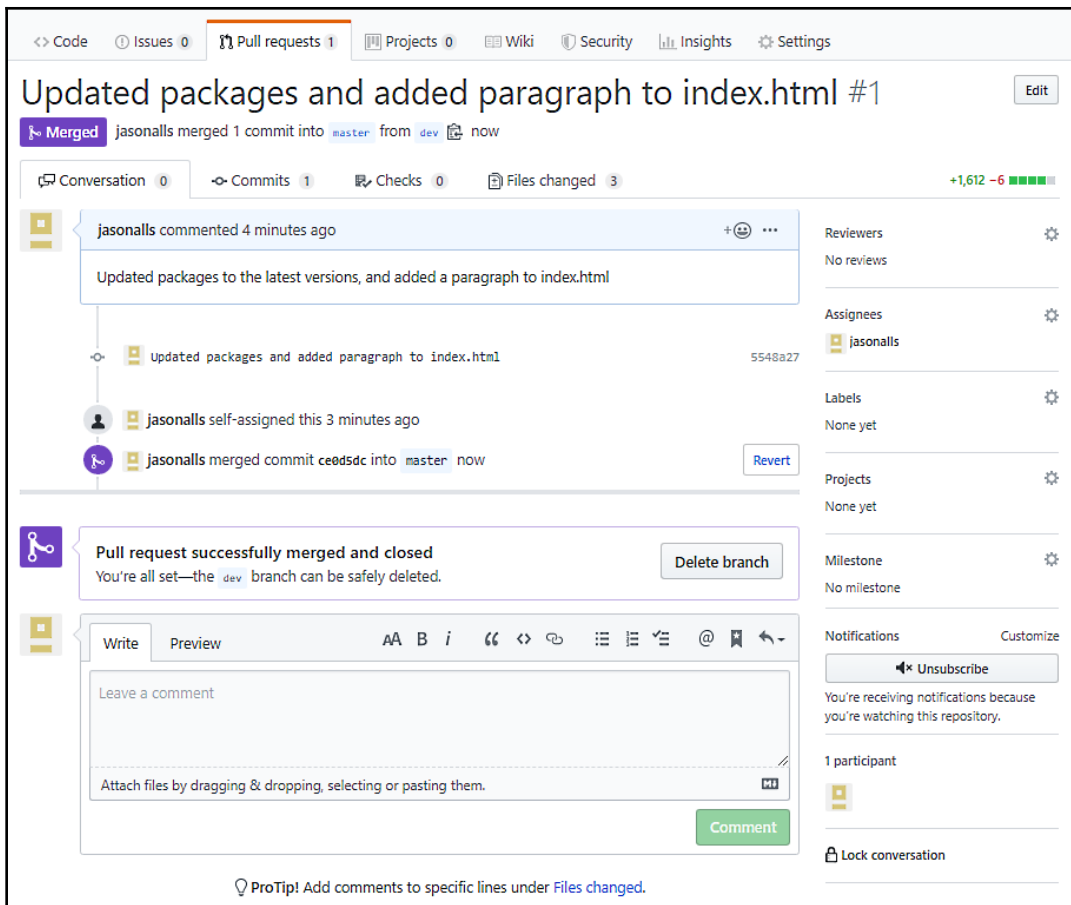
1. Start by cloning a copy of the code under review.
2. Review the comments and changes in the pull request.
3. Check that there are no conflicts with the base branch. If there are, then you will have to reject the pull request with the necessary comments. Otherwise, you can review the changes, make sure the code builds without errors, and make sure there are no compilation warnings. At this stage, you will also look out for code smells and any potential bugs. You will also check that the tests build, run, are correct, and provide good test coverage of the feature to be merged. Make any comments necessary and reject the pull request unless you are satisfied. When satisfied, you can add your comments and merge the pull request by clicking on the **Merge pull request** button, as shown here:



- Now, confirm the merge by entering a comment and clicking on the **Confirm merge** button:



- Once the pull request has been merged and the pull request closed, the branch can be deleted by clicking on the **Delete branch** button, as can be seen in the following screenshot:



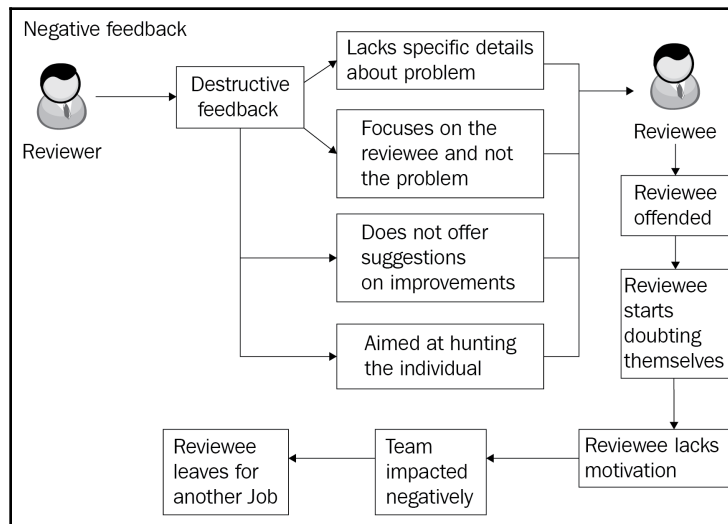
In the previous section, you saw how the reviewee raises a pull request to have their code peer-reviewed before it is merged. And in this section, you have seen how to review a pull request and complete it as part of a code review. Now, we will look at what to review in a peer code review when responding to a pull request.

Effects of feedback on reviewees

When performing a code review of your peer's code, you must also take into consideration the fact that feedback can be positive or negative. Negative feedback does not provide specific details about the problem. The reviewer focuses on the reviewee and not on the problem. Suggestions for improving the code are not offered to the reviewee by the reviewer, and the reviewer's feedback is aimed at hurting the reviewee.

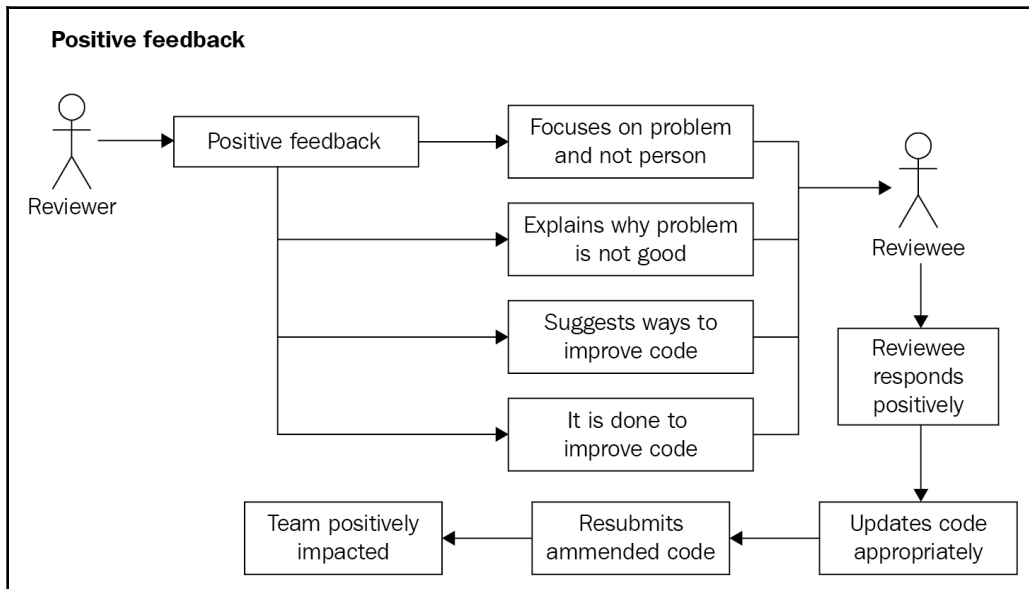
Such negative feedback received by the reviewee offends them. This has a negative impact and can cause them to start doubting themselves. A lack of motivation then develops within the reviewee and this can negatively impact the team, as work is not done on time or to the required level. The bad feelings between the reviewer and the reviewee will also be felt by the team, and an oppressive atmosphere that negatively impacts everyone on the team can ensue. This can lead to other colleagues becoming demotivated, and the overall project can end up suffering as a result.

In the end, it gets to the point where the reviewee has had enough and leaves for a new position somewhere else to get away from it all. The project then suffers time-wise and even financially, as time and money will need to be spent on finding a replacement. Whoever is found to fill the position then has to be trained upon the system and the working procedures and guidelines. The following diagram shows negative feedback from the reviewer toward the reviewee:



Conversely, positive feedback from the reviewer to the reviewee has the opposite effect. When the reviewer provides positive feedback to the reviewee, they focus on the problem and not on the person. They explain why the code submitted is not good, along with the problems it can cause. The reviewer will then suggest to the reviewee ways in which the code can be improved. The feedback provided by the reviewer is only done to improve the quality of the code submitted by the reviewee.

When the reviewee receives the positive (constructive) feedback, they respond in a positive manner. They take on board the reviewer's comments and respond in the appropriate manner by answering any questions, asking any relevant questions themselves, and the code is then updated, based on the reviewer's feedback. The amended code is then resubmitted for review and acceptance. This has a positive impact on the team as the atmosphere remains a positive one, and work is done on time and to the required quality. The following diagram shows the results of positive feedback on the reviewee from the reviewer:



The point to remember is that your feedback can be constructive or destructive. Your aim as a reviewer is to be constructive and not destructive. A happy team is a productive team. A demoralized team is not productive and is damaging to the project. So, always strive to maintain a happy team through positive feedback.

A technique for positive criticism is the feedback sandwich technique. You start with praise on the good points, then you provide constructive criticism, and then you finish with further praise. This technique can be very useful if you have members on the team that doesn't react well to any form of criticism. Your soft skills in dealing with people are just as important as your software skills in delivering quality code. Don't forget that!

We will now move on to look at what we should review.

Knowing what to review

There are different aspects of the code that have to be considered when reviewing it. Primarily, the code being reviewed should only be the code that was modified by the programmer and submitted for review. That's why you should aim to make small submissions often. Small amounts of code are much easier to review and comment on.

Let's go through different aspects a code reviewer should assess for a complete and thorough review.

Company's coding guidelines and business requirement(s)

All code being reviewed should be checked against the company's coding guidelines and the business requirement(s) the code is addressing. All new code should adhere to the latest coding standards and best practices employed by the company.

There are different types of business requirements. These requirements include those of the business and the user/stakeholder as well as functional and implementation requirements. Regardless of the type of requirement the code is addressing, it must be fully checked for correctness in meeting requirements.

For example, if the user/stakeholder requirement states that *as a user, I want to add a new customer account*, does the code under review meet all the conditions set out in this requirement? If the company's coding guidelines stipulate that all code must include unit tests that test the normal flow and exceptional cases, then have all the required tests been implemented? If the answer to any of these questions is *no*, then the code must be commented on, the comments addressed by the programmer, and the code resubmitted.

Naming conventions

The code should be checked to see whether the naming conventions have been followed for the various code constructs, such as classes, interfaces, member variables, local variables, enumerations, and methods. Nobody likes cryptic names that are hard to decipher, especially if the code base is large.

Here are a couple of questions that a reviewer should ask:

- Are the names long enough to be human-readable and understandable?
- Are they meaningful in relation to the intent of the code, but short enough to not irritate other programmers?

As the reviewer, you must be able to read the code and understand it. If the code is difficult to read and understand, then it really needs to be refactored before being merged.

Formatting

Formatting goes a long way to making code easy to understand. Namespaces, braces, and indentation should be employed according to the guidelines, and the start and end of code blocks should be easily identifiable.

Again, here is a set of questions a reviewer should consider asking in their review:

- Is code to be indented using spaces or tabs?
- Has the correct amount of white space been employed?
- Are there any lines of code that are too long that should be spread over multiple lines?
- What about line breaks?
- Following the style guidelines, is there only one statement per line? Is there only one declaration per line?
- Are continuation lines correctly indented using one tab stop?
- Are methods separated by one line?
- Are multiple clauses that make up a single expression separated by parentheses?
- Are classes and methods clean and small, and do they only do the work they are meant to do?

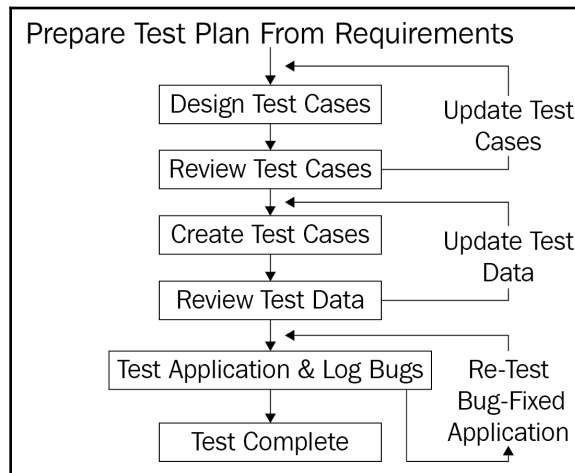
Testing

Tests must be understandable and cover a good subset of use cases. They must cover the normal paths of execution and exceptional use cases. When it comes to testing the code, the reviewer should check for the following:

- Has the programmer provided tests for all the code?
- Is there any code that is untested?
- Do all the tests work?

- Do any of the tests fail?
- Is there adequate documentation of the code, including comments, documentation comments, tests, and product documentation?
- Do you see anything that stands out that, even if it compiles and works in isolation, could cause bugs when integrated into the system?
- Is the code well documented to aid maintenance and support?

Let's see how the process goes:



Untested code has the potential to raise unexpected exceptions during testing and production. But just as bad as code that is not tested are tests that are not correct. This can lead to bugs that are hard to diagnose, can be annoying for the customer, and make more work for you further down the line. Bugs are technical debt and looked upon negatively by the business. Moreover, you may have written the code, but others may have to read it as they maintain and extend the project. It is always a good idea to provide some documentation for your colleagues.

Now, concerning the customer, how are they going to know where your features are and how to use them? Good documentation that is user-friendly is a good idea. And remember, not all your users may be technically savvy. So, cater to the less technical person that may need handholding, but do it without being patronizing.

As a technical authority reviewing the code, do you detect any code smells that may become a problem? If so, then you must flag, comment, and reject the pull request and get the programmer to resubmit their work.

As a reviewer, you should check that those exceptions are not used to control the program flow and that any errors raised have meaningful messages that are helpful to developers and to the customers who will receive them.

Architectural guidelines and design patterns

The new code must be checked to see whether it conforms to the architectural guidelines for the project. The code should follow any coding paradigms that the company employs, such as SOLID, DRY, YAGNI, and OOP. In addition, where possible, the code should employ suitable design patterns.

This is where the **Gang-of-Four (GoF)** patterns come into play. The GOF comprises four authors of a C++ book called *Design Patterns: Elements of Reusable Object-Oriented Software*. The authors were Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Today, their design patterns are heavily used in most, if not all, object-oriented programming languages. Packt has books that cover design patterns, including *.NET Design Patterns*, by Praseen Pai and Shine Xavier. Here is a really good resource that I recommend that you visit: <https://www.dofactory.com/net/design-patterns>. The site covers each of the GoF patterns and provides the definition, UML class diagram, participants, structural code, and some real-world code for the patterns.

GoF patterns consist of creational, structural, and behavioral design patterns. Creational design patterns include Abstract Factory, Builder, Factory Method, Prototype, and Singleton. Structural design patterns include Adapter, Bridge, Composite, Decorator, Façade, Flyweight, and Proxy. Behavioral design patterns include Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor.

The code should also be properly organized and placed in the correct namespace and module. Check the code also to see whether it is too simplistic or over-engineered.

Performance and security

Other things that may need to be considered include performance and security:

- How well does the code perform?
- Are there any bottlenecks that need to be addressed?

- Is the code programmed in such a way to protect against SQL injection attacks and denial-of-service attacks?
- Is code properly validated to keep the data clean so that only valid data gets stored in the database?
- Have you checked the user interface, documentation, and error messages for spelling mistakes?
- Have you encountered any magic numbers or hard coded values?
- Is the configuration data correct?
- Have any secrets accidentally been checked in?

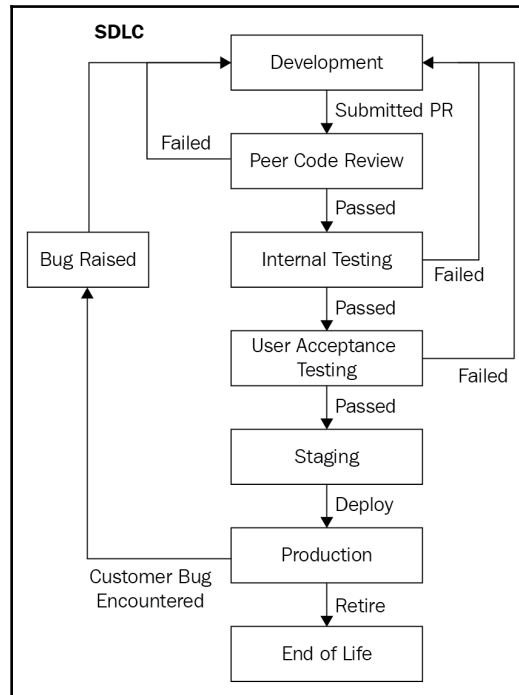
A comprehensive code review will encompass all of the preceding aspects and their respective review parameters. But let's find out when it is actually the right time to even be performing a code review.

Knowing when to send code for review

Code reviews should take place when the development is complete and before the programmer of the code passes the code on to the QA department. Before any code is checked into version control, all the code should build and run without errors, warnings, or information. You can ensure this by doing the following:

- You should run static code analysis on your programs to see whether any issues are raised. If you receive any errors, warnings, or information, then address each point raised. Do not ignore them as they can cause problems further down the line. You can access the **Code Analysis** configuration dialog on the **Code Analysis** page of the Visual Studio 2019 **Project Properties** tab. Right-click on your project and select **Properties | Code Analysis**.
- You should also make sure that all your tests run successfully, and you should aim to have all your new code to be fully covered by normal and exceptional use cases that test the correctness of your code against the specification you are working on.
- If you employ a continuous development software practice within your place of work that integrates your code into a larger system, then you need to make sure that the system integration is successful and that all tests run without failing. If any errors are encountered, then you must fix them before you go any further.

When your code is complete, fully documented, and your tests work, and your system integration all works without any issues, then that is the best time to undergo a peer code review. Once you have reached the point that your peer code review is approved, your code can then be passed on to the QA department. The following diagram shows the **Software Development Life Cycle (SDLC)** from the development of the code through to the end of the life of the code:



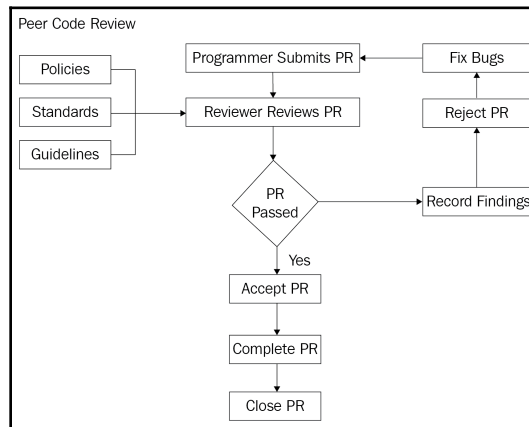
The programmer codes the software as per specifications. They submit the source code to the version control repository and issue a pull request. The request is reviewed. If the request fails, then the request is rejected with comments. If the code review passes, then the code is deployed to the QA team that carry out their own internal testing. Any bugs found are raised for the developers to fix. If the internal testing passes QA, then it is deployed into **User Acceptance Testing (UAT)**.

If UAT fails, then bugs are raised with the DevOps team, who could be developers or infrastructure. If UAT passes QA, then it is deployed to staging. Staging is the team responsible for deploying the product in the production environment. When the software is in the hands of the customer, they raise a bug report if they encounter any bugs. Developers then work on fixing the customer's bugs, and the process is restarted. Once the product reaches the end of its life, it is retired from service.

Providing and responding to review feedback

It is worth remembering that code reviews are aimed at the overall quality of code in keeping with the company's guidelines. Feedback, therefore, should be constructive and not used as an excuse to put down or embarrass a fellow colleague. Similarly, reviewer feedback should not be taken personally and responses to the reviewer should focus on suitable action and explanation.

The following diagram shows the process of issuing a **Pull Request (PR)**, performing a code review, and either accepting or rejecting the PR:



Providing feedback as a reviewer

Workplace bullying can be a problem, and programming environments are not immune. Nobody likes a cocky programmer who thinks they are big. So, it is important that the reviewer has good soft skills and is very diplomatic. Bear in mind that some people can easily be offended and take things the wrong way. So know who you are dealing with and how they are likely to respond; this will help you choose your approach and your words carefully.

As the peer code reviewer, you will be responsible for understanding the requirements and making sure the code meets that requirement. So, look for the answers to these questions:

- Are you able to read and understand the code?
- Can you see any potential bugs?

- Have any trade-offs been made?
- If so, why were the trade-offs made?
- Do the trade-offs incur any technical debt that will need to be factored into the project further down the line?

Once your review is complete, you will have three categories of feedback to choose from: positive, optional, and critical. With **positive feedback**, you can provide commendations on what the programmer has done really well. This is a good way to bolster morale as it can often run low in programming teams. **Optional feedback** can be very useful in helping computer programmers to hone their programming skills in line with the company guidelines, and they can work to improve the overall wellbeing of the software being developed.

Finally, we have critical feedback. **Critical feedback** is necessary for any problems that have been identified and must be addressed before the code can be accepted and passed on to the QA department. This is the feedback where you will need to choose your words carefully to avoid offending anyone. It is important that your critical comments address the specific issue being raised with valid reasons to support the feedback.

Responding to feedback as a reviewee

As the reviewee programmer, you must effectively communicate the background of your code to your reviewer. You can help them by making small commits. Small amounts of code are much easier to review than large amounts of code. The more code being reviewed, the easier it is for things to be missed and slip through the net. While you are waiting for your code to be reviewed, you must not make any further changes to it.

As you can guess, you will receive either positive, optional, or critical feedback from the reviewer. The positive feedback works to boost your confidence in the project as well as your morale. Build upon it and continue with your good practices. You may choose to act or not upon optional feedback, but it's always a good idea to talk it through with your reviewer.

For critical feedback, you must take it seriously and act upon it as this feedback is imperative for the very success of the project. It is very important that you handle critical feedback in a polite and professional manner. Don't allow yourself to be offended by any comments from your reviewer; they are not meant to be personal. This is especially important for new programmers, and programmers who lack confidence.

As soon as you receive your reviewer's feedback, act upon it, and make sure that you discuss it with them as necessary.

Summary

In this chapter, we have discussed the importance of performing code reviews and the complete process of getting code ready for review and responding to reviewer comments as the programmer, along with how to lead a code review and what to look for when performing a review as the code reviewer. It can be seen that there are clearly two roles in a peer code review. These are the reviewer and the reviewee. The reviewer is the person performing the code review, and the reviewee is the person whose code is being reviewed.

You have also seen how you, as a reviewer, can categorize your feedback and why soft skills are important when providing feedback to fellow programmers. And as a reviewee whose code is being scrutinized, you have seen how important it is to build upon positive and optional feedback and how important it is to act upon critical feedback.

By now, you should have a good understanding of why it is important to conduct regular code reviews, and why they should be done before the code is passed on to the QA department. Peer code reviews do take time and can be uncomfortable for both the reviewer and reviewee. But in the long run, they work toward a high-quality product that is easy to extend and maintain, and they lead to better code reuse as well.

In the next chapter, we will be looking at how to write clean classes, objects, and data structures. You will see how we can organize our classes, ensure our classes only have one responsibility, and comment on our classes in order to assist with documentation generation. We will then look at cohesion and coupling, designing for change, and the Law of Demeter. Then, we will look at immutable objects and data structures, hiding data, and exposing methods in objects, before finally looking at data structures.

Questions

1. What are the two roles involved in a peer code review?
2. Who agrees on the people that will be involved in the peer code review?
3. How can you save your reviewer time and effort prior to requesting a peer code review?
4. When reviewing code, what kinds of things must you look out for?
5. What are the three categories of feedback?

Further reading

- <https://docs.microsoft.com/en-us/visualstudio/code-quality/?view=vs-2019>: This documentation by Microsoft provides information on the different tools available to help you analyze and improve the quality and maintainability of your code.
- https://en.wikipedia.org/wiki/Code_review: There are many useful links on this page to further your knowledge of code reviews and their value to your business.
- <https://springframework.guru/gang-of-four-design-patterns/>: Gang-of-Four design patterns book.
- <https://www.packtpub.com/application-development/net-design-patterns>: *.NET Design Patterns*, by Praseed Pai and Shine Xavier.
- <https://help.github.com/en>: GitHub's help page.

3

Classes, Objects, and Data Structures

In this chapter, we will look at organizing, formatting, and commenting on classes. We will also look at writing clean C# objects and data structures that respect the Law of Demeter. In addition, we will look at immutable objects and data structures and the interfaces and classes that define immutable collections in the `System.Collections.Immutable` namespace.

We will cover the following broad topics:

- Organizing classes
- Commenting for document generation
- Cohesion and coupling
- The Law of Demeter
- Immutable objects and data structures

As you progress through this chapter, you will learn the following skills:

- How to effectively organize your classes using namespaces.
- Your classes will become smaller and more meaningful as you learn to program them with only a single responsibility.
- When it comes to writing your own APIs, you will be able to provide good developer documentation by providing comments that aid document generation tools.
- Any programs you write will be easy to modify and extend due to their high cohesion and low coupling.
- Finally, you will be able to apply the Law of Demeter and write and use immutable data structures.

So, let's start by looking at how we can effectively organize our classes by using namespaces.

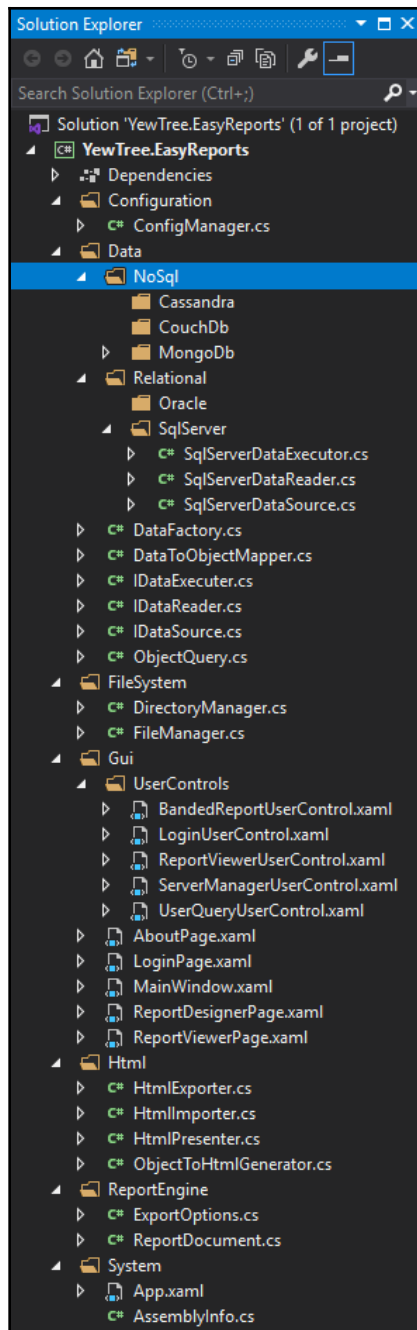
Technical requirements

You can access the code for this chapter on GitHub, at <https://github.com/PacktPublishing/Clean-Code-in-C-/tree/master/CH03>.

Organizing classes

You will notice that the hallmark of a clean project is that it will have well-organized classes. And folders will be used to group classes that belong together. Further, the classes in the folders will be enclosed within namespaces that match the assembly name and folder structure.

Each interface, class, struct, and enum should have its own source file in the correct namespace. Source files should be logically grouped together in the appropriate folders and the namespaces for the source files should match the assembly name and folder structure. The following screenshot demonstrates a clean folder and file structure:





It is a bad idea to have more than one interface, class, struct, or enum in an actual source file. The reason for this is that it can make locating items difficult, despite the fact that we have IntelliSense to assist us.

When thinking about your namespaces, it is a good idea to follow the Pascal casing sequence of company name, product name, technology name, and then plural names for components separated by spaces. See the following for an example:

```
FakeCompany.Product.Wpf.Feature.Subnamespace {} // Product, technology and  
feature specific.
```

The reason for starting with the company name is that it helps to avoid namespace classes. So, if Microsoft and FakeCompany both have a namespace called `System`, which `System` you desire to use can be differentiated by the company name.

Next, any items of code that are able to be reused in multiple projects are best placed in separate assemblies that can be accessed by multiple projects:

```
FakeCompany.Wpf.Feature.Subnamespace {} /* Technology and feature specific.  
Can be used across multiple products. */
```

When using tests in your code, such as when doing **Test-Driven Development (TDD)**, it is always best to keep your test classes in separate assemblies. Test assemblies should always be given the name of the assembly they are testing with the namespace `Tests` appended to the end of the assembly name:

```
FakeCompany.Core.Feature {} /* Technology agnostic and feature specific.  
Can be used across multiple products. */
```

You should never put tests for different assemblies in the same test assembly as each other. Always keep them separate.

In addition, the namespace and type should not use the same name as this can produce compiler conflicts. When pluralizing namespaces, you can forego pluralizing for company names, product names, and acronyms.

To summarize, here are the rules to keep in mind when organizing classes:

- Follow the Pascal casing sequence of company name, product name, technology name, and plural names for components separated by spaces.
- Place reusable items of code in separate assemblies.
- Don't use the same name for the namespace and type.
- Don't pluralize company and product names and acronyms.

We'll move on to the responsibility of classes.

A class should have only one responsibility

Responsibility is the work that has been assigned to the class. In the SOLID set of principles, the S stands for **Single Responsibility Principle (SRP)**. When applied to a class, SRP states that the class must only work on a single aspect of the feature being implemented. The responsibility of that single aspect should be fully encapsulated within the class. Therefore, you should never apply more than one responsibility to a class.

Let's look at an example to understand why:

```
public class MultipleResponsibilities()
{
    public string DecryptString(string text,
        SecurityAlgorithm algorithm)
    {
        // ...implementation...
    }

    public string EncryptString(string text,
        SecurityAlgorithm algorithm)
    {
        // ...implementation...
    }

    public string ReadTextFromFile(string filename)
    {
        // ...implementation...
    }

    public string SaveTextToFile(string text, string filename)
    {
        // ...implementation...
    }
}
```

As you can see in the preceding code, for the `MultipleResponsibilities` class, we have our cryptography functionalities implemented with the `DecryptString` and the `EncryptString` methods. We also have file access implemented with the `ReadTextFromFile` and `SaveTextToFile` methods. This class breaks the SRP principle.

So we need to break this class up into two classes, one for cryptography and the other for file access:

```
namespace FakeCompany.Core.Security
{
    public class Cryptography
    {
        public string DecryptString(string text,
            SecurityAlgorithm algorithm)
        {
            // ...implementation...
        }

        public string EncryptString(string text,
            SecurityAlgorithm algorithm)
        {
            // ...implementation...
        }
    }
}
```

As we can now see from the preceding code, by moving the `EncryptString` and `DecryptString` methods to their own `Cryptography` class in the core security namespace, we have made it easy to reuse the code to encrypt and decrypt strings across different products and technology groups. The `Cryptography` class also complies with SRP.

In the following code, we can see that the `SecurityAlgorithm` parameter of the `Cryptography` class is an enum and has been placed in its own source file. This helps to keep code clean, minimal, and well organized:

```
using System;

namespace FakeCompany.Core.Security
{
    [Flags]
    public enum SecurityAlgorithm
    {
        Aes,
        AesCng,
        MD5,
        SHA5
    }
}
```

Now, in the following `TextFile` class, we again abide by SRP and have a nice reusable class that is in the appropriate core filesystem namespace. The `TextFile` class is reusable across different products and technology groups:

```
namespace FakeCompany.Core.FileSystem
{
    public class TextFile
    {
        public string ReadTextFromFile(string filename)
        {
            // ...implementation...
        }

        public string SaveTextToFile(string text, string filename)
        {
            // ...implementation...
        }
    }
}
```

We've looked at the organization and the responsibility of classes. Now let's take a look at commenting on classes for the benefit of other developers.

Commenting for documentation generation

Documenting your source code is always a good idea, whether it is an internal project or external software that will be used by other developers. Internal projects suffer because of developer turnover and often poor, or little to no documentation available to help new developers get up to speed. Many third-party APIs fail to get off the ground or uptake is slower than expected, often with adopters abandoning the APIs through frustration because of the poor state of the developer documentation.

It is always a good idea to include copyright notices at the top of each source code file and to comment on your namespaces, interfaces, classes, enums, structs, methods, and properties. Your copyright comments should be first in the source file, above the `using` statements and take the form of a multiline comment that starts with `/*` and ends with `*/`:

```
/*
*****
 * Copyright 2019 PacktPub
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
copy of
 * this software and associated documentation files (the "Software"), to
```

```

deal in
* the Software without restriction, including without limitation the
rights to use,
* copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the
* Software, and to permit persons to whom the Software is furnished to do
so,
* subject to the following conditions:
*
* The above copyright notice and this permission notice shall be included
in all
* copies or substantial portions of the Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN THE
* SOFTWARE.
*****
*****/

using System;

/// <summary>
/// The CH3.Core.Security namespace contains fundamental types used
/// for the purpose of implementing application security.
/// </summary>
namespace CH3.Core.Security
{
    /// <summary>
    /// Encrypts and decrypts provided strings based on the selected
    /// algorithm.
    /// </summary>
    public class Cryptography
    {
        /// <summary>
        /// Decrypts a string using the selected algorithm.
        /// </summary>
        /// <param name="text">The string to be decrypted.</param>
        /// <param name="algorithm">
        /// The cryptographic algorithm used to decrypt the string.
        /// </param>

```

```
    /// <returns>Decrypted string</returns>
    public string DecryptString(string text,
        SecurityAlgorithm algorithm)
    {
        // ...implementation...
        throw new NotImplementedException();
    }

    /// <summary>
    /// Encrypts a string using the selected algorithm.
    /// </summary>
    /// <param name="text">The string to encrypt.</param>
    /// <param name="algorithm">
    /// The cryptographic algorithm used to encrypt the string.
    /// </param>
    /// <returns>Encrypted string</returns>
    public string EncryptString(string text,
        SecurityAlgorithm algorithm)
    {
        // ...implementation...
        throw new NotImplementedException();
    }
}
```

The preceding code sample provides an example of a documented namespace and class with documented methods. You will see that the documentation comments for the namespace and contained members start with the documentation comment `///` and are directly above the item being commented on. When you type the three forward slashes, Visual Studio automatically generates the XML tags based on the line below.

For example, in the preceding code, the namespace only has a summary and so does the class, but both methods contain a summary, a couple of parameter comments, and a return comment.

The following table contains the different XML tags that you can use in your documentation comments.

Tag	Section	Purpose
<c>	<c>	Formats text as code
<code>	<code>	Provides source code as output
<example>	<example>	Provides an example
<exception>	<exception>	Describes the exceptions that can be thrown by the method
<include>	<include>	Includes XML from an external file
<list>	<list>	Adds a list or table

<para>	<para>	Adds structure to text
<param>	<param>	Describes the parameter of a constructor or method
<paramref>	<paramref>	Tags a word to identify it is a parameter
<permission>	<permission>	Describes the security accessibility of the member
<remarks>	<remarks>	Provides additional information
<returns>	<returns>	Describes the return type
<see>	<see>	Adds a hyperlink
<seealso>	<seealso>	Adds a <i>see also</i> entry
<summary>	<summary>	Summarizes the type or member
<value>	<value>	Describes the value
<typeparam>		Describes the type parameter
<typeparamref>		Tags a word to identify it as a type parameter

From the preceding table, it is clear that you have plenty of scope for documenting your source code. So it is a good idea to make the best use of the available tags to document your code. The better the documentation, the quicker and easier it will be for other developers to get up to speed with using the code.

It is now time to look at cohesion and coupling.

Cohesion and coupling

In a well-designed C# assembly, code will be correctly grouped together. This is known as **high cohesion**. **Low cohesion** is when you have code grouped together that does not belong together.

You want related classes to be as independent as possible. The more dependent one class is on another class, the higher the coupling. This is known as **tight coupling**. The more independent classes are of one another, the lower the cohesion. This is known as low cohesion.

So, in a well-defined class, you want high cohesion and low coupling. We'll now look at examples of tight coupling followed by low coupling.

An example of tight coupling

In the following code example, the `TightCouplingA` class breaks encapsulation and makes the `_name` variable directly accessible. The `_name` variable should be private and modified only by the properties of methods within its enclosing class. The `Name` property provides get and set methods to validate the `_name` variable, but this is pretty pointless as those checks can be bypassed and the properties not called:

```
using System.Diagnostics;

namespace CH3.Coupling
{
    public class TightCouplingA
    {
        public string _name;

        public string Name
        {
            get
            {
                if (!_name.Equals(string.Empty))
                    return _name;
                else
                    return "String is empty!";
            }
            set
            {
                if (value.Equals(string.Empty))
                    Debug.WriteLine("String is empty!");
            }
        }
    }
}
```

On the other hand, in the following code, the `TightCouplingB` class creates an instance of `TightCouplingA`. It then introduces tight coupling between the two classes by directly accessing the `_name` member variable and setting it to `null`, and then directly accessing to print its value to the debug output window:

```
using System.Diagnostics;

namespace CH3.Coupling
{
    public class TightCouplingB
    {
        public TightCouplingB()
        {

```

```
        {
            TightCouplingA tca = new TightCouplingA();
            tca._name = null;
            Debug.WriteLine("Name is " + tca._name);
        }
    }
}
```

Now let's look at the same simple example using a low coupling.

An example of low coupling

In this example, we have two classes, `LooseCouplingA` and `LooseCouplingB`. `LooseCouplingA` declares a private instance variable named `_name`, and this variable is set via a public property.

`LooseCouplingB` creates an instance of `LooseCouplingA` and gets and sets the value of `Name`. Because the `_name` data member cannot be set directly, the checks on setting and getting the value of that data member are performed.

And so we have an example of loose coupling. Let's have a look at the two classes called `LooseCouplingA` and `LooseCouplingB` that show this in action:

```
using System.Diagnostics;

namespace CH3.Coupling
{
    public class LooseCouplingA
    {
        private string _name;
        private readonly string _stringIsEmpty = "String is empty";

        public string Name
        {
            get
            {
                if (_name.Equals(string.Empty))
                    return _stringIsEmpty;
                else
                    return _name;
            }

            set
            {
                if (value.Equals(string.Empty))
```

```
        Debug.WriteLine("Exception: String length must be  
            greater than zero.");  
    }  
}  
}
```

In the `LooseCouplingA` class, we declare the `_name` field as private and so prevent the data from being directly modified. The `_name` data is made indirectly accessible by the `Name` property:

```
using System.Diagnostics;  
  
namespace CH3.Coupling  
{  
    public class LooseCouplingB  
    {  
        public LooseCouplingB()  
        {  
            LooseCouplingA lca = new LooseCouplingA();  
            lca = null;  
            Debug.WriteLine($"Name is {lca.Name}");  
        }  
    }  
}
```

The `LooseCouplingB` class is unable to directly access the `_name` variable of the `LooseCouplingB` class, and so modifies the data member via a property.

Well, we've looked at coupling and now know how to avoid tightly coupled code and implement loosely coupled code. So now, it is time for us to look at some examples of low cohesion and high cohesion.

An example of low cohesion

When a class has more than one responsibility, it is said to be a low cohesive class. Have a look at the following code:

```
namespace CH3.Cohesion  
{  
    public class LowCohesion  
    {  
        public void ConnectToDatasource() { }  
        public void ExtractDataFromDataSource() { }  
        public void TransformDataForReport() { }  
    }  
}
```



```
        public void AssignDataAndGenerateReport() { }
        public void PrintReport() { }
        public void CloseConnectionToDataSource() { }
    }
}
```

As we can see, the preceding class has at least three responsibilities:

- Connecting to and disconnecting from a data source
- Extracting data and transforming it ready for report insertion
- Generating a report and printing it out

You will see clearly how this breaks the SRP. Next, we will break this class down into three classes that adhere to the SRP.

An example of high cohesion

In this example, we are going to break down the `LowCohesion` class into three classes that obey the SRP. These will be called `Connection`, `DataProcessor`, and `ReportGenerator`. Let's see how much cleaner the code is after we implement the three classes.

In the following class, you can see that the only methods in that class are related to connecting to a data source:

```
namespace CH3.Cohesion
{
    public class Connection
    {
        public void ConnectToDatasource() { }
        public void CloseConnectionToDataSource() { }
    }
}
```

The class itself is named `Connection`, so this is an example of a high cohesive class.

In the following code, the `DataProcessor` class contains two methods that process data by extracting data from the data source and transforming that data for insertion into the report:

```
namespace CH3.Cohesion
{
    public class DataProcessor
    {
        public void ExtractDataFromDataSource() { }
        public void TransformDataForReport() { }
    }
}
```

```
    }  
}
```

So this is another example of a highly cohesive class.

In the following code, the `ReportGenerator` class only has methods associated with generating and outputting the report:

```
namespace CH3.Cohesion  
{  
    public class ReportGenerator  
    {  
        public void AssignDataAndGenerateReport() { }  
        public void PrintReport() { }  
    }  
}
```

Again, this is another example of a highly cohesive class.

Looking at each of the three classes, we can see that they contain only methods that pertain to their single responsibility. And so each of the three preceding classes is highly cohesive.

It is now time to look at how we design our code for change by using interfaces in place of classes so that code can be injected into constructors and methods using dependency injection and inversion of control.

Design for change

When designing for change, you should change the *what* to the *how*.

The *what* is the requirement of the business. As any seasoned person involved in a role within software development will tell you that requirements frequently change. As such, the software has to be adaptable to meet those changes. The business is not interested in *how* the requirements are implemented by the software and infrastructure teams, only that the requirements are met precisely on time and on budget.

On the other hand, the software and infrastructure teams are more focused on *how* those business requirements are to be met. Regardless of the technology and processes that are adopted for the project to implement the requirements, the software and target environment must be adaptable to changing requirements.

But that is not all. You see, software versions often change with bug fixes and new features. As new features are implemented and refactoring takes place, the software code becomes deprecated and eventually obsolete. On top of that, software vendors have a road map of their software that forms part of their application life cycle management. Eventually, software versions get to the point where they are retired and no longer supported by the vendor. This can force a major migration from the current version, which will no longer be supported, to the new supported version, and this can bring with it breaking changes that must be addressed.

Interface-oriented programming

Interface-Oriented Programming (IOP) helps us to program polymorphic code. Polymorphism in OOP is defined as different classes having their own implementations of the same interface. And so, by using interfaces, we can morph our software to meet the needs of the business.

Let's consider a database connection example. An application may be required to connect to different data sources. But how can the database code remain the same no matter what database is employed? Well, the answer lies in the use of interfaces.

You have different database connection classes that implement the same database connection interface, but they each have their own versions of the implemented methods. This is known as polymorphism. The database then accepts a database connection parameter that is of the database connection interface type. You can then pass into the database any database connection type that implements the database connection interface. Let's code this example so that it makes things a little more clear.

Start by creating a simple .NET Framework console application. Then update the `Program` class as follows:

```
static void Main(string[] args)
{
    var program = new Program();
    program.InterfaceOrientedProgrammingExample();
}

private void InterfaceOrientedProgrammingExample()
{
    var mongoDb = new MongoDBConnection();
    var sqlServer = new SqlServerConnection();
    var db = new Database(mongoDb);
    db.OpenConnection();
    db.CloseConnection();
}
```

```
        db = new Database(sqlServer);  
        db.OpenConnection();  
        db.CloseConnection();  
    }
```

In this code, the `Main()` method creates a new instance of the `Program` class and then calls the `InterfaceOrientedProgrammingExample()` method. In that method, we instantiate two different database connections, one for MongoDB and one for SQL Server. We then instantiate the database with a MongoDB connection, open the database connection, and then close it. Then we instantiate a new database using the same variable and pass in a SQL Server connection, then open the connection and close the connection. As you can see, we only have one `Database` class with a single constructor, yet the `Database` class will work with any database connection that implements the required interface. So, let's add the `IConnection` interface:

```
public interface IConnection  
{  
    void Open();  
    void Close();  
}
```

The interface has only two methods called `Open()` and `Close()`. Add the MongoDB class that will implement this interface:

```
public class MongoDBConnection : IConnection  
{  
    public void Close()  
    {  
        Console.WriteLine("Closed MongoDB connection.");  
    }  
  
    public void Open()  
    {  
        Console.WriteLine("Opened MongoDB connection.");  
    }  
}
```

We can see that the class implements the `IConnection` interface. Each method prints out a message to the console. Now add that `SqlServerConnection` class:

```
public class SqlServerConnection : IConnection  
{  
    public void Close()  
    {  
        Console.WriteLine("Closed SQL Server Connection.");  
    }  
}
```

```
        public void Open()
        {
            Console.WriteLine("Opened SQL Server Connection.");
        }
    }
```

The same goes for the `Database` class. It implements the `IConnection` interface, and for each method invocation, a message is printed to the console. And now for the `Database` class, as follows:

```
public class Database
{
    private readonly IConnection _connection;

    public Database(IConnection connection)
    {
        _connection = connection;
    }

    public void OpenConnection()
    {
        _connection.Open();
    }

    public void CloseConnection()
    {
        _connection.Close();
    }
}
```

The `Database` class accepts an `IConnection` parameter. This sets the `_connection` member variable. The `OpenConnection()` method opens the database connection, and the `CloseConnection()` method closes the database connection. Well, it's time to run the program. You should see the following output in the console window:

```
Opened MongoDB connection.
Closed MongoDB connection.
Opened SQL Server Connection.
Closed SQL Server Connection.
```

So now, you can see the advantage of programming to interfaces. You can see how they enable us to extend the program without having to modify the existing code. That means that if we need to support more databases, then all we have to do is write more connection objects that implement the `IConnection` interface.

Now that you know how interfaces work, we can look at how to apply them to dependency injection and inversion of control. Dependency injection helps us to write clean code that is loosely coupled and easy to test, and inversion of control enables the interchanging of software implementations as required, as long as those implementations implement the same interface.

Dependency injection and inversion of control

In C#, we have the ability to address changing software needs using **Dependency Injection (DI)** and **Inversion of Control (IoC)**. These two terms do have different meanings but are often used interchangeably to mean the same thing.

With IoC, you program a framework that accomplishes tasks by calling modules. An IoC container is used to keep a register of modules. These modules are loaded when requested by the user or configuration requests them.

DI removes internal dependencies from classes. Dependent objects are then injected by an external caller. An IoC container uses DI to inject dependent objects into an object or method.

In this chapter, you will find some useful resources that will help you to understand IoC and DI. You will then be able to use these techniques in your programs.

Let's see how we can implement our own simple DI and IoC without any third-party frameworks.

An example of DI

In this example, we are going to roll our own simple DI. We will have an `ILogger` interface that will have a single method with a string parameter. We will then produce a class called `TextFileLogger` that implements the `ILogger` interface and outputs a string to a text file. Finally, we will have a `Worker` class that will demonstrate constructor injection and method injection. Let's look at the code.

The following interface has a single method that will be used for implementing classes to output a message according to the implementation of the method:

```
namespace CH3.DependencyInjection
{
    public interface ILogger
    {
        void OutputMessage(string message);
    }
}
```

```
    }  
}
```

The `TextFileLogger` class implements the `ILogger` interface and outputs the message to a text file:

```
using System;  
  
namespace CH3.DependencyInjection  
{  
    public class TextFileLogger : ILogger  
    {  
        public void OutputMessage(string message)  
        {  
            System.IO.File.WriteAllText(FileName(), message);  
        }  
  
        private string FileName()  
        {  
            var timestamp = DateTime.Now.ToFileTimeUtc().ToString();  
            var path = Environment.GetFolderPath(Environment  
                .SpecialFolder.MyDocuments);  
            return $"{path}_{timestamp}";  
        }  
    }  
}
```

The `Worker` class provides an example of constructor DI and method DI. Notice that the parameter is an interface. So, any class that implements that interface can be injected at runtime:

```
namespace CH3.DependencyInjection  
{  
    public class Worker  
    {  
        private ILogger _logger;  
  
        public Worker(ILogger logger)  
        {  
            _logger = logger;  
            _logger.OutputMessage("This constructor has been injected  
                with a logger!");  
        }  
  
        public void DoSomeWork(ILogger logger)  
        {  
            logger.OutputMessage("This methods has been injected  
                with a logger!");  
        }  
    }  
}
```

```
    }  
  }  
}
```

The `DependencyInject` method runs the example to show DI in action:

```
private void DependencyInject()  
{  
    var logger = new TextFileLogger();  
    var di = new Worker(logger);  
    di.DoSomeWork(logger);  
}
```

As you can see with the code we've just looked at, we start by producing a new instance of the `TextFileLogger` class. This object is then injected into the constructor of the worker. We then call the `DoSomeWork` method and pass in the `TextFileLogger` instance. In this simple example, we have seen how to inject code into a class via its constructor and via methods.

What is good about this code is it removes the dependency between the worker and the `TextFileLogger` instance. This makes it easy for us to replace the `TextFileLogger` instance with any other type of logger that implements the `ILogger` interface. So we could have used, for example, an event viewer logger or even a database logger. Using DI is a good way to reduce coupling in your code.

Now that we've seen DI at work, we should also look at IoC. And we'll do that now.

An example of IoC

In this example, we are going to register dependencies with an IoC container. We will then use DI to inject the necessary dependencies.

In the following code, we have an IoC container. The container registers the dependencies to be injected in a dictionary, and reads values from the configuration metadata:

```
using System;  
using System.Collections.Generic;  
  
namespace CH3.InversionOfControl  
{  
    public class Container  
    {  
        public delegate object Creator(Container container);  
  
        private readonly Dictionary<string, object> configuration = new
```



```
        Dictionary<string, object>();  
private readonly Dictionary<Type, Creator> typeToCreator = new  
    Dictionary<Type, Creator>();  
  
public Dictionary<string, object> Configuration  
{  
    get { return configuration; }  
}  
  
public void Register<T>(Creator creator)  
{  
    typeToCreator.Add(typeof(T), creator);  
}  
  
public T Create<T>()  
{  
    return (T)typeToCreator[typeof(T)](this);  
}  
  
public T GetConfiguration<T>(string name)  
{  
    return (T)configuration[name];  
}  
}  
}
```

Then, we create a container, and we use the container to configure metadata, register types, and create instances of dependencies:

```
private void InversionOfControl()  
{  
    Container container = new Container();  
    container.Configuration["message"] = "Hello World!";  
    container.Register<ILogger>(delegate  
    {  
        return new TextFileLogger();  
    });  
    container.Register<Worker>(delegate  
    {  
        return new Worker(container.Create<ILogger>());  
    });  
}
```

Next up, we will look at how to limit an object's knowledge to knowing only about its close relatives using the Law of Demeter. This will help us to write a clean C# code that avoids the use of navigation trains.

The Law of Demeter

The Law of Demeter aims to remove navigation trains (dot counting), and it also aims to provide good encapsulation with loosely coupled code.

A method that understands a navigation train breaks the Law of Demeter. For example, have a look at the following code:

```
report.Database.Connection.Open(); // Breaks the Law of Demeter.
```

Each unit of code should have a limited amount of knowledge. That knowledge should only be of relevant code that is closely related. With the Law of Demeter, you must tell and not ask. Using this law, you may only call methods of objects that are one or more of the following:

- Passed as arguments
- Created locally
- Instance variables
- Globals

Implementing the Law of Demeter can be difficult, but there are advantages to telling rather than asking. One such benefit is the decoupling of your code.

It is good to see a bad example that breaks the Law of Demeter, along with one that obeys the Law of Demeter, so we will see this in the following sections.

A good and a bad example (chaining) of the Law of Demeter

In the good example, we have the report instance variable. On the report variable object instance, the method to open the connection is called. This does not break the law.

The following code is a `Connection` class with a method that opens a connection:

```
namespace CH3.LawOfDemeter
{
    public class Connection
    {
        public void Open()
        {
            // ... implementation ...
        }
    }
}
```

```
    }  
}
```

The Database class creates a new Connection object and opens a connection:

```
namespace CH3.LawOfDemeter  
{  
    public class Database  
    {  
        public Database()  
        {  
            Connection = new Connection();  
        }  
  
        public Connection Connection { get; set; }  
  
        public void OpenConnection()  
        {  
            Connection.Open();  
        }  
    }  
}
```

In the Report class, a Database object is instantiated and then a connection to the database is opened:

```
namespace CH3.LawOfDemeter  
{  
    public class Report  
    {  
        public Report()  
        {  
            Database = new Database();  
        }  
  
        public Database Database { get; set; }  
  
        public void OpenConnection()  
        {  
            Database.OpenConnection();  
        }  
    }  
}
```

So far, we have seen good code that obeys the Law of Demeter. But the following is code that breaks this law.

In the `Example` class, the Law of Demeter is broken because we introduce method chaining, as in `report.Database.Connection.Open()`:

```
namespace CH3.LawOfDemeter
{
    public class Example
    {
        public void BadExample_Chaining()
        {
            var report = new Report();
            report.Database.Connection.Open();
        }

        public void GoodExample()
        {
            var report = new Report();
            report.OpenConnection();
        }
    }
}
```

In this bad example, the `Database` getter is called on the `report` instance variable. This is acceptable. But then a call is made to the `Connection` getter that returns a different object. This breaks the Law of Demeter, as does the final call to open the connection.

Immutable objects and data structures

Immutable types are normally thought of as just value types. With value types, it makes sense that when they are set, you don't want them to change. But you can also have immutable object types and immutable data structure types. Immutable types are a type whose internal state does not change once they have been initialized.

The behavior of immutable types does not astonish or surprise fellow programmers and so conforms to the **principle of least astonishment (POLA)**. The POLA conformity of immutable types adheres to any contracts made between clients, and because it is predictable, programmers will find it easy to reason about its behavior.

Since immutable types are predictable and do not change, you are not going to be in for any nasty surprises. So you don't have to worry about any undesirable effects due to them being altered in some way. This makes immutable types ideal for sharing between threads as they are thread-safe and there is no need for defensive programming.

When you create an immutable type and use object validation, you have a valid object for the lifetime of that object.

Let's have a look at an example of an immutable type in C#.

An example of an immutable type

We are now going to look at an immutable object. The `Person` object in the following code has three private member variables. The only time these can be set is during the creation time in the constructor. Once set, they are unable to be modified for the rest of the object's lifetime. Each variable is only readable via read-only properties:

```
namespace CH3.ImmutableObjectsAndDataStructures
{
    public class Person
    {
        private readonly int _id;
        private readonly string _firstName;
        private readonly string _lastName;

        public int Id => _id;
        public string FirstName => _firstName;
        public string LastName => _lastName;
        public string FullName => $"{_firstName} {_lastName}";
        public string FullNameReversed => $"{_lastName}, {_firstName}";

        public Person(int id, string firstName, string lastName)
        {
            _id = id;
            _firstName = firstName;
            _lastName = lastName;
        }
    }
}
```

Now we have seen how easy it is to write immutable objects and data structures, we will look at data and methods in objects.

Objects should hide data and expose methods

The state of your object is stored in member variables. These member variables are data. Data should not be directly accessible. You should only provide access to data via exposed methods and properties.

Why should you hide your data and expose your methods?

Hiding data and exposing methods is known in the OOP world as encapsulation. Encapsulation hides the inner workings of a class from the outside world. This makes it easy to be able to change value types without breaking existing implementations that rely on the class. Data can be made read/writable, writable, or read-only providing more flexibility to you regarding data access and usage. You can also validate input and so prevent data from receiving invalid values. Encapsulating also makes testing your classes much easier, and you can make your classes more reusable and extendable.

Let's look at an example.

An example of encapsulation

The following code example shows an encapsulated class. The `Car` object is mutable. It has properties that get and set the data values once they have been initialized by the constructor. The constructor and the set properties perform the validation of the parameter arguments. If the value is invalid, an invalid argument exception is thrown, otherwise the value is passed back and the data value is set:

```
using System;

namespace CH3.Encapsulation
{
    public class Car
    {
        private string _make;
        private string _model;
        private int _year;

        public Car(string make, string model, int year)
        {
            _make = ValidateMake(make);
            _model = ValidateModel(model);
            _year = ValidateYear(year);
        }
    }
}
```

```
private string ValidateMake(string make)
{
    if (make.Length >= 3)
        return make;
    throw new ArgumentException("Make must be three
        characters or more.");
}

public string Make
{
    get { return _make; }
    set { _make = ValidateMake(value); }
}

// Other methods and properties omitted for brevity.
}
```

The benefit of the preceding code is that if you need to change the validation for the code that gets or sets the data values, you can do so without breaking the implementation.

Data structures should expose data and have no methods

Structures differ from classes in that they use value equality in place of reference equality. Other than that, there is not much difference between a struct and a class.

There is a debate as to whether a data structure should make the variables public or hide them behind get and set properties. It is purely down to you which you choose, but personally I always think it best to hide data even in structs and only provide access via properties and methods. There is one caveat in terms of having clean data structures that are safe, and that is that once created, structs should not allow themselves to be mutated by methods and get properties. The reason for this is that changes to temporary data structures will be discarded.

Let's now look at a simple data structure example.

An example of data structure

The following code is a simple data structure:

```
namespace CH3.Encapsulation
{
    public struct Person
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public Person(int id, string firstName, string lastName)
        {
            Id = id;
            FirstName = firstName;
            LastName = lastName;
        }
    }
}
```

As you can see, the data structure is not that much different from a class in that it has a constructor and properties.

With this, we come to the end of the chapter and will now review what we've learned.

Summary

In this chapter, we learned about organizing our namespaces in folders and packages, and how good organization can help to prevent namespace classes. We then moved on to classes and responsibility and looked at why classes should only have one responsibility. We also looked at cohesion and coupling and why it is important to have high cohesion and low coupling.

Good documentation requires public members to be correctly commented on in documentation tools, and we saw how to do this using XML comments. The importance of why you should design for change was also discussed with basic examples of DI and IoC.

The Law of Demeter showed you how to not to talk to strangers, but only immediate friends, and how to avoid chaining. And finally, we looked at objects and data structures and what they should hide and what they should make public.

In the next chapter, we will briefly cover functional programming in C# and how to write clean methods that are small. We will also learn to avoid having more than two parameters in our methods, as methods with many parameters can become unwieldy. Plus we will learn to avoid duplication which can be a troublesome source of bugs when fixed in one location, but still exist elsewhere in your code.

Questions

1. How can we organize our classes in C#?
2. How many responsibilities should a class have?
3. How do you comment on your code for document generators?
4. What does cohesion mean?
5. What does coupling mean?
6. Should cohesion be high or low?
7. Should coupling be tight or loose?
8. What mechanisms are available that help you design for change?
9. What is DI?
10. What is IoC?
11. Name one benefit of using immutable objects.
12. What should objects hide and show?
13. What should structures hide and show?

Further reading

- For more detail in regard to understanding the different kinds of cohesion and coupling, check out <https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/>.
- Many tutorials on IoC can be found at <https://www.tutorialsteacher.com/ioc/>.

4

Writing Clean Functions

Clean functions are methods that are small (they have two or fewer arguments) and avoid duplication. The ideal method has no parameters and does not modify the program's state. Small methods are less prone to exceptions, so you will be writing much more robust code that benefits you in the long run as you will have fewer bugs to fix.

Functional programming is a software coding methodology that treats computations as the mathematical evaluation of computations. This chapter will teach you the benefits of treating computations as the evaluation of mathematical functions in order to void changing an object's state.

Large methods (also known as functions) can be unwieldy to read and prone to errors, so writing small methods has its advantages. Hence, we will look at how large methods can be broken up into smaller methods. In this chapter, we will cover functional programming in C# and how to write small, clean methods.

Constructors and methods with multiple parameters can become a real pain to work with, so we will have to look for ways to work around and pass multiple parameters, as well as how to avoid using more than two parameters. The main reason for reducing the number of parameters we have is that they can become hard to read, be a source of irritation to fellow programmers, and cause visual stress if there are enough of them. They can also be a sign that the method is trying to do too much, or that you need to consider refactoring your code.

In this chapter, we will cover the following topics:

- Understanding functional programming
- Keeping methods small
- Avoiding duplication
- Avoiding multiple parameters

By the time you have worked through this chapter, you will have the skills to do the following:

- Describe what functional programming is
- Provide existing examples of functional programming in the C# programming language
- Write functional C# code
- Avoid writing methods with more than two arguments
- Write immutable data objects and structures
- Keep your methods small
- Write code that adheres to the **Single Responsibility Principle (SRP)**

Let's get started!

Understanding functional programming

The only thing that sets functional programming aside from other methods of programming is that functions do not modify data or state. You will use functional programming in scenarios such as deep learning, machine learning, and artificial intelligence when it is necessary to perform different sets of operations on the same set of data.

The *LINQ syntax* within .NET Framework is an example of functional programming. So, if you are wondering what functional programming looks like, and if you have used LINQ before, then you have been subjected to functional programming and should know what it looks like.

Since functional programming is a deep subject and many books, courses, and videos exist on this topic, we will only touch on the topic briefly in this chapter by looking at pure functions and immutable data.

A pure function is restricted to only operating on the data that is passed into it. As a result, the method is predictable and avoids producing side effects. This benefits programmers because such methods are easier to reason about and test.

Once an immutable data object or data structure has been initialized, the contained data values will not be modified. Because the data is only set and not modified, you can easily reason about what the data is, how it is set, and what the outcome of any operation will be, given the inputs. Immutable data is also easier to test as you know what your inputs are and what outputs are expected. This makes writing test cases much easier as you don't have so many things to consider, such as object state. The benefit of immutable objects and structures is that they are thread-safe. Thread-safe objects and structures make for good **data transfer objects (DTOs)** that can be passed between threads.

But structs can still be mutable if they contain reference types. One way around this would be to make the reference type immutable. C# 7.2 added support for `readonly struct` and `ImmutableStruct`. So, even if our structures contain reference types, we can now use these new C# 7.2 constructs to make structures with reference types immutable.

Now, let's have a look at a pure function example. The only way to set the properties of an object is via the constructor at construction time. The class is a `Player` class whose only job is to hold the name of the player and their high score. A method is provided that updates the player's high score:

```
public class Player
{
    public string PlayerName { get; }
    public long HighScore { get; }

    public Player(string playerName, long highScore)
    {
        PlayerName = playerName;
        HighScore = highScore;
    }

    public Player UpdateHighScore(long highScore)
    {
        return new Player(PlayerName, highScore);
    }
}
```

Notice that the `UpdateHighScore` method does not update the `HighScore` property. Instead, it instantiates and returns a new `Player` class by passing in the `PlayerName` variable, which is already set in the class, and `highScore`, which is the method parameter. You have now seen a very simple example of how to program your software without changing its state.



Functional programming is a very large subject and requires a mind shift that can be very difficult for both procedural and object-oriented programmers. Since it is outside the scope of this book (to delve deep into the topic of functional programming), you are actively encouraged to peruse the functional programming resources on offer from PacktPub for yourself.

Packt has some very good books and videos that specialize in teaching the top tiers of functional programming. You will find links to some Packt functional programming resources at the end of this chapter, in the *Further reading* section.

Before we move on, we will look at some LINQ examples since LINQ is an example of functional programming in C#. It will be good to have an example dataset. The following code builds a list of vendors and products. We'll start by writing the `Product` structure:

```
public struct Product
{
    public string Vendor { get; }
    public string ProductName { get; }
    public Product(string vendor, string productName)
    {
        Vendor = vendor;
        ProductName = productName;
    }
}
```

Now that we have our struct, we will add some sample data inside the `GetProducts()` method:

```
public static List<Product> GetProducts()
{
    return new List<Products>
    {
        new Product("Microsoft", "Microsoft Office"),
        new Product("Oracle", "Oracle Database"),
        new Product("IBM", "IBM DB2 Express"),
        new Product("IBM", "IBM DB2 Express"),
        new Product("Microsoft", "SQL Server 2017 Express"),
        new Product("Microsoft", "Visual Studio 2019 Community Edition"),
        new Product("Oracle", "Oracle JDeveloper"),
        new Product("Microsoft", "Azure"),
        new Product("Microsoft", "Azure"),
        new Product("Microsoft", "Azure Stack"),
        new Product("Google", "Google Cloud Platform"),
        new Product("Amazon", "Amazon Web Services")
    }
}
```

```
    };  
}
```

Finally, we can start to use LINQ on our list. In the preceding example, we will get a distinct list of products, ordered by the vendor's names, and print out the results:

```
class Program  
{  
    static void Main(string[] args)  
    {  
        var vendors = (from p in GetProducts()  
                        select p.Vendor)  
                        .Distinct()  
                        .OrderBy(x => x);  
        foreach (var vendor in vendors)  
            Console.WriteLine(vendor);  
        Console.ReadKey();  
    }  
}
```

Here, we obtain a list of vendors by calling `GetProducts()` and selecting only the `Vendor` column. Then, we filter the list so that it only includes a vendor once by calling the `Distinct()` method. The list of vendors is then ordered alphabetically by calling `OrderBy(x => x)`, where `x` is the vendor's name. Upon obtaining the ordered list of distinct vendors, we then loop through the list and print the vendor's name. Finally, we wait for the user to press any key to exit the program.

One of the benefits of functional programming is that your methods are much smaller than the methods in other types of programming. Next, we will take a look at why it is good to keep methods small, as well as the techniques we can use, including functional programming.

Keeping methods small

While programming clean and readable code, it is important to keep the methods small. Preferably, in the C# world, it is best to keep methods *under 10 lines* long. The perfect length is no more than *4 lines*. A good way to keep methods small is to consider if you should be trapping for errors or bubbling them further up the call stack. With defensive programming, you can become a little too defensive, and this can add to the amount of code you find yourself writing. Besides, methods that trap errors will be longer than methods that don't.

Let's consider the following code, which can throw an `ArgumentNullException`:

```
public UpdateView(MyEntities context, DataItem dataItem)
{
    InitializeComponent();
    try
    {
        DataContext = this;
        _dataItem = dataItem;
        _context = context;
        nameTextBox.Text = _dataItem.Name;
        DescriptionTextBox.Text = _dataItem.Description;
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex);
        throw;
    }
}
```

In the preceding code, we can clearly see that there are two locations where an `ArgumentNullException` may be raised. The first line of code to potentially raise an `ArgumentNullException` is `nameTextBox.Text = _dataItem.Name;`; the second line of code that may potentially raise the same exception is `DescriptionTextBox.Text = _dataItem.Description;`. We can see that the exception handler catches the exception when it occurs, writes it to the console, and then simply throws it back up the stack.

Notice that, from a human reading perspective, there are *8 lines* of code that form the `try/catch` block.

You can completely replace the `try/catch` exception handling with a single line of text by writing your own argument validator. To explain this, we will provide an example.

Let's start by looking at the `ArgumentValidator` class. The purpose of this class is to throw an `ArgumentNullException` with the name of the method that contains the null argument:

```
using System;
namespace CH04.Validators
{
    internal static class ArgumentValidator
    {
        public static void NotNull(
            string name,
            [ValidatedNotNull] object value
        )
```

```
        {
            if (value == null)
                throw new ArgumentNullException(name);
        }
    }

    [AttributeUsage(
        AttributeTargets.All,
        Inherited = false,
        AllowMultiple = true)
    ]
    internal sealed class ValidatedNotNullAttribute : Attribute
    {
    }
}
```

Now that we have our null validation class, we can perform the new way of validating parameters for null values in our methods. So, let's look at a simple example:

```
public ItemsUpdateView(
    Entities context,
    ItemsView itemView
)
{
    InitializeComponent();
    ArgumentValidator.NotNull("ItemsUpdateView", itemView);
    // ### implementation omitted ###
}
```

As you can clearly see, we have replaced the whole of the `try catch` block with a one-liner at the top of the method. When this validation detects a null argument, an `ArgumentNullException` is thrown, preventing the code from continuing. This makes the code much easier to read, and also helps with debugging.

Now, we'll look at formatting functions with indentation so that they are easy to read.

Indenting code

A very long method is hard to read and follow at the best of times, especially when you have to scroll through the method many times to get to the bottom of it. But having to do that with methods that are not properly formatted with the correct levels of indentation can be a real nightmare.

If you ever encounter any method code that is poorly formatted, then make it your own responsibility, as a professional coder, to tidy the code up before you do anything else. Any code between braces is known as a **code block**. Code within a code block should be indented by one level. Code blocks within code blocks should also be indented by one level, as shown in the following example:

```
public Student Find(List<Student> list, int id)
{
    Student r = null;foreach (var i in list)
    {
        if (i.Id == id)
            r = i;                }        return r;
    }
```

The preceding example demonstrates bad indentation and also bad loop programming. Here, you can see that a list of students is being searched in order to find and return a student with the specified ID that was passed in as a parameter. What annoys some programmers and reduces the performance of the application is that the loop in the preceding code continues, even when the student has been found. We can improve the indentation and the performance of the preceding code as follows:

```
public Student Find(List<Student> list, int id)
{
    Student r = null;
    foreach (var i in list)
    {
        if (i.Id == id)
        {
            r = i;
            break;
        }
    }
    return r;
}
```

In the preceding code, we have improved the formatting and made sure that the code is properly indented. We've added a `break` to the `for` loop so that the `foreach` loop is terminated when a match is found.

Not only is the code now more readable, but it also performs much better. Imagine that the code is being run against a university with 73,000 students on campus and via distance learning. Consider that if the student matches the ID is the first in the list, then without the `break` statement, the code would have to run 72,999 unnecessary computations. You can see how much of a difference the `break` statement makes to the performance of the preceding code.

We have left the return value in its original location as the compiler can complain that not all code paths return a value. This is also why we added the `break` statement. It is clear that proper indentation improves the readability of the code, thus aiding the programmer's understanding of it. This enables the programmer to make any changes that they deem necessary.

Avoiding duplication

Code can be either **DRY** or **WET**. WET code stands for **Write Every Time** and is the opposite of DRY, which stands for **Don't Repeat Yourself**. The problem with WET code is that it is the perfect candidate for *bugs*. Let's say your test team or a customer finds a bug and reports it to you. You fix the bug and pass it on, only for it to come back and bite you as many times as that code is encountered within your computer program.

Now, we DRY our WET code by removing duplication. One way we can do this is by extracting the code and putting it into a method and then centralizing the method in such a way that it is accessible to all the areas of the computer program that need it.

Time for an example. Imagine that you have a collection of expense items that consist of `Name` and `Amount` properties. Now, consider having to get the decimal `Amount` for an expense item by `Name`.

Say you had to do this 100 times. For this, you could write the following code:

```
var amount = ViewModel
    .ExpenseLines
    .Where(e => e.Name.Equals("Life Insurance"))
    .FirstOrDefault()
    .Amount;
```

There is no reason why you can't write that same code 100 times. But there is a way to write it only once, thus reducing the size of your codebase and making you more productive. Let's have a look at how we can do this:

```
public decimal GetValueByName(string name)
{
    return ViewModel
        .ExpenseLines
        .Where(e => e.Name.Equals(name))
        .FirstOrDefault()
        .Amount;
}
```

To extract the required value from the `ExpenseLines` collection within your `ViewModel`, all you have to do is pass the name of the value you require into the `GetValueName(string name)` method, as shown in the following code:

```
var amount = GetValueByName("Life Insurance");
```

That one line of code is very readable, and the lines of code to get the value are contained in a single method. So, if the method needs to be changed for whatever reason (such as a bug fix), you only have to modify the code in one place.

The next logical step to writing good functions is to have as few parameters as possible. In the next section, we'll look at why we should have no more than two parameters, as well as how to work with just parameters, even if we need plenty more.

Avoiding multiple parameters

Niladic methods are the ideal type of methods in C#. Such methods have no parameters (also known as *arguments*). Monadic methods only have one parameter. Dyadic methods have two parameters. Triadic methods have three parameters. Methods that have more than three parameters are known as polyadic methods. You should aim to keep the number of parameters to a minimum (preferably less than three).

In the ideal world of C# programming, you should do your best to avoid triadic and polyadic methods. The reason for this is not because it is bad programming, but because it makes your code easier to read and understand. Methods with lots of parameters can cause visual stress to programmers, and can also be a source of irritation. IntelliSense can also be difficult to read and understand as you add more parameters.

Let's look at a bad example of a polyadic method that updates a user's account information:

```
public void UpdateUserInfo(int id, string username, string firstName,
    string lastName, string addressLine1, string addressLine2, string
    addressLine3, string addressLine3, string addressLine4, string city, string
    postcode, string region, string country, string homePhone, string
    workPhone, string mobilePhone, string personalEmail, string workEmail,
    string notes)
{
    // ### implementation omitted ###
}
```

As shown by the `UpdateUserInfo` method, the code is horrible to read. How can we modify the method so that it transforms from a polyadic method into a monadic method? The answer is simple – we pass in a `UserInfo` object. First of all, before we modify the method, let's take a look at our `UserInfo` class:

```
public class UserInfo
{
    public int Id { get; set; }
    public string Username { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string AddressLine1 { get; set; }
    public string AddressLine2 { get; set; }
    public string AddressLine3 { get; set; }
    public string AddressLine4 { get; set; }
    public string City { get; set; }
    public string Region { get; set; }
    public string Country { get; set; }
    public string HomePhone { get; set; }
    public string WorkPhone { get; set; }
    public string MobilePhone { get; set; }
    public string PersonalEmail { get; set; }
    public string WorkEmail { get; set; }
    public string Notes { get; set; }
}
```

We now have a class that contains all the information we need to pass into the `UpdateUserInfo` method. The `UpdateUserInfo` method can now be transformed from a polyadic method into a monadic method, as follows:

```
public void UpdateUserInfo(UserInfo userInfo)
{
    // ### implementation omitted ###
}
```

How much better does the preceding code look? It is smaller and much more readable. The rule of thumb should be to have less than three parameters, and ideally none. If your class is obeying the SRP, then consider implementing the *parameter object pattern*, as we have done here.

Implementing SRP

All objects and methods that you write should, at most, have one responsibility and no more. Objects can have multiple methods, but those methods, when combined, should all work toward the single purpose of the object they belong to. Methods can call multiple methods, where each does different things. But the method itself should only do one thing.

A method that knows and does far too much is known as a **God method**. And likewise, an object that knows and does too much is known as a **God object**. God objects and methods are hard to read, maintain, and debug. Such objects and methods can often have the same bug repeated many times. People who are good at the programming craft will avoid God objects and God methods. Let's look at a method that is doing more than one thing:

```
public void SrpBrokenMethod(string folder, string filename, string text,
    emailFrom, password, emailTo, subject, message, mediaType)
{
    var file = $"{folder}{filename}";
    File.WriteAllText(file, text);
    MailMessage message = new MailMessage();
    SmtplibClient smtp = new SmtplibClient();
    message.From = new MailAddress(emailFrom);
    message.To.Add(new MailAddress(emailTo));
    message.Subject = subject;
    message.IsBodyHtml = true;
    message.Body = message;
    Attachment emailAttachment = new Attachment(file);
    emailAttachment.ContentDisposition.Inline = false;
    emailAttachment.ContentDisposition.DispositionType =
        DispositionTypeNames.Attachment;
    emailAttachment.ContentType.MediaType = mediaType;
    emailAttachment.ContentType.Name = Path.GetFileName(filename);
    message.Attachments.Add(emailAttachment);
    smtp.Port = 587;
    smtp.Host = "smtp.gmail.com";
    smtp.EnableSsl = true;
    smtp.UseDefaultCredentials = false;
    smtp.Credentials = new NetworkCredential(emailFrom, password);
    smtp.DeliveryMethod = SmtplibDeliveryMethod.Network;
    smtp.Send(message);
}
```

`SrpBrokenMethod` is clearly doing more than one thing, so it breaks the SRP. We will now break this method down into a number of smaller methods that only do one thing. We will also address the issue of the polyadic nature of the method in that it has more than two parameters.

Before we begin to break down the method into smaller methods that do only one thing, we need to look at all the actions that the method is performing. The method starts by writing text to a file. It then creates an email message, assigns an attachment, and finally sends the email. So, for this, we need methods for the following:

- Write text to file
- Create an email message
- Add an email attachment
- Send email

Looking at the current method, we have four parameters that are passed into it for writing text to a file: one for the folder, one for the filename, one for the text, and one for the media type. The folder and filename can be combined into a single parameter called `filename`. If `filename` and `folder` are two separate variables that are used inside the calling code, then they can be passed into the method as a single interpolated string, such as `${folder}${filename}`.

As for the media type, this could be privately set inside a struct during construction time. We could use that struct to set the properties we need so that we can pass the struct in with the three properties as a single parameter. Let's look at the code that accomplishes this:

```
public struct TextFileData
{
    public string FileName { get; private set; }
    public string Text { get; private set; }
    public MimeTypes.MimeType MimeType { get; }

    public TextFileData(string filename, string text)
    {
        Text = text;
        MimeType = MimeTypes.TextPlain;
        FileName = $"{filename}-{GetFileTimestamp()}";
    }

    public void SaveTextFile()
    {
        File.WriteAllText(FileName, Text);
    }

    private static string GetFileTimestamp()
    {
        var year = DateTime.Now.Year;
        var month = DateTime.Now.Month;
        var day = DateTime.Now.Day;
        var hour = DateTime.Now.Hour;
```

```

        var minutes = DateTime.Now.Minute;
        var seconds = DateTime.Now.Second;
        var milliseconds = DateTime.Now.Millisecond;
        return
        $"{year}{month}{day}@{hour}{minutes}{seconds}{milliseconds}";
    }
}

```

The `TextFileData` constructor ensures that the `FileName` value is unique by calling the `GetFileTimestamp()` method and appending it to the end of `FileName`. To save the text file, we call the `SaveTextFile()` method. Notice that `MimeType` is set internally and is set to `MimeType.TextPlain`. We could have simply hardcoded `MimeType` as `MimeType = "text/plain";`, but the advantage of using an enum is that the code is reusable, with the added benefit of you not having to remember the text for a specific `MimeType` or look it up on the internet. Now, we'll code enum and add a description to the enum value:

```

[Flags]
public enum MimeType
{
    [Description("text/plain")]
    TextPlain
}

```

Well, we've got our enum, but now we need a way to extract the description so that it can be easily assigned to a variable. Therefore, we will create an extension class that will enable us to get the description of an enum. This enables us to set `MimeType`, as follows:

```
MimeType = MimeType.TextPlain;
```

Without the extension method, the value of `MimeType` would be 0. But with the extension method, the value of `MimeType` is "text/plain". You can now reuse this extension in other projects and build it up as you require.

The next class we will write is the `Smtp` class, whose responsibility is to send an email via the `Smtp` protocol:

```

public class Smtp
{
    private readonly SmtpClient _smtp;

    public Smtp(Credential credential)
    {
        _smtp = new SmtpClient
        {
            Port = 587,
            Host = "smtp.gmail.com",

```

```
        EnableSsl = true,
        UseDefaultCredentials = false,
        Credentials = new NetworkCredential(
            credential.EmailAddress, credential.Password),
        DeliveryMethod = SmtpDeliveryMethod.Network
    };
}
public void SendMessage(MailMessage mailMessage)
{
    _smtp.Send(mailMessage);
}
}
```

The `Smtp` class has a constructor that takes a single parameter of the `Credential` type. This credential is used to log into the email server. The server is configured in the constructor. When the `SendMessage(MailMessage mailMessage)` method is called, the message is sent.

Let's write a `DemoWorker` class that splits the work into different methods:

```
public class DemoWorker
{
    TextFileData _textFileData;

    public void DoWork()
    {
        SaveTextFile();
        SendEmail();
    }

    public void SendEmail()
    {
        Smtp smtp = new Smtp(new Credential("fakegmail@gmail.com",
            "fakeP@55w0rd"));
        smtp.SendMessage(GetMailMessage());
    }

    private MailMessage GetMailMessage()
    {
        var msg = new MailMessage();
        msg.From = new MailAddress("fakegmail@gmail.com");
        msg.To.Add(new MailAddress("fakehotmail@hotmail.com"));
        msg.Subject = "Some subject";
        msg.IsBodyHtml = true;
        msg.Body = "Hello World!";
        msg.Attachments.Add(GetAttachment());
        return msg;
    }
}
```



```
private Attachment GetAttachment()
{
    var attachment = new Attachment(_textFileData.FileName);
    attachment.ContentDisposition.Inline = false;
    attachment.ContentDisposition.DispositionType =
        DispositionTypeNames.Attachment;
    attachment.ContentType.MediaType =
        MimeTypes.TextPlain.Description();
    attachment.ContentType.Name =
        Path.GetFileName(_textFileData.FileName);
    return attachment;
}
private void SaveTextFile()
{
    _textFileData = new TextFileData(
        $"{Environment.SpecialFolder.MyDocuments}attachment",
        "Here is some demo text!"
    );
    _textFileData.SaveTextFile();
}
}
```

The `DemoWorker` class shows a much cleaner version of sending an email message. The main method responsible for saving an attachment and sending it as an attachment via email is called `DoWork()`. This method only contains two lines of code. The first line calls the `SaveTextFile()` method, while the second line calls the `SendEmail()` method.

The `SaveTextFile()` method creates a new `TextFileData` struct and passes in the filename and some text. It then calls the `SaveTextFile()` method in the `TextFileData` struct, which is responsible for saving the text to the file specified.

The `SendEmail()` method creates a new `Smtp` class. The `Smtp` class has a `Credential` parameter, while the `Credential` class has two string parameters for email address and password. The email and password are used to log into the SMTP server. Once the SMTP server has been created, the `SendMessage(MailMessage mailMessage)` method is called.

This method requires a `MailMessage` object to be passed in. So, we have a method called `GetMailMethod()` that builds a `MailMessage` object that is then passed into the `SendMessage(MailMessage mailMessage)` method. `GetMailMethod()` adds an attachment to `MailMessage` by calling the `GetAttachment()` method.

As you can see from these modifications, our code is now more compact and readable. That is the key to good quality code that is easy to modify and maintain: it must be easy to read and understand. That is why it is important for your methods to be small and clean with as few parameters as possible.

Does your method break the SRP? If it does, you should consider breaking the method up into as many methods as there are responsibilities. And that concludes this chapter on writing clean functions. It is now time to summarize what you have learned and test your knowledge.

Summary

In this chapter, you have seen how functional programming can improve the safety of your code by not modifying the state, which can give rise to bugs, especially in multithreaded applications. By keeping methods small with meaningful names and no more than two parameters, you have seen how much cleaner your code is and easier to read. You have also seen how we can remove duplication in our code and the benefits of doing so. Code that is easy to read is easier to maintain and extend than code that is hard to read and decipher!

We will now move on and look at the topic of exception handling. In the next chapter, you will learn how to use exception handling appropriately, write your own custom C# exceptions that provide meaningful information, and write code that avoids raising `NullPointerExceptions`.

Questions

1. What do you call a method that has no parameters?
2. What do you call a method that has one parameter?
3. What do you call a method that has two parameters?
4. What do you call a method that has three parameters?
5. What do you call a method that has more than three parameters?
6. What two method types should be avoided and why?
7. In layman's terms, what is functional programming?
8. What are some advantages of functional programming?
9. Name one disadvantage of functional programming.
10. What is WET code, and why should it be avoided?

11. What is DRY code, and why should you use it?
12. How do you DRY out WET code?
13. Why should methods be as small as possible?
14. How do you implement validation without having to implement `try/catch` blocks?

Further reading

Here are some additional resources so that you can delve deeper into the realms of C# functional programming:

- *Functional C#* by Wisnu Anggoro: <https://www.packtpub.com/application-development/functional-c>. This book is devoted to C# functional programming and is a good place to start if you want to know more.
- *Functional Programming in C#* by Jovan Poppavic (MSFT): <https://www.codeproject.com/Articles/375166/Functional-programming-in-Csharp>. This is an in-depth article on functional C# programming. It contains diagrams and has a 5-star rating.

5

Exception Handling

In the previous chapter, we looked at functions. Despite the best efforts of programmers to write robust code, functions will, at some point, generate exceptions. This could be for a number of reasons, such as a missing file or folder, an empty or null value, the location can't be written to, or the user is denied access. So, with that in mind, in this chapter, you will learn about appropriate ways to use exception handling to produce clean C# code. First, we will start by looking at checked and unchecked exceptions with regards to arithmetic `OverflowExceptions`. We will look at what they are, why they are used, and some examples of them being used in code.

Then, we'll look at how we can avoid the `NullReferenceException` exception. After that, we'll look at implementing specific business rules for specific types of exceptions. With our fresh understanding of exceptions and exception business rules, we will set about building our own custom exceptions and then finish off by looking at why we should not use exceptions to control the flow of our computer programs.

In this chapter, we will cover the following topics:

- Checked and unchecked exceptions
- Avoiding `NullReferenceExceptions`
- Business rule exceptions
- Exceptions should provide meaningful information
- Building your own custom exceptions

By the end of this chapter, you will have the skills to do the following:

- You will be able to understand what checked and unchecked exceptions are, and why they are in C#.
- You will be able to understand what an `OverflowException` is and how to trap them at compile time.
- You will know what `NullReferenceExceptions` are and how to avoid them.

- You will be able to write your own custom exceptions that provide meaningful information to the customer and that aid you and fellow programmers to easily identify and resolve any issues that are raised.
- You will be able to understand why you should not use exceptions to control program flow.
- You will know how to replace business rule exceptions with C# statements and Boolean checks to control program flow.

Checked and unchecked exceptions

In unchecked mode, an arithmetic overflow is *ignored*. In this situation, the high-order bits that cannot be assigned to the destination type are discarded from the result.

By default, C# operates in the unchecked context while performing non-constant expressions at runtime. But compile-time constant expressions are *always* checked by default. When an arithmetic overflow is encountered in checked mode, an `OverflowException` is raised. One reason why unchecked exceptions are used is to increase performance. Checked exceptions can decrease the performance of methods by a small amount.

The rule of thumb is to make sure that you perform arithmetic operations in the checked context. Any arithmetic overflow exceptions will be picked up as compile-time errors, and you can then fix them before you release your code. That is much better than releasing your code and then having to fix customer runtime errors.

Running code in unchecked mode is dangerous as you are making assumptions about the code. Assumptions are not facts and they can lead to exceptions being raised at runtime. Runtime exceptions lead to poor customer satisfaction and can produce serious follow-on exceptions that negatively impact a customer in some way.

Allowing an application to continue running that has experienced an overflow exception is very dangerous from a business perspective. The reason for this is that data can end up in a non-reversible invalid state. If the data is critical customer data, then this can be considerably costly to the business, and you don't want that on your shoulders.

Consider the following code. This code demonstrates how bad an unchecked overflow can be in the world of customer banking:

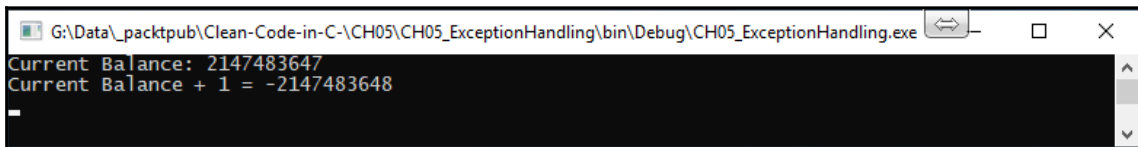
```
private static void UncheckedBankAccountException()
{
    var currentBalance = int.MaxValue;
    Console.WriteLine($"Current Balance: {currentBalance}");
}
```

```

        currentBalance = unchecked(currentBalance + 1);
        Console.WriteLine($"Current Balance + 1 = {currentBalance}");
        Console.ReadKey();
    }

```

Imagine the horror on this customer's face when they see that adding £1 to their bank balance of £2,147,483,647 causes them to be in debt by -£2,147,483,648!



Now, it's time to demonstrate checked and unchecked exceptions with some code examples. First, start a new **console application** and declare some variables:

```
static byte y, z;
```

The preceding code declares two bytes that we will use in our arithmetic code examples. Now, add the `CheckedAdd()` method. This method will raise a checked `OverflowException` if an arithmetic overflow is encountered when adding two numbers that result in a number that is too big to be stored as a byte:

```

private static void CheckedAdd()
{
    try
    {
        Console.WriteLine("### Checked Add ###");
        Console.WriteLine($"x = {y} + {z}");
        Console.WriteLine($"x = {checked((byte) (y + z))}");
    }
    catch (OverflowException oex)
    {
        Console.WriteLine($"CheckedAdd: {oex.Message}");
    }
}

```

Then, write the `CheckedMultiplication()` method. Again, a checked `OverflowException` will be raised if an arithmetic overflow is detected during the multiplication, which results in a number that is larger than a byte:

```

private static void CheckedMultiplication()
{
    try
    {

```

```
        Console.WriteLine("### Checked Multiplication ###");
        Console.WriteLine($"x = {y} x {z}");
        Console.WriteLine($"x = {checked((byte) (y * z))}");
    }
    catch (OverflowException oex)
    {
        Console.WriteLine($"CheckedMultiplication: {oex.Message}");
    }
}
```

Next, we add the `UncheckedAdd()` method. This method will ignore any overflow that happens as a result of an addition, and so an `OverflowException` will not be raised. The result of this overflow will be stored as a byte, but the value will be incorrect:

```
private static void UncheckedAdd()
{
    try
    {
        Console.WriteLine("### Unchecked Add ###");
        Console.WriteLine($"x = {y} + {z}");
        Console.WriteLine($"x = {unchecked((byte) (y + z))}");
    }
    catch (OverflowException oex)
    {
        Console.WriteLine($"CheckedAdd: {oex.Message}");
    }
}
```

And now, we add the `UncheckedMultiplication()` method. This method will not throw an `OverflowException` when an overflow is encountered as the result of this multiplication. The exception will simply be ignored. This will result in an incorrect number being stored as a byte:

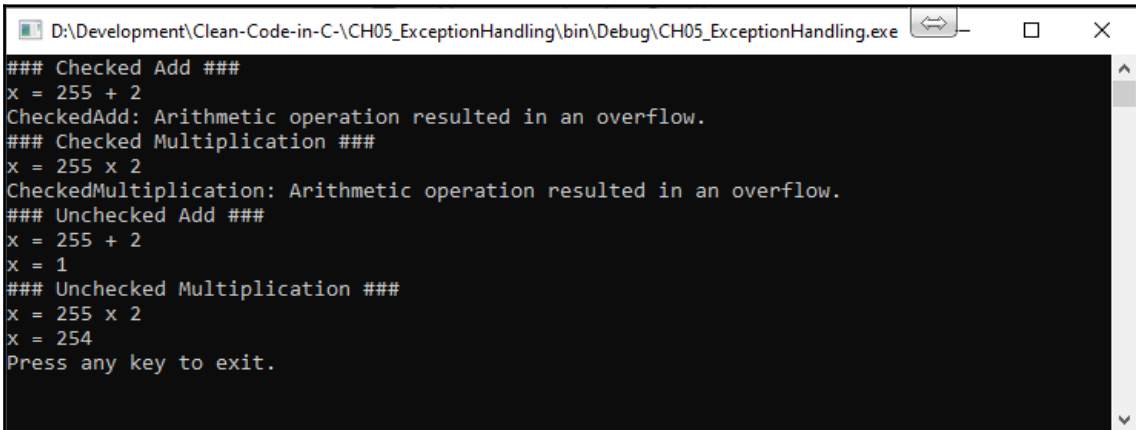
```
private static void UncheckedMultiplication()
{
    try
    {
        Console.WriteLine("### Unchecked Multiplication ###");
        Console.WriteLine($"x = {y} x {z}");
        Console.WriteLine($"x = {unchecked((byte) (y * z))}");
    }
    catch (OverflowException oex)
    {
        Console.WriteLine($"CheckedMultiplication: {oex.Message}");
    }
}
```

Finally, it is time to modify our `Main(string[] args)` method so that we can initialize the variables and execute the methods. Here, we add the maximum value for a byte to the `y` variable and 2 to the `z` variable. Then, we run the `CheckedAdd()` and `CheckedMultiplication()` methods, which will both generate `OverflowException()`. This is thrown because the `y` variable contains the maximum value for a byte.

So, by adding or multiplying by 2, you are exceeding the address space needed to store the variable. Next, we will run the `UncheckedAdd()` and `UncheckedMultiplication()` methods. Both these methods ignore overflow exceptions, assign the result to the `x` variable, and disregard any bits that overflow. Finally, we print a message to the screen and then exit when the user presses any key:

```
static void Main(string[] args)
{
    y = byte.MaxValue;
    z = 2;
    CheckedAdd();
    CheckedMultiplication();
    UncheckedAdd();
    UncheckedMultiplication();
    Console.WriteLine("Press any key to exit.");
    Console.ReadLine();
}
```

When we run the preceding code, we end up with the following output:



```
## Checked Add ##
x = 255 + 2
CheckedAdd: Arithmetic operation resulted in an overflow.
## Checked Multiplication ##
x = 255 x 2
CheckedMultiplication: Arithmetic operation resulted in an overflow.
## Unchecked Add ##
x = 255 + 2
x = 1
## Unchecked Multiplication ##
x = 255 x 2
x = 254
Press any key to exit.
```

As you can see, when we use checked exceptions, exceptions are raised when `OverflowException` is encountered. But when we use unchecked exceptions, no exception is raised.

It is apparent from the preceding screenshot that problems can arise from unexpected values and that certain behaviors can arise from using unchecked exceptions. Therefore, the rule of thumb when performing arithmetic operations must be to always use checked exceptions.

Now, let's move on and look at a very common exception that is encountered frequently by programmers, known as `NullPointerException`.

Avoiding `NullPointerException`s

`NullPointerException` is a common exception that has been experienced by most programmers. It is thrown when an attempt is made to access a property or method on a null object.

To defend against computer program crashes, the common course of action among fellow programmers is to use `try{...}catch (NullPointerException){...}` blocks. This is a part of defensive programming. But the problem is that, a lot of the time, the error is simply *logged* and *rethrown*. Besides this, a lot of wasted computations are performed that could have been avoided.

A much better way of handling `ArgumentNullException`s is to implement `ArgumentNullValidator`. The parameters of a method are usually the source of a null object. It makes sense to test the parameters of a method before they are used and, if they are found to be invalid for any reason, to throw an appropriate `Exception`. In the case of `ArgumentNullValidator`, you would place this validator at the top of the method and then test each parameter. If any parameter was found to be null, then `NullPointerException` would be thrown. This would save computations and remove the need to wrap your method's code in a `try...catch` block.

To make things clear, we will write `ArgumentNullValidator` and use it in a method to test the method's arguments:

```
public class Person
{
    public string Name { get; }
    public Person(string name)
    {
        Name = name;
    }
}
```

In the preceding code, we have created the `Person` class with a single read-only property called `Name`. This will be the object that we will use to pass into the example methods to cause `NullReferenceException`. Next, we will create our `Attribute` for the validator called `ValidatedNotNullAttribute`:

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple =
true)]
internal sealed class ValidatedNotNullAttribute : Attribute { }
```

Now that we have our `Attribute`, it's time to write the validator:

```
internal static class ArgumentNullValidator
{
    public static void NotNull(string name,
        [ValidatedNotNull] object value)
    {
        if (value == null)
        {
            throw new ArgumentException(name);
        }
    }
}
```

`ArgumentNullValidator` takes two arguments:

- The name of the object
- The object itself

The object is checked to see if it is `null`. If it is `null`, `ArgumentException` is thrown, passing in the name of the object.

The following method is our `try/catch` example method. Notice that we log a message and throw the exception. However, we don't use the declared exception parameter, and so by rights, this should be removed. You will see this quite often in code. It is unnecessary and should be removed to tidy the code up:

```
private void TryCatchExample(Person person)
{
    try
    {
        Console.WriteLine($"Person's Name: {person.Name}");
    }
    catch (NullReferenceException nre)
    {
        Console.WriteLine("Error: The person argument cannot be null.");
        throw;
    }
}
```

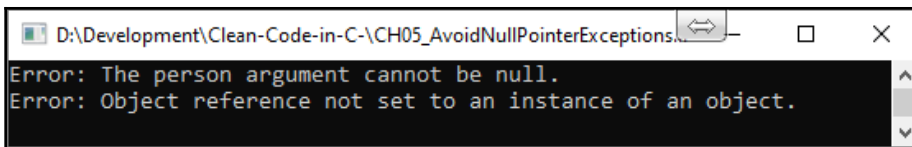
```
    }  
}
```

Next, we will write our example method that will use `ArgumentNullValidator`. We will call it `ArgumentNullValidatorExample`:

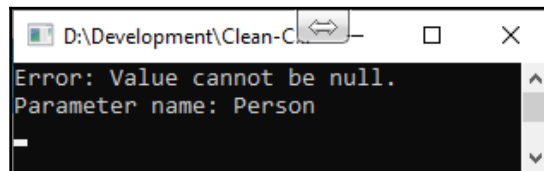
```
private void ArgumentNullValidatorExample(Person person)  
{  
    ArgumentNullValidator.NotNull("Person", person);  
    Console.WriteLine($"Person's Name: {person.Name}");  
    Console.ReadKey();  
}
```

Notice that we have gone from nine lines, including braces, to only two lines. We also don't attempt to use the value before it has been validated. All we need to do now is modify our `Main` method to run the methods. Test each method by commenting out one of the methods and running the program. When you do this, it is best to step through your code to see what's going on.

The following is the output of running the `TryCatchExample` method:



The following is the output of running `ArgumentNullValidatorExample`:



If you study the previous screenshots carefully, you will see that we have only logged the error once when using `ArgumentNullValidatorExample`. When throwing the exception using `TryCatchExample`, the exception is logged twice.

The first time, we have a meaningful message, but the second time, the message is *cryptic*. However, the exception that is logged by the calling method, `Main`, is not cryptic at all. It is, in fact, very helpful as it shows us that the value cannot be null for the `Person` parameter.

Hopefully, this section has shown you the value of checking your parameters in your constructors and methods before you use them. By doing this, you can see how argument validators reduce your code, thus making it more readable.

Now, we will look at implementing business rules for specific exceptions.

Business rule exceptions

Technical exceptions are exceptions that are thrown by a computer program as a result of programmer mistakes and/or environmental issues such as there not being enough disk space.

But business rule exceptions are different. Business rule exceptions imply that such behavior is expected and is used to control program flow, when in fact, exceptions should be an exception to the normal flow of the program and not the expected output of a method.

For example, picture a person at an ATM drawing out £100 from their account that has £0 in it and does not have the ability to go overdrawn. The ATM accepts the user request to draw £100 out, and so it issues the `Withdraw(100);` command. The `Withdraw` method checks the balance, discovers that the account has insufficient funds, and so throws `InsufficientFundsException()`.

You may think that having such exceptions is a good idea as they are explicit and help identify issues so that you can carry out a very specific action upon receiving such exceptions – but no! This is not a good idea.

In such a scenario, when the user submits the request, the amount requested should be checked to see if it can be withdrawn. If it can, then the transaction should go ahead, as requested by the user. But if the validation check identifies that the transaction is unable to go ahead, then the program should follow normal program flow to cancel the transaction and inform the user who issued the request without raising an exception.

The withdrawal scenario we've just looked at shows that the programmer has correctly pondered upon the normal flow of the program and the different outcomes. The program flow has been appropriately coded using Boolean checks to allow for the successful withdrawal transactions and to prevent disallowed withdrawal transactions.

Let's see how we would implement a withdrawal from a bank account that does not allow an overdraft scenario using **Business Rule Exceptions (BREs)**. Then, we'll take a look at how we would implement the same scenario but using normal program flow instead of employing BREs.

Start a new console application and add two folders called `BankAccountUsingExceptions` and `BankAccountUsingProgramFlow`. Update your `void Main(string[] args)` method with the following code:

```
private static void Main(string[] args)
{
    var usingBrExceptions = new UsingBusinessRuleExceptions();
    usingBrExceptions.Run();
    var usingPflow = new UsingProgramFlow();
    usingPflow.Run();
}
```

The preceding code runs each scenario. `UsingBusinessRuleExceptions()` demonstrates the use of exceptions as the expected output that's used to control program flow, while `UsingProgramFlow()` demonstrates the clean way of controlling program flow without the use of exceptional conditions.

We now need a class to hold our current account information. So, add a class called `CurrentAccount` to your Visual Studio console project, as follows:

```
internal class CurrentAccount
{
    public long CustomerId { get; }
    public decimal AgreedOverdraft { get; }
    public bool IsAllowedToGoOverdrawn { get; }
    public decimal CurrentBalance { get; }
    public decimal AvailableBalance { get; private set; }
    public int AtmDailyLimit { get; }
    public int AtmWithdrawalAmountToday { get; private set; }
}
```

The properties of this class can only be set internally or externally via the constructor. Now, add the constructor that takes the customer identifier as the only parameter:

```
public CurrentAccount(long customerId)
{
    CustomerId = customerId;
    AgreedOverdraft = GetAgreedOverdraftLimit();
    IsAllowedToGoOverdrawn = GetIsAllowedToGoOverdrawn();
    CurrentBalance = GetCurrentBalance();
    AvailableBalance = GetAvailableBalance();
    AtmDailyLimit = GetAtmDailyLimit();
    AtmWithdrawalAmountToday = 0;
}
```

The current account constructor initializes all the properties. As shown in the preceding code, some properties are initialized using methods. Let's implement each of the methods in turn:

```
private static decimal GetAgreedOverdraftLimit()
{
    return 0;
}
```

`GetAgreedOverdraftLimit()` returns the value of the agreed overdraft limit on the account. In this example, it is hardcoded to zero. But in a real scenario, it would extract the actual figure from a configuration file or other data store. This would allow non-technical users to update the agreed overdraft limit without developers having to change the code.

`GetIsAllowedToGoOverdrawn()` determines if the account can be overdrawn, even if it has not been agreed, as some banks allow. In this case, we just return `false` to determine that the account is unable to go overdrawn:

```
private static bool GetIsAllowedToGoOverdrawn()
{
    return false;
}
```

For the purpose of this example, we will set the user's account balance to £250 in the `GetCurrentBalance()` method:

```
private static decimal GetCurrentBalance()
{
    return 250.00M;
}
```

As a part of our example, we need to make sure that even if the person has £250 in their account, but their available balance is less than that, they are unable to withdraw more than the available balance as this would cause them to go overdrawn. To do this, we will set the available balance to £173.64 in the `GetAvailableBalance()` method:

```
private static decimal GetAvailableBalance()
{
    return 173.64M;
}
```

Here, in the UK, ATM machines will either allow you to withdraw a maximum of £200 or £250. So, in the `GetAtmDailyLimit()` method, we will set the ATM daily limit to £250:

```
private static int GetAtmDailyLimit()
{

```

```
        return 250;
    }
```

Let's write the code for our two scenarios by using business rule exceptions and normal program flow to handle different conditions within a program.

Example 1 – handling conditions with business rule exceptions

Add a new class to your project called `UsingBusinessRuleExceptions` and then add the following `Run ()` method:

```
public class UsingBusinessRuleExceptions
{
    public void Run()
    {
        ExceedAtmDailyLimit();
        ExceedAvailableBalance();
    }
}
```

The `Run ()` method calls two methods:

- The first method is called `ExceedAtmDailyLimit ()`. This method intentionally exceeds the daily amount that is allowed to be withdrawn from an ATM. `ExceedAtmDailyLimit ()` causes `ExceededAtmDailyLimitException`.
- Secondly, the `ExceedAvailableBalance ()` method is called, which intentionally causes an `InsufficientFundsException`. Add the `ExceedAtmDailyLimit ()` method:

```
private void ExceedAtmDailyLimit()
{
    try
    {
        var customerAccount = new CurrentAccount(1);
        customerAccount.Withdraw(300);
        Console.WriteLine("Request accepted. Take cash and card.");
    }
    catch (ExceededAtmDailyLimitException eadlex)
    {
        Console.WriteLine(eadlex.Message);
    }
}
```

The `ExceedAtmDailyLimit()` method creates a new `CustomerAccount` method and passes in the customer's identifier, as represented by the number 1. Then, an attempt is made to withdraw £300. If the request is successful, then the message `Request accepted. Take cash and card.` is printed to the console window. Should the request fail, then the method traps `ExceededAtmLimitException` and prints the exception's message to the console window:

```
private void ExceedAvailableBalance()
{
    try
    {
        var customerAccount = new CurrentAccount(1);
        customerAccount.Withdraw(180);
        Console.WriteLine("Request accepted. Take cash and card.");
    }
    catch (InsufficientFundsException ifex)
    {
        Console.WriteLine(ifex.Message);
    }
}
```

The `ExceedAvailableBalance()` method creates a new `CurrentAccount` and passes in the customer identifier, as represented by the number 1. An attempt is then made to withdraw £180. Since `GetAvailableMethod()` returns £173.64, the method causes an `InsufficientFundsException`.

With that, we've seen how to manage different conditions using business rule exceptions. Now, let's look at the proper way to manage the same conditions using normal program flow, without the use of exceptions.

Example 2 – handling conditions with normal program flow

Add a class called `UsingProgramFlow` and then add the following code to it:

```
public class UsingProgramFlow
{
    private int _requestedAmount;
    private readonly CurrentAccount _currentAccount;

    public UsingProgramFlow()
    {
        _currentAccount = new CurrentAccount(1);
    }
}
```



```
    }  
}
```

In the constructor of the `UsingProgramFlow` class, we will create a new `CurrentAccount` class and pass in the customer identifier. Next, we'll add the `Run()` method:

```
public void Run()  
{  
    _requestedAmount = 300;  
    Console.WriteLine($"Request: Withdraw {_requestedAmount}");  
    WithdrawMoney();  
    _requestedAmount = 180;  
    Console.WriteLine($"Request: Withdraw {_requestedAmount}");  
    WithdrawMoney();  
    _requestedAmount = 20;  
    Console.WriteLine($"Request: Withdraw {_requestedAmount}");  
    WithdrawMoney();  
}
```

The `Run()` method sets the `_requestedAmount` variable three times. Each time it does this, a message is printed stating the withdrawn amount on the console window before calling the `WithdrawMoney()` method. Now, add the `ExceedsDailyLimit()` method:

```
private bool ExceedsDailyLimit()  
{  
    return (_requestedAmount > _currentAccount.AtmDailyLimit)  
        || (_requestedAmount + _currentAccount.AtmWithdrawalAmountToday >  
            _currentAccount.AtmDailyLimit);  
}
```

The `ExceedDailyLimit()` method returns `true` if `_requestedAmount` exceeds the daily ATM withdrawal limit. Otherwise, it returns `false`. Now, add the `ExceedsAvailableBalance()` method:

```
private bool ExceedsAvailableBalance()  
{  
    return _requestedAmount > _currentAccount.AvailableBalance;  
}
```

The `ExceedsAvailableBalance()` method returns `true` if the requested amount is more than is available for withdrawal. Finally, we come to the last method, called `WithdrawMoney()`:

```
private void WithdrawMoney()  
{  
    if (ExceedsDailyLimit())  
        Console.WriteLine("Cannot exceed ATM Daily Limit. Request
```

```
denied.");
    else if (ExceedsAvailableBalance())
        Console.WriteLine("Cannot exceed available balance. You have no
agreed
    overdraft facility. Request denied.");
    else
        Console.WriteLine("Request granted. Take card and cash.");
}
```

The `WithdrawMoney()` method does not use BREs to control the program flow. Instead, this method calls Boolean validation methods that determine the program flow. If `_requestedAmount` exceeds the ATM daily limit, as determined by the call to `ExceedsDailyLimit()`, then the request is denied. Otherwise, the next check is carried out to see if `_requestedAmount` is more than `AvailableBalance`. If it is, then the request is rejected. If not, then the code is executed that grants the request.

I hope you can see that it makes more sense to control the flow of a program using the available logic rather than expecting exceptions to be thrown. The code is a lot cleaner and more correct. Exceptions should be reserved for exceptional circumstances that are not a part of the business requirements.

When proper exceptions are raised in the correct manner, it is important for them to be meaningful. Cryptic error messages are no good for anyone and can actually add unnecessary stress for end users or developers. Now, we are going to look at providing meaningful information in any of the exceptions that are raised by our computer programs.

Exceptions should provide meaningful information

Critical errors that state "There is no error" and then kill a program are just not useful at all. I have experienced the actual "There is no error" critical exception first hand. It is a critical exception that stops an application from working. Yet the message is informing us that there is no error. Well, if there is no error, then why has a critical exception warning appeared on the screen? And why am I unable to continue using the application? Obviously, for the critical exception to be raised, there must be a critical exception somewhere that occurred. But where and why?

What makes such exceptions even more annoying is when they are deep-rooted in the framework or library that you are using (which you have no control over), and where you have no access to the source code. Such exceptions have caused programmers to say negative things out of frustration. I've been guilty of this and I've experienced fellow colleagues do the same. One of the main reasons for the frustration is the unhelpful fact that the code has raised an error and the user or programmer has been informed, but there is no helpful information to suggest what the problem is or where to look or even what remedial action to take.

Exceptions must provide information that is human-friendly, especially to the technically challenged. During my time developing dyslexia testing and assessment software, I have worked with many teachers and IT technicians.

It can be said that many IT technicians and teachers at all levels of ability have often been clueless when it comes to responding to software exception messages.

One error that has perplexed many of the end users of the software I've supported has been **Error 76: Path not found**. This is an old Microsoft exception that has been around as far back as Windows 95, and that still exists today. For the end user of the software that raises this exception, the error message is totally useless. It would be useful for the end user to know what file and location cannot be found and to know what steps to take to remedy the situation.

A potential solution would be to implement the following steps:

1. Check for the existence of the location.
2. If the location does not exist or access is denied, then display the file save or open dialog as needed.
3. Save the user-selected location to a configuration file for future use.
4. On subsequent runs of the same code, use the location set by the user.

But if you were to stay with the error message, then you should at least provide the name of the location and/or file that is missing.

With that said, it is now time to look at how we can build our own exceptions to provide just the right amount of information that will be useful to the end user and to the programmer. But take note: you must be careful not to disclose sensitive information or data.

Building your own custom exceptions

Microsoft .NET Framework already has a good number of exceptions that can be raised that you are able to trap. But there may be instances where you'll require a custom exception that provides more detailed information or that is more end user friendly in its terminology.

So, we are now going to look at what the requirements are for building our own custom exceptions. It is surprisingly simple to build your own custom exception. All you have to do is give your class a name that ends with `Exception` and inherit from `System.Exception`. Then, you need to add three constructors, as shown in the following code example:

```
public class TickerListNotFoundException : Exception
{
    public TickerListNotFoundException() : base()
    {
    }

    public TickerListNotFoundException(string message)
        : base(message)
    {
    }

    public TickerListNotFoundException(
        string message,
        Exception innerException
    )
        : base(message, innerException)
    {
    }
}
```

`TickerListNotFoundException` inherits from the `System.Exception` class. It contains three mandatory constructors:

- A default constructor
- A constructor that accepts a string of text for the exception message
- A constructor that accepts a string of text for the exception message and an `Exception` object for the inner exception

We are now going to write and execute three methods that will use each of our custom exception's constructors. You will be able to clearly see the benefit of using custom exceptions to create more meaningful exceptions:

```
static void Main(string[] args)
{
    ThrowCustomExceptionA();
    ThrowCustomExceptionB();
    ThrowCustomExceptionC();
}
```

The preceding code shows our updated `Main(string[] args)` method, which has been updated to execute our three methods. These will test each of our custom exception's constructors in turn:

```
private static void ThrowCustomExceptionA()
{
    try
    {
        Console.WriteLine("throw new TickerListNotFoundException()");
        throw new TickerListNotFoundException();
    }
    catch (Exception tlnfex)
    {
        Console.WriteLine(tlnfex.Message);
    }
}
```

The `ThrowCustomExceptionA()` method throws a new `TickerListNotFoundException` by using the default constructor. When you run the code, the message that's printed to the console window informs the user that a `CH05_CustomExceptions.TickerListNotFoundException` has been thrown:

```
private static void ThrowCustomExceptionB()
{
    try
    {
        Console.WriteLine("throw new
        TickerListNotFoundException(Message)");
        throw new TickerListNotFoundException("Ticker list not found.");
    }
    catch (Exception tlnfex)
    {
        Console.WriteLine(tlnfex.Message);
    }
}
```

ThrowCustomExceptionB() throws a new `TickerListNotFoundException` by using the constructor that accepts a text message. In this case, the end user is informed that the ticker list hasn't been found:

```
private static void ThrowCustomExceptionC()
{
    try
    {
        Console.WriteLine("throw new TickerListNotFoundException(Message,
            InnerException);");
        throw new TickerListNotFoundException(
            "Ticker list not found for this exchange.",
            new FileNotFoundException(
                "Ticker list file not found.",
                @"F:\TickerFiles\LSE\AimTickerList.json"
            )
        );
    }
    catch (Exception tlnfex)
    {
        Console.WriteLine($"{tlnfex.Message}\n{tlnfex.InnerException}");
    }
}
```

Finally, the `ThrowCustomExceptionC()` method throws a `TickerListNotFoundException` by using the constructor that takes a text message and inner exception. In our example, we provide a meaningful message stating that the ticker list has not been found for this exchange. The inner `FileNotFoundException` expands upon this by providing the name of the specific file that was not found, which happens to be the ticker list of Aim companies on the **London Stock Exchange (LSE)**.

Here, we can see that there are genuine advantages to creating your own custom exceptions. But in most cases, using the intrinsic exceptions within .NET Framework should suffice. The main benefit of custom exceptions is that they're more meaningful exceptions that aid with debugging and resolution.

Here is a brief list of C# exception handling best practices:

- Use try/catch/finally blocks to recover from errors or release resources.
- Handle common conditions without throwing exceptions.
- Design classes so that exceptions can be avoided.
- Throw exceptions instead of returning an error code.
- Use the predefined .NET exception types.
- End exception class names with the word **Exception**.

- Include three constructors in custom exception classes.
- Ensure that exception data is available when code executes remotely.
- Use grammatically correct error messages.
- Include a localized string message in every exception.
- In custom exceptions, provide additional properties as needed.
- Place throw statements so that the stack trace will be helpful.
- Use exception builder methods.
- Restore state when methods don't complete due to exceptions.

Now, it's time to summarize what we have learned in regard to exception handling.

Summary

In this chapter, you learned about checked exceptions and unchecked exceptions. Checked exceptions prevent arithmetic overflow conditions from entering any production code as they are trapped at compile time. Unchecked exceptions go unchecked at compile time and can often make it into production code. This can lead to some *hard-to-track-down* bugs in your code through unexpected data values and even result in exceptions being thrown that cause your programs to crash.

You then learned about the common `NullPointerException` and how to validate parameters that have been passed in using custom `Attribute` and `Validator` classes, which are placed at the top of your methods. These allow you to provide meaningful feedback when validation fails. This leads to more robust programs in the long run.

Then, we discussed using **BREs** to control program flow. You were shown how to control the program flow by expecting exceptional output. Then, you saw how to achieve better control over the flow of computer code without using exceptions by using conditional checks.

The discussion then moved onto the importance of providing meaningful exception messages and how this can be achieved; that is, by writing your own custom exceptions that inherit from the `Exception` class and implement the required three parameters. Through the examples provided, you learned how to use your custom exceptions and how they aid better debugging and resolution.

So, now, it is time to put what you've learned to the test by answering some questions. There is also further reading for you to do if you wish to expand upon what you have learned in this chapter.

In the next chapter, we will be looking at unit testing and how to write your tests first so that they fail. Then, we will write just enough code for the tests to pass and refactor the working code before moving on to the next unit test.

Questions

1. What is a checked exception?
2. What is an unchecked exception?
3. What is an arithmetic overflow exception?
4. What is a `NullPointerException`?
5. How can you validate null parameters to improve your overall code?
6. What does BRE stand for?
7. Are BREs good or bad practice, and why do you think that?
8. What is the alternative to BREs, is it good or bad, and why do you think that?
9. How can you provide meaningful exception messages?
10. What are the requirements for writing your own custom exceptions?

Further reading

- <https://docs.microsoft.com/en-us/dotnet/standard/exceptions/>: This is the official documentation for handling and throwing exceptions in .NET.
- <https://reflectoring.io/business-exceptions/>: The author of this article provides five reasons why BREs are a bad idea after originally believing they were a good idea. There is extra information in this article that was not covered in this chapter.
- <https://docs.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions>: The best practices from Microsoft in regard to C# exception handling, along with code examples and explanations.

6

Unit Testing

Previously, we looked at exception handling, how to implement it properly, and how this can be useful to the customer and the programmer when issues occur. In this chapter, we will look at how programmers can implement their own **quality assurance (QA)** to provide quality code that is robust and less likely to generate exceptions in production.

We start by looking at why we should test our own code, and what makes a good test. We then look at several testing tools that are available to C# programmers. Then, we move on to the three pillars of unit testing that are *Fail, Pass, and Refactor*. Finally, we look at redundant unit tests and why they should be removed.

In this chapter, we will cover the following topics:

- Understanding the reasons for a good test
- Understanding the testing tools
- TDD methodology practice – fail, pass, and refactor
- Removing redundant tests, comments, and dead code

By the end of this chapter, you will have gained the following skills:

- Be able to describe the benefits of good code
- Be able to describe potential negatives that can arise from not unit testing
- Be able to install and use MSTest to write and run unit tests
- Be able to install and use NUnit to write and run unit tests
- Be able to install and use Moq to write fake (mock) objects
- Be able to install and use SpecFlow to write software that adheres to customer specifications
- Be able to write tests that fail, then make them pass, and then perform any necessary refactoring

Technical Requirements

To access the code files of this chapter, you can visit this link: <https://github.com/PacktPublishing/Clean-Code-in-C-/tree/master/CH06>.

Understanding the reasons for a good test

As a programmer, it is nice to work on a new development project that you find interesting, especially if you are highly motivated to do so. But it can be extremely frustrating if you get called away to work on a bug instead. It can be worse if it is not your code, and you don't have the full understanding behind the code. It is even worse still if it is your own code and you have that *"What was I thinking?"* moment! The more you get called away from new development to perform maintenance on existing code, the more you begin to appreciate the need for unit testing. As this appreciation grows, you begin to see the real benefits of learning testing methodologies and techniques such as **Test-Driven Development (TDD)** and **Behavioral-Driven Development (BDD)**.

When you've spent a period of time working as a maintenance programmer on other people's code, you get to see the good, the bad, and the ugly. Such code can be a positive education that opens your eyes to a better way of programming by understanding what not to do and why not to do it. The bad code can make you shout *No. Just no!* and the ugly code can cause your eyes to bleed and your mind to go numb.

Dealing directly with customers, providing them with technical support, you see just how crucial a good customer experience is to the success of the business. Conversely, you also get to see how a bad customer experience can lead to some very frustrated, angry, and extremely foul-mouthed customers; and how quickly sales can be lost due to customer refunds and loss of customers because of very harmful customer rants on social media and review sites.

As a tech lead, it is your responsibility to perform technical code reviews to ensure that staff adhere to the company's coding guidelines and policies, triage bugs, and assist the project manager in managing the people you are responsible for leading. It is important as a tech lead to be good at high-level project management, requirements gathering and analysis, architectural design, and clean programming. You also need to have good people skills.

Your project manager is only interested in delivering a project on time and to budget according to the needs of the business. They really don't care about how you code the software, only that you get it done on time and to the agreed budget. Most importantly, they care that the released software exactly matches what the business asks for – no more and no less – and that the software is to a very high and professional standard, as the quality of the code can equally boost or destroy a company brand. When a project manager is harsh with you, you know the business is putting them under increased pressure. And so that pressure trickles down to you.

As a tech lead, you are sandwiched between the project manager and the team working on the project. In your everyday work, you will be running scrum meetings and dealing with problems. Those problems may be the coders needing resources from the analysts, testers waiting for bugs to be fixed by the developers, and so on. But the most difficult job will be to perform peer code reviews and provide constructive feedback that gets the desired results without offending people. That is why you should take clean coding very seriously, because if you criticize a person's code, you open yourself up for a backlash if your own code is not up to scratch. But also, you will be the one to get it in the neck from the project manager if the software fails testing or goes out with loads of bugs.

That is why, as a tech lead, it is a good idea for you to encourage TDD. The best way to do that is by *leading by example*. Now I know that even degree-educated and experienced programmers can be very stand-offish to TDD. One of the most common reasons is that it can be hard to learn and put into practice, and appear to be more time consuming, especially when code becomes more complex. I have experienced these kinds of objections from my colleagues who prefer not to unit test.

But as a programmer, if you want to be truly confident (such that once you've written a piece of code, you can be confident in its quality and that it will not be returned to you to fix your own bugs), then TDD is a fantastic way to up your game as a programmer. When you learn to test first before you start programming, it soon becomes *habitual*. Such a habit, as a programmer, is very useful and beneficial to you, especially when there comes a time to find a new position, as many employment opportunities advertise for people with TDD or BDD experience.

Another thing to consider while writing code is that bugs in a simple, non-critical note-taking app are not the end of the world. But what if you work in the defense or health sectors? Consider a weapon of mass destruction that has been programmed to go in a specific direction to hit a specific target in enemy territory, but something goes wrong, and the missile aims for civilian populations that belong to your allies. Or, consider what would happen if you had a loved one that was on critical life support that died because of a bug in the software of the medical equipment that was your own fault. Then, what about some safety software going wrong on a passenger jet flying over a populated area that causes the plane to crash into the ground, killing people on the plane and on the ground?

The more critical the software, the more the use of unit testing techniques (such as TDD and BDD) needs to be taken seriously. We will be discussing BDD and TDD tools later in this chapter. When writing software, think about how you would be affected if you were the customer and something went wrong with the code you are writing. How would it affect your family, friends, and colleagues? Also, think of the moral and legal implications if you were responsible for a critical failure.

It is important to understand why, as a programmer, you should learn to test your own code. It is true what they say that *"programmers should never test their own code"*. But it is only true in the context where the code is finished and ready for testing before it goes into production. So while the code is still being programmed, programmers should always be testing their own code. Yet some businesses are so time-constrained that proper QA is often sacrificed so that the business can be the first to market.

It may be very important for a business to be the first to market, but first impressions count. If a business is first to market, and the product has some serious flaws that become globally broadcast, this can have a long-lasting negative impact on a business. So you must think very carefully as a programmer and do your best to ensure that if the software has flaws, you are not the one responsible. When things go wrong in a business, heads will roll. And in Teflon Management, the managers will pass the guilt for driving ridiculous deadlines from themselves all the way down to the programmers that had to meet the deadline and make sacrifices to do so.

So you see, it is very important as a programmer that you test your code and test it often, especially prior to releasing it to the testing team. That is why you are actively encouraged to transition into the mindset and habitual behavior of writing your tests first, based upon the specification that you currently implementing. Your tests should fail to start with. You then write only enough code to get the tests to pass, and then you refactor your code as you need to.

It is hard to get started with TDD or BDD. But once you get the hang of it, TDD and BDD become second nature. And you will probably find that in the long term, you are left with cleaner code that is easy to read and maintain. You may also find that your confidence in your ability to modify the code without breaking it may also be greatly improved. Obviously, there is more code in the sense that you have the production method and the test method(s). But you may actually end up writing less code overall, as you will not be adding extra code that you think may be needed!

Picture yourself at your computer with a software specification that you have to translate into working software. A bad habit that many programmers have, and that I've been guilty of in the past, is that they jump straight into coding without doing any real design work. In my experience, this actually prolongs the time it takes to develop a piece of code and can often lead to more bugs and code that is hard to maintain and extend. In fact, although it appears to be counter-intuitive to some programmers, proper planning and design actually speed up coding, especially when you factor in maintenance and extensions.

This is where the test team comes in. Before we go any further, let's describe use cases, test designs, test cases, and test suites, and how they relate to one another.

A use case explains the process flow for a single operation, such as adding a customer record. A test design will comprise one or more test cases that test for different scenarios that could take place for the single use case. The test cases may be carried out manually, or they may be automated tests that are executed by a test suite. A test suite is a piece of software used to discover and run tests and to report their outcomes to an end user. The writing of use cases will be the role of the business analyst. As for the test design, test cases, and test suite, these will be the responsibility of the dedicated test team. Developers need not be concerned with putting together the use cases, test designs of test cases, and their execution in the test suite. Developers must focus on writing and using their unit tests to write code that fails, then runs, and is then refactored as necessary.

Software testers collaborate with programmers. This collaboration normally starts at project inception, and continues right through to the end. Both the development team and testing team will collaborate by sharing test cases for each product backlog item. This process normally consists of writing test cases. For the tests to pass, they will have to meet test criteria. These test cases will normally be run using a combination of manual testing and some test suite automation.

During the development phases, the testers write their QA tests and the developers write their unit tests. When developers submit their code to the test team, the test team will run through their battery of tests. The outcome of those tests will be fed back to the developers and the project stakeholders. If problems are encountered, this is known as technical debt. The development team will have to factor in time to address the issues raised by the test team. When the test team confirms that the software has been completed to the required level of quality, then the code is passed on to infrastructure to release into production.

Assuming we are starting a brand new project (also known as a greenfield project), we would select the appropriate project type and tick the option to include a test project. This would create a solution that consists of our main project and the test project.

The type of project that we create and any features of projects to be implemented will be dependent upon use cases. Use cases are used during system analysis to identify, confirm, and organize software requirements. From use cases, test cases can be assigned to the acceptance criteria. As a programmer, you can take these use cases and their test cases to build up your own unit tests for each test case. Your tests are then run as part of a test suite. In Visual Studio 2019, you can access the **Test Explorer** from the **View | Test Explorer** menu. When you build your project, tests will be discovered. When tests are discovered, they are viewed in the **Test Explorer**. You can then run and/or debug your tests in the **Test Explorer**.

It is worth noting at this stage that it will be the responsibility of the testers and not the developers to design tests and come up with a suitable number of test cases. They are also responsible for QA once the software leaves the hands of the developers. But it is still the responsibility of the developer to unit test their code, and this is where test cases can be a real help and motivation for writing unit tests in your code.

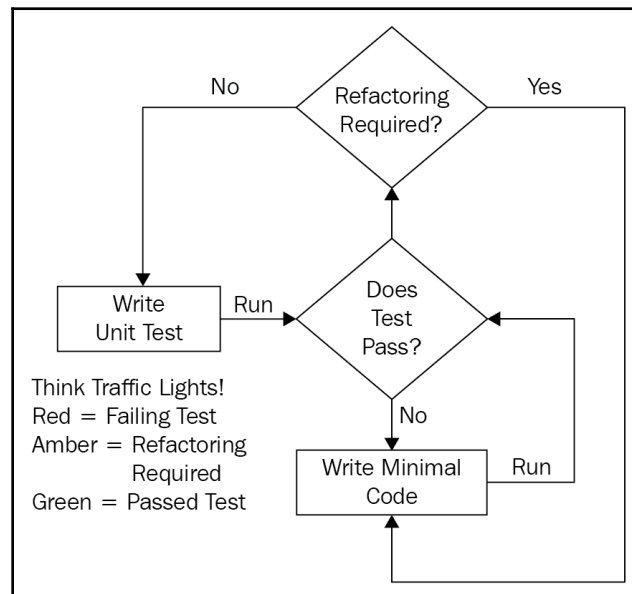
When the solution is created, the first thing you do is open the provided test class. In that test class, you write the pseudocode for what you must accomplish. You then go step by step through the pseudocode and add your test methods that test each step that must be accomplished in order to reach your goal of a completed software project. Each test method that you write is written to fail. You then write just enough code to pass the test. Then, once the test passes, you refactor your code before progressing to the next test. So, you can see that unit testing is not rocket science. But what does it take to write a good unit test?

Any code that is under test will be expected to provide a specific function. A function takes in input and produces output.

In a normally functioning computer program, a method (or function) will have an *acceptable* range of inputs and outputs, and an *unacceptable* range of inputs and outputs. And so the perfect unit test will test the lowest acceptable value, the highest acceptable value, and will provide test cases that are outside of the acceptable range of values both high and low.

Unit tests must be atomic, which means that they should only test one thing. Since methods can be chained together in the same class and even across multiple classes in multiple assemblies, it is often useful to provide fake or mock objects for the classes under test to keep them atomic. The output must determine whether it passes or fails. Good unit tests must never be inconclusive.

The result of a test should be repeatable, in that it either always passes or always fails in given conditions. That is, the same test run over and over again should not have different outcomes each time it is run. If it does, then it is not repeatable. Unit tests should not have to rely on other tests being run before them, and they should be isolated from other methods and classes. You should also aim for unit tests that run in milliseconds. Any test that takes one second or more to run is taking too long. If code takes longer than a second, then you should consider refactoring or implementing a mock object for testing. And since we are busy programmers, unit tests should be easy to set up and not require a lot of coding or configuration. The following diagram shows the unit testing life cycle:



We'll be writing unit tests and mock objects during this chapter. But before we do, we'll need to look at some of the tools that are available to us as C# programmers.

Understanding the testing tools

The testing tools we'll be looking at within Visual Studio are **MSTest**, **NUnit**, **Moq**, and **SpecFlow**. Each testing tool creates a console application and the relevant test project. NUnit and MSTest are unit testing frameworks. NUnit is much older than MSTest, and so has a more mature and full-featured API compared to MSTest. I personally prefer NUnit over MSTest.

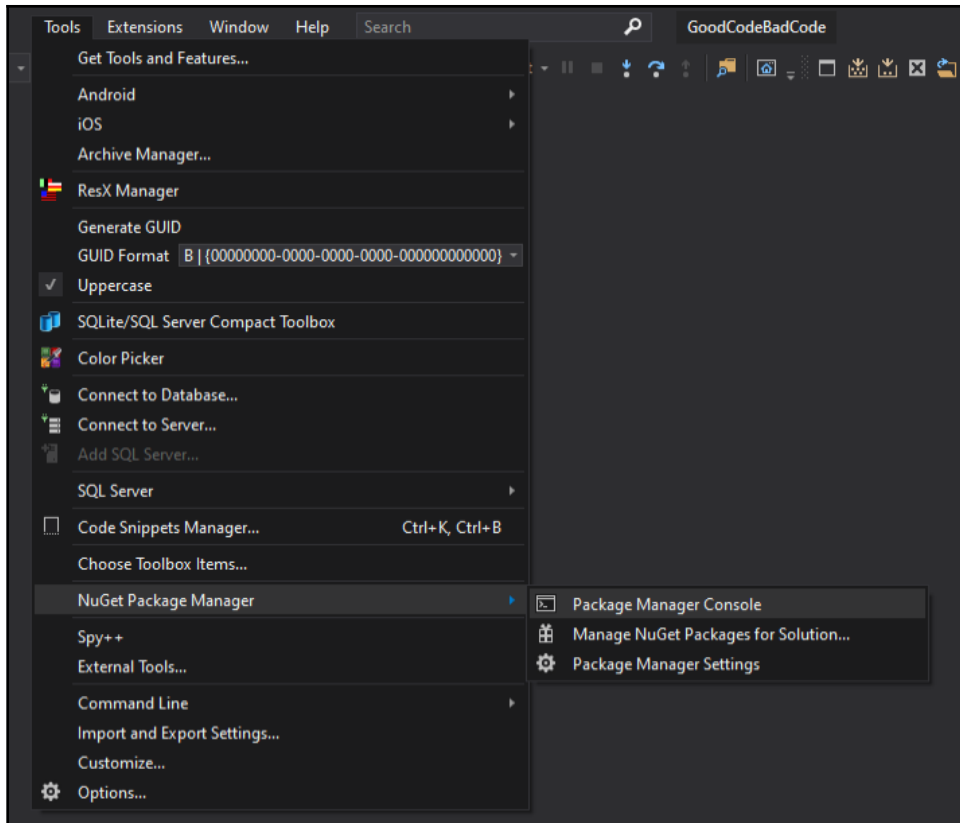
Moq is different from MSTest and NUnit as it is not a testing framework but a mocking framework. A mocking framework replaces the real classes in your project with mock (fake) implementations that are used for testing purposes. You can use Moq together with MSTest or NUnit. And finally, SpecFlow is a BDD framework. You start by writing a feature in a feature file using business language that the user and the techy alike will understand. Then a step file is generated for that feature. The step file contains the methods as steps necessary to implement that feature.

By the end of this chapter, you will understand what each tool does and will be able to use them in your own projects. So, let's get started by looking at MSTest.

MSTest

In this section, we will install and configure the MSTest Framework. We will write a test class with test methods and initialize it. We will perform assembly setup and cleanup, class cleanup, and method cleanup, and perform assertions.

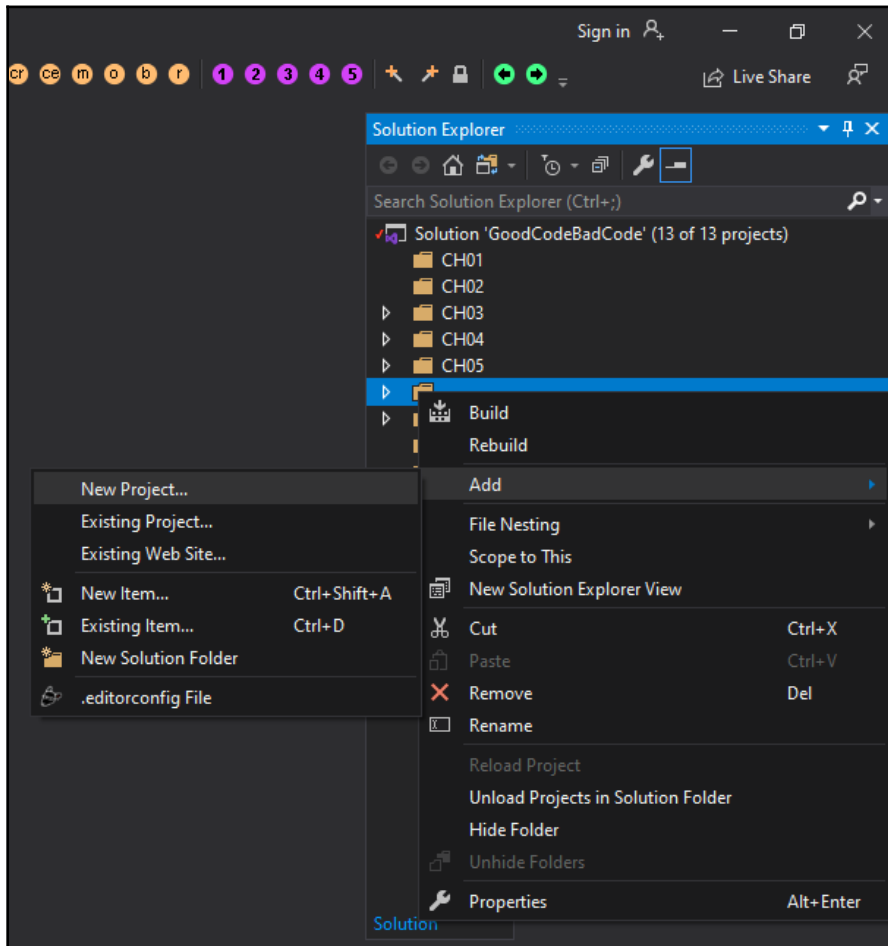
To install the MSTest Framework from the command line in Visual Studio, you will need to open the **Package Manager Console** via **Tools | NuGet Package Manager | Package Manager Console**:



Then, run the following three commands to install the MSTest Framework:

```
install-package mstest.testframework  
install-package mstest.testadapter  
install-package microsoft.net.tests.sdk
```

Alternatively, you can add a new project and select **Unit Test Project (.NET Framework)** from the **Context | Add** menu in the **Solution Explorer**. See the screenshot that follows. When naming test projects, the accepted standard is in the form of `<ProjectName>.Tests`. This helps to associate them with the tests and distinguish them from the project that is under test:



The following code is the default unit test code that is generated when you add an MSTest project to your solution. As you can see, the class imports the `Microsoft.VisualStudio.TestTools.UnitTesting` namespace.

The `[TestClass]` attribute identifies to the MS Test Framework that this class is a test class. The `[TestMethod]` attribute marks the method as a test method. All classes that have the `[TestMethod]` attribute will appear in the test player.

The `[TestClass]` and `[TestMethod]` attributes are mandatory:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace CH05_MSTestUnitTesting.Tests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

There are other methods and attributes that can optionally be combined to produce a complete test execution workflow. These include `[AssemblyInitialize]`, `[AssemblyCleanup]`, `[ClassInitialize]`, `[ClassCleanup]`, `[TestInitialize]`, and `[TestCleanup]`. As their names imply, the initialization attributes are used to perform any initialization at the assembly, class, and method level prior to tests being run. Likewise, the cleanup attributes run at the method, class, and assembly level after tests have been run to perform any necessary cleanup operations. We will look at each in turn and add them to your project as we will see its order of execution when we run the final code.

The `WriteSeparatorLine()` method is a helper method for the purpose of separating our testing method outputs. This will help us to more easily follow what's going on with our test class:

```
private static void WriteSeparatorLine()
{
    Debug.WriteLine("-----");
}
```

Optionally, assign the `[AssemblyInitialize]` attribute to execute code before the tests are executed:

```
[AssemblyInitialize]
public static void AssemblyInit(TestContext context)
```

```
{
    WriteSeparatorLine();
    Debug.WriteLine("Optional: AssemblyInitialize");
    Debug.WriteLine("Executes once before the test run.");
}
```

Then, you can optionally assign the `[ClassInitialize]` attribute to execute code once before the tests are executed:

```
[ClassInitialize]
public static void TestFixtureSetup(TestContext context)
{
    WriteSeparatorLine();
    Console.WriteLine("Optional: ClassInitialize");
    Console.WriteLine("Executes once for the test class.");
}
```

Then, run the setup code before each unit test by assigning the `[TestInitialize]` attribute to a setup method:

```
[TestInitialize]
public void Setup()
{
    WriteSeparatorLine();
    Debug.WriteLine("Optional: TestInitialize");
    Debug.WriteLine("Runs before each test.");
}
```

When you have finished your test run, you can optionally assign the `[AssemblyCleanup]` attribute to perform any necessary cleanup operations:

```
[AssemblyCleanup]
public static void AssemblyCleanup()
{
    WriteSeparatorLine();
    Debug.WriteLine("Optional: AssemblyCleanup");
    Debug.WriteLine("Executes once after the test run.");
}
```

The optional method marked as `[ClassCleanup]` runs once after all tests in the class have been executed. You cannot guarantee when this method will run, as it may not run immediately after the execution of all tests:

```
[ClassCleanup]
public static void TestFixtureTearDown()
{
    WriteSeparatorLine();
    Debug.WriteLine("Optional: ClassCleanup");
}
```

```

        Debug.WriteLine("Runs once after all tests in the class have been
        executed.");
        Debug.WriteLine("Not guaranteed that it executes instantly after all
        tests the class have executed.");
    }

```

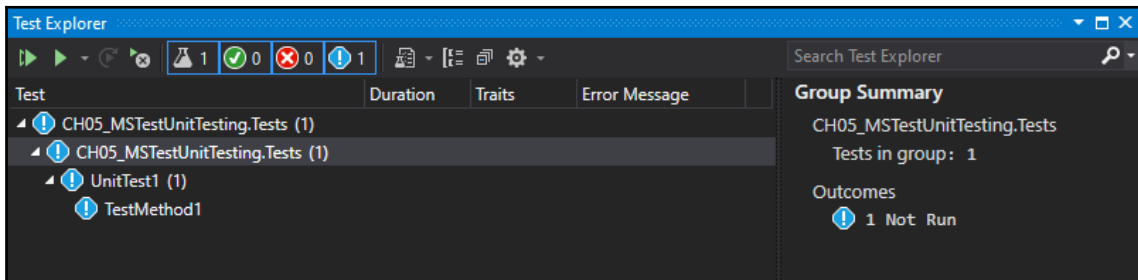
To perform clean up operations after each test has been run, apply the `[TestCleanup]` attribute to the test cleanup method:

```

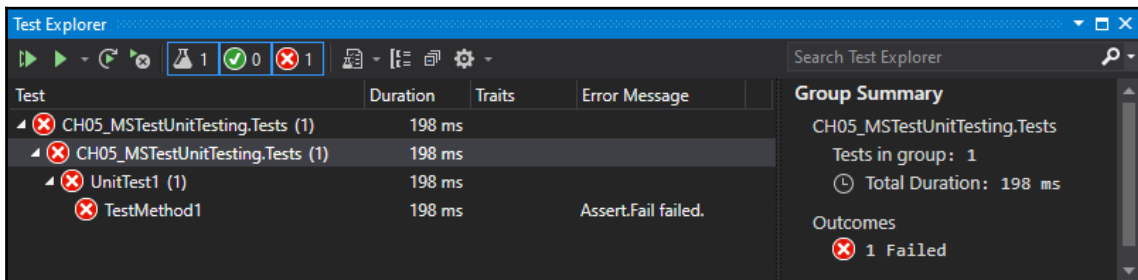
[TestCleanup]
public void TearDown()
{
    WriteSeparatorLine();
    Debug.WriteLine("Optional: TestCleanup");
    Debug.WriteLine("Runs after each test.");
    Assert.Fail();
}

```

Now that our code is in place, build it. Then, from the **Test** menu, select **Test Explorer**. You should see the following test in the **Test Explorer**. As you can from the following screenshot, the test has not yet been run:



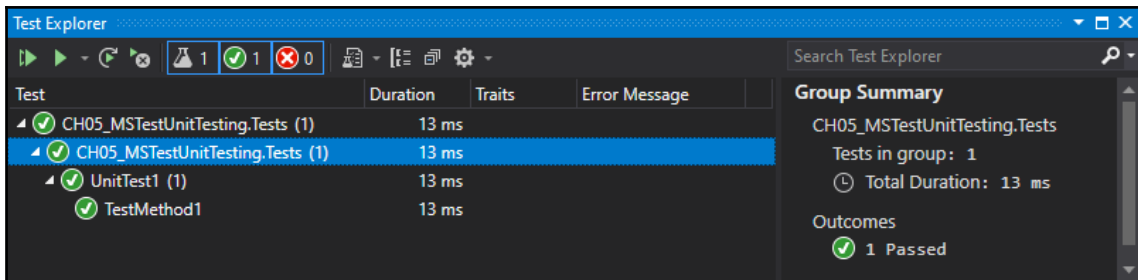
So, let's run our only test. Oh no! Our test has failed, as shown in the following screenshot:



Update the `TestMethod1()` code as shown in the following snippet, and then run the test again:

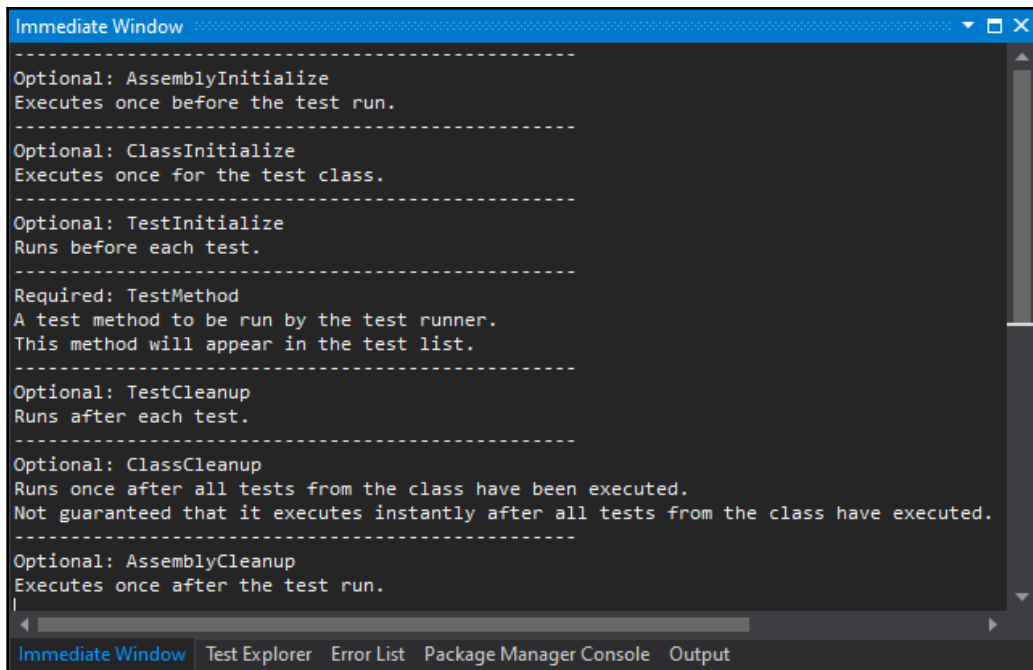
```
[TestMethod]
public void TestMethod1()
{
    WriteSeparatorLine();
    Debug.WriteLine("Required: TestMethod");
    Debug.WriteLine("A test method to be run by the test runner.");
    Debug.WriteLine("This method will appear in the test list.");
    Assert.IsTrue(true);
}
```

You see that the test has passed in the **Test Explorer**, as shown in the screenshot that follows:



So, from the previous screenshots, you can see that tests that have not been executed are *blue*, tests that fail are *red*, and tests that pass are *green*. From **Tools | Options | Debugging | General**, select **Redirect all Output Window text to the Immediate Window**. Then, select **Run | Debug All Tests**.

As you run through the tests and the output is printed to **Immediate Window**, it will become apparent in what order the attributes are being executed. The following screenshot shows the output from our test methods:



As you have seen already, we have used two `Assert` methods—these being `Assert.Fail()` and `Assert.IsTrue(true)`. The `Assert` class is very useful and so it pays to be aware of the methods available in the class for unit testing. These available methods are listed and described as follows:

Methods	Description
<code>Assert.AreEqual()</code>	Tests whether the specified values are equal and throws an exception if the two values are not equal.
<code>Assert.AreNotEqual()</code>	Tests whether the specified values are unequal and throws an exception if the two values are equal.
<code>Assert.AreNotSame()</code>	Tests whether the specified objects refer to different objects and throws an exception if the two inputs refer to the same object.
<code>Assert.AreSame()</code>	Tests whether the specified objects both refer to the same object and throws an exception if the two inputs do not refer to the same object.
<code>Assert.Equals()</code>	This object will always throw with <code>Assert.Fail</code> . Hence, we can use <code>Assert.AreEqual</code> instead.
<code>Assert.Fail()</code>	Throws an <code>AssertFailedException</code> exception.
<code>Assert.Inconclusive()</code>	Throws an <code>AssertInconclusiveException</code> exception.

<code>Assert.IsFalse()</code>	Tests whether the specified condition is false and throws an exception if the condition is true.
<code>Assert.IsInstanceOfType()</code>	Tests whether the specified object is an instance of the expected type and throws an exception if the expected type is not in the inheritance hierarchy of the object.
<code>Assert.IsNotInstanceOfType()</code>	Tests whether the specified object is an instance of the wrong type and throws an exception if the specified type is in the inheritance hierarchy of the object.
<code>Assert.IsNotNull()</code>	Tests whether the specified object is non-null and throws an exception if it is null.
<code>Assert.IsNull()</code>	Tests whether the specified object is null and throws an exception if it is not null.
<code>Assert.IsTrue()</code>	Tests whether the specified condition is true and throws an exception if the condition is false.
<code>Assert.ReferenceEquals()</code>	Determines whether the specified object instances are the same instance.
<code>Assert.ReplaceNullChars()</code>	Replaces null characters (' <code>\0</code> ') with " <code>\\0</code> ".
<code>Assert.That()</code>	Gets the singleton instance of the <code>Assert</code> functionality.
<code>Assert.ThrowsException()</code>	Tests whether the code specified by delegate action throws given an exception of type <code>T</code> (and not a derived type) and throws <code>AssertFailedException</code> if the code does not throw an exception, or throws an exception of a type other than <code>T</code> . In simple words, this takes a delegate and asserts that it throws the expected exception with the expected message.
<code>Assert.ThrowsExceptionAsync()</code>	Tests whether the code specified by delegate action throws given the exception of type <code>T</code> (and not a derived type) and throws <code>AssertFailedException</code> if the code does not throw an exception, or throws an exception of a type other than <code>T</code> .

Now that we have had a look at `MSTest`, it is time to look at `NUnit`.

NUnit

If `NUnit` is not installed for Visual Studio, then download and install it via **Extensions | Manage Extensions**. After that, create a new `NUnit Test Project (.NET Core)`. The following code contains the default class created by `NUnit`, called `Tests`:

```
public class Tests
{
    [SetUp]
    public void Setup()
```



```

    {
    }

    [Test]
    public void Test1()
    {
        Assert.Pass();
    }
}

```

As you can see from the `Test1` method, the test methods also use an `Assert` class, as does MSTest for testing assertions in code. The NUnit `Assert` class makes the following methods available to us (note that methods marked as [NUnit] in the following table are specific to NUnit; all others are also present in MSTest):

Methods	Description
<code>Assert.AreEqual()</code>	Verifies that two items are equal. If they are not equal, then an exception is thrown.
<code>Assert.AreNotEqual()</code>	Verifies that two items are not equal. If they are equal, then an exception is thrown.
<code>Assert.AreNotSame()</code>	Verifies that two objects do not refer to the same object. If they do, then an exception is thrown.
<code>Assert.AreSame()</code>	Verifies that two objects refer to the same object. If they don't, then an exception is thrown.
<code>Assert.ByVal()</code>	[NUnit] Applies a constraint to an actual value, succeeding if the constraint is satisfied and throwing an assertion exception on failure. Used as a synonym for <code>That</code> in rare cases where a private setter causes a Visual Basic compilation error.
<code>Assert.Catch()</code>	[NUnit] Verifies that a delegate throws an exception when called and returns it.
<code>Assert.Contains()</code>	[NUnit] Verifies whether a value is contained in a collection.
<code>Assert.DoesNotThrow()</code>	[NUnit] Verifies that a method does not throw an exception.
<code>Assert.Equal()</code>	[NUnit] Do not use. Use <code>Assert.AreEqual()</code> instead.
<code>Assert.Fail()</code>	Throws an <code>AssertionException</code> .
<code>Assert.False()</code>	[NUnit] Verifies a condition is false. Throws an exception if the condition is true.
<code>Assert.Greater()</code>	[NUnit] Verifies that the first value is greater than the second value. Throws an exception if it is not.
<code>Assert.GreaterOrEqual()</code>	[NUnit] Verifies that the first value is greater than or equal to the second value. Throws an exception if it is not.
<code>Assert.Ignore()</code>	[NUnit] Throws <code>IgnoreException</code> with the message and arguments that are passed in. This causes the test to be reported as ignored.

<code>Assert.Inconclusive()</code>	Throws <code>InconclusiveException</code> with the message and arguments that are passed in. This causes the test to be reported as inconclusive.
<code>Assert.IsAssignableFrom()</code>	[NUnit] Verifies that an object may be assigned a value of a given type.
<code>Assert.IsEmpty()</code>	[NUnit] Verifies whether a value such as a string or collection is empty.
<code>Assert.IsFalse()</code>	Verifies whether a condition is false. Throws an exception if it is true.
<code>Assert.IsInstanceOf()</code>	[NUnit] Verifies that an object is an instance of a given type.
<code>Assert.NAN()</code>	[NUnit] Verifies that the value is not a number. If it is, then an exception is thrown.
<code>Assert.IsNotAssignableFrom()</code>	[NUnit] Verifies that an object is not assignable from a given type.
<code>Assert.IsNotEmpty()</code>	[NUnit] Verifies that a string or collection is not empty.
<code>Asserts.IsNotInstanceOf()</code>	[NUnit] Verifies that the object is not an instance of a given type.
<code>Assert.IsNotNull()</code>	Verifies that an object is not null. If it is, then an exception is thrown.
<code>Assert.IsNull()</code>	Verifies that an object is null. If it is not, then an exception is thrown.
<code>Assert.IsTrue()</code>	Verifies that a condition is true. If it is false, then an exception is thrown.
<code>Assert.Less()</code>	[NUnit] Verifies that the first value is less than the second value. If not, then an exception is thrown.
<code>Assert.LessOrEqual()</code>	[NUnit] Verifies that the first value is less than or equal to the second value. If not, then an exception is thrown.
<code>Assert.Multiple()</code>	[NUnit] Wraps code containing a series of assertions, which should all be executed, even if they fail. Failed results are saved and reported at the end of the code block.
<code>Assert.Negative()</code>	[NUnit] Verifies that a number is negative. If not, then an exception is thrown.
<code>Assert.NotNull()</code>	[NUnit] Verifies that an object is not null. If it is null, then an exception is thrown.
<code>Assert.NotZero()</code>	[NUnit] Verifies that a number is not zero. If it is zero, then an exception is thrown.
<code>Assert.Null()</code>	[NUnit] Verifies that an object is null. If not, then an exception is thrown.
<code>Assert.Pass()</code>	[NUnit] Throws <code>SuccessException</code> with the message and arguments that are passed in. This allows a test to be cut short, with a result of success returned to NUnit.
<code>Assert.Positive()</code>	[NUnit] Verifies that a number is positive.

<code>Assert.ReferenceEquals()</code>	[NUnit] Do not use. Throws <code>InvalidOperationException</code> .
<code>Assert.That()</code>	Verifies that a condition is true. If not, then an exception is thrown.
<code>Assert.Throws()</code>	Verifies that a delegate throws a particular exception when it is called.
<code>Assert.True()</code>	[NUnit] Verifies that a condition is true. If not, then an exception is called.
<code>Assert.Warn()</code>	[NUnit] Issues a warning using the message and arguments provided.
<code>Assert.Zero()</code>	[NUnit] Verifies that a number is zero.

The NUnit life cycle begins with the `TestFixtureSetup` that is executed once before the first test `SetUp`. Then, `SetUp` is executed before each test. After each test has executed, `TearDown` is executed. And finally, `TestFixtureTearDown` is executed once after the last test `TearDown`. We are now going to update the `Tests` class so that we can debug and see the NUnit life cycle in action:

```
using System;
using System.Diagnostics;
using NUnit.Framework;

namespace CH06_NUnitUnitTesting.Tests
{
    [TestFixture]
    public class Tests : IDisposable
    {
        public TestClass()
        {
            WriteSeparatorLine();
            Debug.WriteLine("Constructor");
        }

        public void Dispose()
        {
            WriteSeparatorLine();
            Debug.WriteLine("Dispose");
        }
    }
}
```

We have added the `[TestFixture]` to the class and implemented the `IDisposable` interface. The `[TestFixture]` attribute is optional for non-parameterized and non-generic fixtures. A class will be treated as a `[TestFixture]` as long as at least one method is marked with the `[Test]`, `[TestCase]`, or `[TestCaseSource]` attributes.

The `WriteSeparatorLine()` method acts as a separator for our debug output. This method will be called at the top of all our methods in the `Tests` class:

```
private static void WriteSeparatorLine()
{
    Debug.WriteLine("-----");
}
```

The method marked with the `[OneTimeSetUp]` attribute will only run once before any tests in that class are run. Any initialization that is required for all the different tests would be carried out here:

```
[OneTimeSetUp]
public void OneTimeSetup()
{
    WriteSeparatorLine();
    Debug.WriteLine("OneTimeSetup");
    Debug.WriteLine("This method is run once before any tests in this
        class are run.");
}
```

The method marked with `[OneTimeTearDown]` is run once after all the tests have been run, and before the class is disposed:

```
[OneTimeTearDown]
public void OneTimeTearDown()
{
    WriteSeparatorLine();
    Debug.WriteLine("OneTimeTearDown");
    Debug.WriteLine("This method is run once after all tests in this
        class have been run.");
    Debug.WriteLine("This method runs even when an exception occurs.");
}
```

The method marked with the `[Setup]` attribute runs once before every test method:

```
[Setup]
public void Setup()
{
    WriteSeparatorLine();
    Debug.WriteLine("Setup");
    Debug.WriteLine("This method is run before each test method is run.");
}
```

The method marked with the `[TearDown]` attribute is run once after every test method has completed:

```
[TearDown]
public void Teardown()
{
    WriteSeparatorLine();
    Debug.WriteLine("Teardown");
    Debug.WriteLine("This method is run after each test method
        has been run.");
    Debug.WriteLine("This method runs even when an exception occurs.");
}
```

The `Test2()` method is a test method as denoted by the `[Test]` attribute and will be the second test method to run as determined by the `[Order(1)]` attribute. This method throws `InconclusiveException`:

```
[Test]
[Order(1)]
public void Test2()
{
    WriteSeparatorLine();
    Debug.WriteLine("Test:Test2");
    Debug.WriteLine("Order: 1");
    Assert.Inconclusive("Test 2 is inconclusive.");
}
```

The `Test1()` method is a test method as denoted by the `[Test]` attribute and will be the first test method to be run as determined by the `[Order(0)]` attribute. The method passes `SuccessException`:

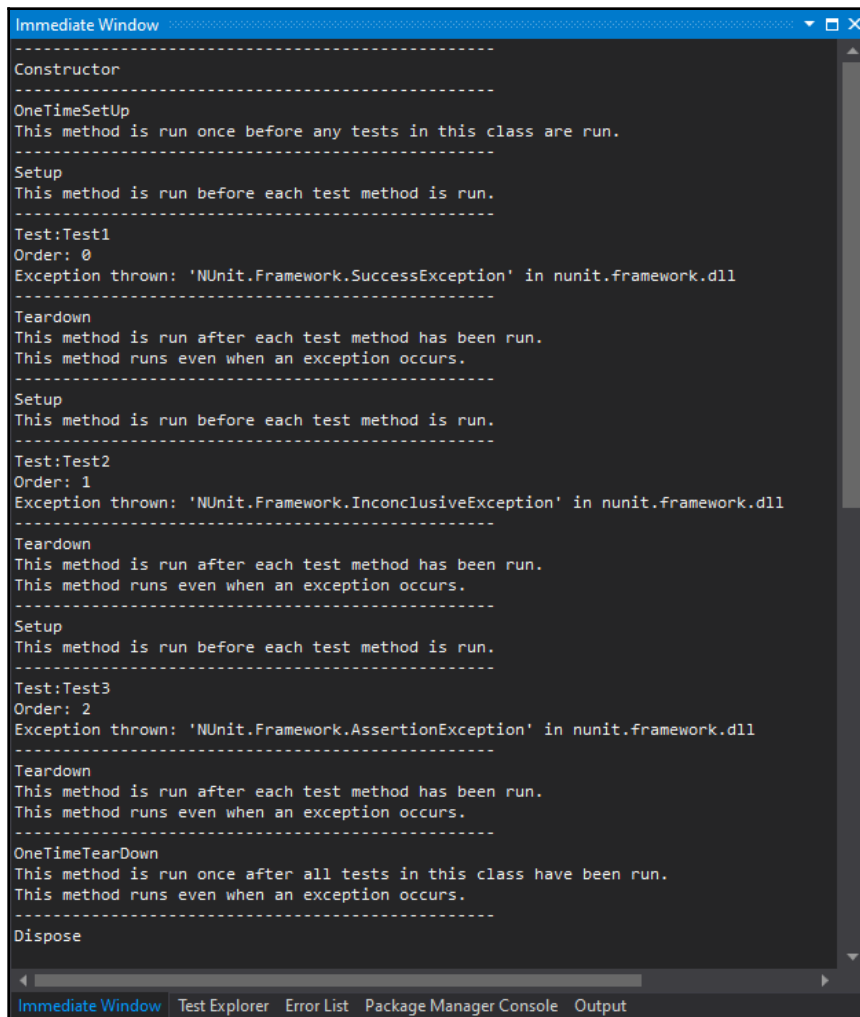
```
[Test]
[Order(0)]
public void Test1()
{
    WriteSeparatorLine();
    Debug.WriteLine("Test:Test1");
    Debug.WriteLine("Order: 0");
    Assert.Pass("Test 1 passed with flying colours.");
}
```

The `Test3()` method is a test method as denoted by the `[Test]` attribute and will be the third test method to run as determined by the `[Order(2)]` attribute. The method throws `AssertionException`:

```
[Test]
[Order(2)]
```

```
public void Test3()
{
    WriteSeparatorLine();
    Debug.WriteLine("Test:Test3");
    Debug.WriteLine("Order: 2");
    Assert.Fail("Test 1 failed dismally.");
}
```

When you debug all the tests, your immediate window should look like the following screenshot:

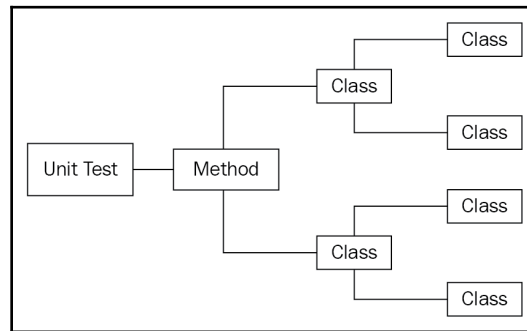


You have now been exposed to MSTest and NUnit, and have seen the testing life cycle for each framework in action. It's now time to have a look at Moq.

As you can see from the table of NUnit methods compared to the table of MSTest methods, NUnit enables more fine-grained unit testing over MSTest and executes with better performance, which is why it is more widely used than MSTest.

Moq

A unit test should only test the method under test. See the following diagram. If a method under test calls other methods that are either in the current class or in different classes, then not only test methods but other methods are also tested:



One way to overcome this is to use mock (fake) objects. The mock object will only test the method you want to test, and you can make the mock object work any way that you want to. If you were to write your own mock objects, you would soon come to appreciate that there is a lot of hard work involved. This may be unacceptable in time-sensitive projects, and the more complex your code becomes, the more complex your mock objects become.

You will inevitably give it up as a bad job, or you will look for a mocking framework that suits your needs. Rhino Mocks and Moq are two mocking frameworks for the .NET Framework. For the purposes of this chapter, we will only be looking at Moq, which is easier to learn and use compared to Rhino Mocks. For more information on Rhino Mocks, visit <http://hibernatingrhinos.com/oss/rhino-mocks>.

When testing using Moq, we start by adding the mock object and then configure the mock object to do something. We then assert that the configuration is working and that the mock was invoked. These steps enable us to determine that the mock is correctly set up. Moq only produces test doubles. It does not test the code. You still need a test framework such as NUnit to test your code.

We'll now look at an example of using Moq and NUnit together.

Create a new console application and call it `CH06_Moq`. Add the following interface and classes—`IFoo`, `Bar`, `Baz`, and `UnitTests`. Then, via the Nuget package manager, install Moq, NUnit, and NUnit3TestAdapter. Update the `Bar` class with the following code:

```
namespace CH06_Moq
{
    public class Bar
    {
        public virtual Baz Baz { get; set; }
        public virtual bool Submit() { return false; }
    }
}
```

The `Bar` class has a virtual property of type `Baz` and a virtual method called `Submit()` that returns a Boolean value of `false`. Now update the `Baz` class as follows:

```
namespace CH06_Moq
{
    public class Baz
    {
        public virtual string Name { get; set; }
    }
}
```

The `Baz` class has a single virtual property of type `string` called `Name`. Modify the `IFoo` file to contain the following source code:

```
namespace CH06_Moq
{
    public interface IFoo
    {
        Bar Bar { get; set; }
        string Name { get; set; }
        int Value { get; set; }
        bool DoSomething(string value);
        bool DoSomething(int number, string value);
        string DoSomethingStringy(string value);
        bool TryParse(string value, out string outputValue);
        bool Submit(ref Bar bar);
        int GetCount();
        bool Add(int value);
    }
}
```


The `IFoo` interface has a number of properties and methods. As you can see, the interface has a reference to the `Bar` class, and we know that the `Bar` class contains a reference to the `Baz` class. We will now start updating our `UnitTests` class to test our newly-created interface and classes using `NUnit` and `Moq`. Modify the `UnitTests` class file so that it looks like the code that follows:

```
using Moq;
using NUnit.Framework;
using System;

namespace CH06_Moq
{
    [TestFixture]
    public class UnitTests
    {
    }
}
```

Now, add the `AssertThrows` method that asserts whether a designated exception has been thrown or not:

```
public bool AssertThrows<TException>(
    Action action,
    Func<TException, bool> exceptionCondition = null
) where TException : Exception
{
    try
    {
        action();
    }
    catch (TException ex)
    {
        if (exceptionCondition != null)
        {
            return exceptionCondition(ex);
        }
        return true;
    }
    catch
    {
        return false;
    }
    return false;
}
```

The `Assert.Throws` method is a generic method that will return `true` if your method throws the designated exception, and `false` if it does not. We will be using this method when we test exceptions further in this chapter. Now, add the `DoSomethingReturnsTrue()` method:

```
[Test]
public void DoSomethingReturnsTrue()
{
    var mock = new Mock<IFoo>();
    mock.Setup(foo => foo.DoSomething("ping")).Returns(true);
    Assert.IsTrue(mock.Object.DoSomething("ping"));
}
```

The `DoSomethingReturnsTrue()` method creates a new mock implementation of the `IFoo` interface. Then it sets up the `DoSomething()` method to accept a string containing the word "ping", and then returns `true`. Finally, the method asserts that when the `DoSomething()` method is called with the text "ping", the method returns a value of `true`. We'll now implement a similar test method that returns `false` if the value is "tracert":

```
[Test]
public void DoSomethingReturnsFalse()
{
    var mock = new Mock<IFoo>();
    mock.Setup(foo => foo.DoSomething("tracert")).Returns(false);
    Assert.IsFalse(mock.Object.DoSomething("tracert"));
}
```

The `DoSomethingReturnsFalse()` method follows the same procedure as the `DoSomethingReturnsTrue()` method. We create a mock object of the `IFoo` interface, set it up to return `false` if the parameter value is "tracert", and then assert that `false` is returned for a parameter value of "tracert". Next, we'll test our arguments:

```
[Test]
public void OutArguments()
{
    var mock = new Mock<IFoo>();
    var outString = "ack";
    mock.Setup(foo => foo.TryParse("ping", out outString)).Returns(true);
    Assert.AreEqual("ack", outString);
    Assert.IsTrue(mock.Object.TryParse("ping", out outString));
}
```

The `OutArguments()` method creates an implementation of the `IFoo` interface. A string that will be used as an out parameter is then declared and assigned the value "ack". Next, the `TryParse()` method of the `IFoo` mock object is set up to return `true` for an input value of "ping" and to output the string value of "ack". We then assert that the `outString` is equal to the value "ack". The final check asserts that `TryParse()` returns `true` for the input value of "ping":

```
[Test]
public void RefArguments()
{
    var instance = new Bar();
    var mock = new Mock<IFoo>();
    mock.Setup(foo => foo.Submit(ref instance)).Returns(true);
    Assert.AreEqual(true, mock.Object.Submit(ref instance));
}
```

The `RefArguments()` method creates an instance of the `Bar` class. Then, a mock implementation of the `IFoo` interface is created. The `Submit()` method is then set up to return `true` if the reference type passed in is of type `Bar`. We then assert that the argument that is passed in is `true` of type `Bar`. In our `AccessInvocationArguments()` test method, we create a new implementation of the `IFoo` interface:

```
[Test]
public void AccessInvocationArguments()
{
    var mock = new Mock<IFoo>();
    mock.Setup(foo => foo.DoSomethingStringy(It.IsAny<string>()))
        .Returns((string s) => s.ToLower());
    Assert.AreEqual("i like oranges!", mock.Object.DoSomethingStringy("I
LIKE ORANGES!"));
}
```

Then we set up the `DoSomethingStringy()` method to convert the input to lowercase and return it. Finally, we assert that the string returned is the string passed in that has been converted to lowercase:

```
[Test]
public void ThrowingWhenInvokedWithSpecificParameters()
{
    var mock = new Mock<IFoo>();
    mock.Setup(foo => foo.DoSomething("reset"))
        .Throws<InvalidOperationException>();
    mock.Setup(foo => foo.DoSomething(""))
        .Throws(new ArgumentException("command"));
    Assert.IsTrue(
        Assert.Throws<InvalidOperationException>(
```

```
        () => mock.Object.DoSomething("reset")
    )
);
Assert.IsTrue(
    Assert.Throws<ArgumentException>(
        () => mock.Object.DoSomething("")
    )
);
Assert.Throws(
    Is.TypeOf<ArgumentException>()
    .And.Message.EqualTo("command"),
    () => mock.Object.DoSomething("")
);
}
```

In our final test method called `ThrowingWhenInvokedWithSpecificParameters()`, we create a mock implementation of the `IFoo` interface. We then configure the `DoSomething()` method to throw `InvalidOperationException` when the passed-in value is "reset".

An `ArgumentException` exception of "command" is thrown when an empty string is passed in. We then assert that `InvalidOperationException` is thrown when the input value is "reset". When the input value is an empty string, we assert that `ArgumentException` is thrown, with the assertion that the message of `ArgumentException` is "command".

You've now seen how to use a mocking framework called `Moq` to create mock objects to test your code using `NUnit`. The last tool we will now look at is called **SpecFlow**. `SpecFlow` is a BDD tool.

SpecFlow

User-focused behavioral tests that are written ahead of the code are the primary function behind BDD. BDD is a software development methodology that evolved from TDD. You start BDD with a list of features. Features are specifications written in a formal business language. This language is understandable by all stakeholders on a project. Once the features have been agreed and generated, it is up to the developers to then develop step definitions for the feature statements. Once the step definitions have been created, the next step is to create the external project to implement the feature and add a reference to it. The step definitions are then extended to implement the application code for the feature.

One benefit of this approach is that you, as a programmer, are guaranteed to deliver on what the business has asked for, rather than give them what you think they asked for. This can save the business a lot of money and hours. Past history has shown that many projects failed because of the lack of clarity on what needed to be delivered between the business teams and the programming teams. BDD helps to alleviate this potential hazard when developing new features.

In this section of the chapter, we will develop a very simple calculator example using the BDD software development methodology by using SpecFlow.

We will start by writing a feature file that will act as our specification with acceptance criteria. Then we will generate our step definitions from our feature file that will generate our required methods. Once our step definitions have generated the required methods, we will then write the code for them so that our feature is complete.

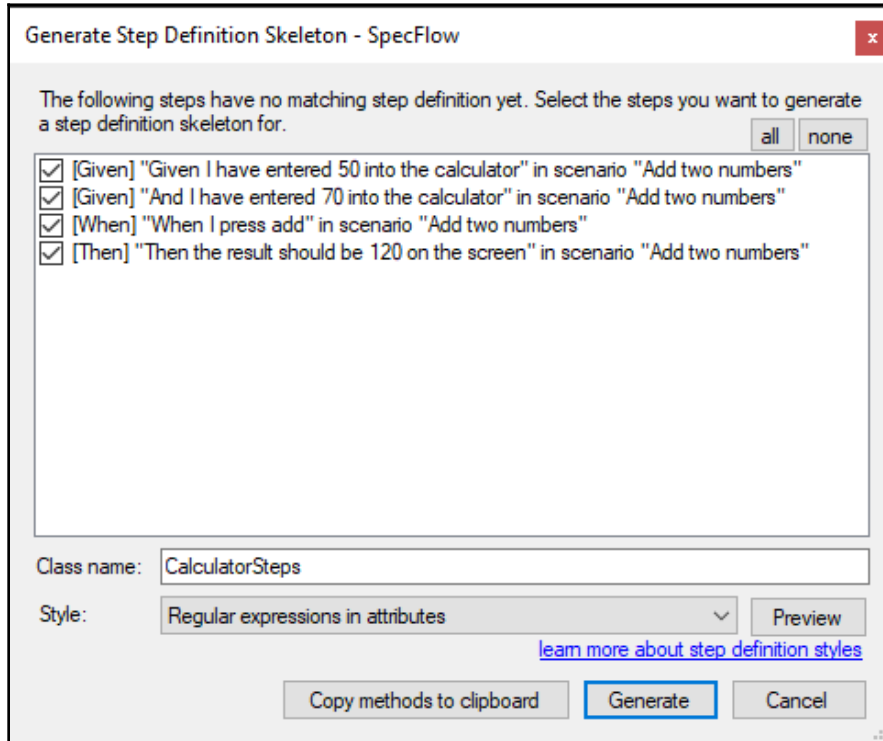
Create a new class library and add the following packages—NUnit, NUnit3TestAdapter, SpecFlow, SpecRun.SpecFlow and SpecFlow.NUnit. Add a new SpecFlow Feature file called `Calculator`:

```
Feature: Calculator
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers

@mytag
Scenario: Add two numbers
  Given I have entered 50 into the calculator
  And I have entered 70 into the calculator
  When I press add
  Then the result should be 120 on the screen
```

The preceding text is the text automatically added to the `Calculator.feature` file upon creation. So we'll use this as our starting point for learning BDD using SpecFlow. As of the time of writing, it worth noting that SpecFlow and SpecMap have been acquired by **Tricentis**. Tricentis has stated that SpecFlow, SpecFlow+, and SpecMap will all remain free, so now is a good time to learn and use SpecFlow and SpecMap if you haven't already done so.

Now that we have our feature file, we need to create step definitions that will bind our feature request to our code. Right-click in the code editor and a context menu will pop up. Select **Generate step definitions**. You should see the following dialog:



Enter the name `CalculatorSteps` for the class name. Click on the **Generate** button to generate the step definition and save the file. Open the `CalculatorSteps.cs` file and you should see the following code:

```
using TechTalk.SpecFlow;

namespace CH06_SpecFlow
{
    [Binding]
    public class CalculatorSteps
    {
        [Given(@"I have entered (.*) into the calculator")]
        public void GivenIHaveEnteredIntoTheCalculator(int p0)
        {
            ScenarioContext.Current.Pending();
        }
    }
}
```

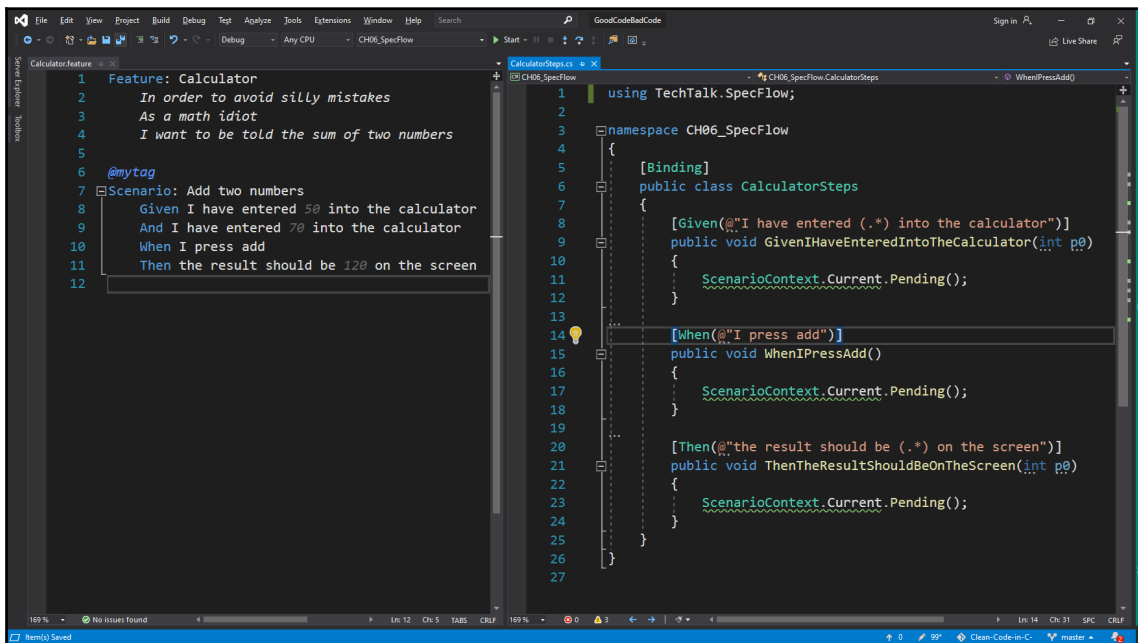
```

        [When(@"I press add")]
        public void WhenIPressAdd()
        {
            ScenarioContext.Current.Pending();
        }

        [Then(@"the result should be (.*) on the screen")]
        public void ThenTheResultShouldBeOnTheScreen(int p0)
        {
            ScenarioContext.Current.Pending();
        }
    }
}

```

A comparison of the contents of the steps file with the feature file is shown in the following screenshot:



The code that implements the feature must be in a separate file. Create a new class library and call it `CH06_SpecFlow.Implementation`. Then, add a file called `Calculator.cs`. Add a reference to the newly created library in the SpecFlow project, and the following line to the top of the `CalculatorSteps.cs` file:

```
private Calculator _calculator = new Calculator();
```

We are now in a position to extend our step definitions so that they implement the application code. In the `CalculatorSteps.cs` file, replace all the `p0` parameters with a number. This makes the parameter requirement more *explicit*. At the top of the `Calculate` class, add two public properties called `FirstNumber` and `SecondNumber`, as shown in the code that follows:

```
public int FirstNumber { get; set; }
public int SecondNumber { get; set; }
```

In the `CalculatorSteps` class, update the `GivenIHaveEnteredIntoTheCalculator()` method as shown:

```
[Given(@"I have entered (.*) into the calculator")]
public void GivenIHaveEnteredIntoTheCalculator(int number)
{
    calculator.FirstNumber = number;
}
```

Now, add the second method, `GivenIHaveAlsoEnteredIntoTheCalculator()`, if it does not already exist, and assign the `number` parameter to the calculator's second number:

```
public void GivenIHaveAlsoEnteredIntoTheCalculator(int number)
{
    calculator.SecondNumber = number;
}
```

Add `private int result;` to the top of the `CalculatorSteps` class and before any steps. Add the `Add()` method to the `Calculator` class:

```
public int Add()
{
    return FirstNumber + SecondNumber;
}
```

Now, update the `WhenIPressAdd()` method in the `CalculatorSteps` class and update the `result` variable with the result of calling the `Add()` method:

```
[When(@"I press add")]
public void WhenIPressAdd()
{
    _result = _calculator.Add();
}
```

Next up, modify the `ThenTheResultShouldBeOnTheScreen()` method as follows:

```
[Then(@"the result should be (.*) on the screen")]
public void ThenTheResultShouldBeOnTheScreen(int expectedResult)
```



```
{  
    Assert.AreEqual(expectedResult, _result);  
}
```

Build your project and run your tests. You should see that the tests pass. Only the code required by the feature to pass has been written and your code has passed the test.

You can find out more about SpecFlow at <https://specflow.org/docs/>. We've covered some of the tools available for you to develop and test your code. Now it is time to see a really simple example of how we go about coding using TDD. We'll start by writing code that fails. Then, we'll write just enough code for the test to compile. And finally, we will refactor the code.

TDD methodology practice – fail, pass, and refactor

In this section, you will learn to write tests that fail. Then you will learn to write just enough code to make the test pass, and then if necessary, you will perform any refactoring that needs to take place.

Before we delve into a practical example of TDD, let's consider why we need TDD. In the previous section, you saw how we can create feature files and generate step files from them to write code that meets a business need. Another way to ensure that your code meets the business requirements is with TDD. With TDD, you start with a test that fails. Then, you write just enough code to make the test pass, and as the need arises, you perform refactoring of your new code. This process is repeated until such time as all the features have been coded.

But *why* do we need TDD?

Business software specifications are put together by business analysts who work with project stakeholders to design new software, or extensions and modifications to existing software. Some software is critical and cannot afford to be buggy. Such software includes financial systems that handle private and business investments; medical equipment, including critical life support and scanning equipment, that requires functional software for it to work; transport signaling software for traffic management and navigation systems; space flight systems; and weapon systems.

Okay, but where does TDD fit in?

Well, you've been given a specification to write a piece of software. The first thing you need to do is create your project. Then, you write the pseudocode for the functionality that you are going to implement. You then progress to writing the tests for each piece of pseudocode. The test fails. You then write the required code that causes the test to pass, and then you refactor your code as needed. What you are doing here is writing code that is well tested and robust. You are able to guarantee that your code will execute as expected in isolation. If your code is a component of a larger system, then it will be the responsibility of the test team to test the integration of your code, not you. You, as a developer, have earned the confidence in your code to release it to the test team. If the test team identify use cases that have previously been overlooked, they will share them with you. You will then write further tests and make them pass before releasing the updated code to them. Such a way of working ensures that code is of the highest standard and can be trusted to work as expected by given the expected outputs for the given inputs. And finally, TDD makes software progress measurable, which is good news for managers.

It's time for our little demonstration of TDD. In this example, we will use TDD to develop a simple logging application that can handle inner exceptions, and logs exceptions to a timestamped text file. We will write the program and get the tests to pass. Once we have written our program and got all the tests to pass, then we will refactor our code to make it reusable and easier to read, and of course, we will make sure that our tests still pass.

1. Create a new console application and call it `CH06_FailPassRefactor`. Add a class called `UnitTests` with the following pseudocode:

```
using NUnit.Framework;

namespace CH06_FailPassRefactor
{
    [TestFixture]
    public class UnitTests
    {
        // The PseudoCode.
        // [1] Call a method to log an exception.
        // [2] Build up the text to log including
        // all inner exceptions.
        // [3] Write the text to a file with a timestamp.
    }
}
```

2. We'll write our first unit test to satisfy the condition [1]. In our unit test, we will test create the `Logger` variable, call the `Log()` method, and pass the test. So, let's write the code:

```
// [1] Call a method to log an exception.
[Test]
public void LogException()
{
    var logger = new Logger();
    var logFileName = logger.Log(new ArgumentException("Argument
cannot be null"));
    Assert.Pass();
}
```

This test will not run as the project will not build. That is because the `Logger` class does not exist. So add an internal class called `Logger` to the project. Then run your test. The build will still *fail*, and the test won't be run because we are now missing the `Log()` method. So let's add the `Log()` method to our `Logger` class. Then, we'll try and run our test again. This time, the test should succeed.

3. At this stage, we will perform any necessary refactoring. But since we have just started, there is no refactoring to do, so we can move on to our next test.

Our code to generate the log message and save it to disk will feature private members. With NUnit, you don't test private members. The school of thought is that if you have to test private members, then there must be something wrong with your code. So, we'll move on to our next unit test, which will determine whether the log file exists. Before we write our unit test, we will write a method that returns an exception with an inner exception that has an inner exception. We will pass the returned exception into the `Log()` method in our unit test:

```
private Exception GetException()
{
    return new Exception(
        "Exception: Main exception.",
        new Exception(
            "Exception: Inner Exception.",
            new Exception("Exception: Inner Exception Inner
Exception")
        )
    );
}
```

4. Now, we have our `GetException()` method in place where we can write our unit test to check whether the log file exists:

```
[Test]
public void CheckFileExists()
{
    var logger = new Logger();
    var logFile = logger.Log(GetException());
    FileAssert.Exists(logFile);
}
```

5. If we build our code and run the `CheckFileExists()` test, it will fail, so we need to write the code for it to succeed. In the `Logger` class, add `private StringBuilder _stringBuilder;` to the top of the `Logger` class. Then, modify the `Log()` method and add the following method to the `Logger` class:

```
private StringBuilder _stringBuilder;

public string Log(Exception ex)
{
    _stringBuilder = new StringBuilder();
    return SaveLog();
}

private string SaveLog()
{
    var fileName = $"LogFile{DateTime.UtcNow.GetHashCode()}.txt";
    var dir =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
    var file = $"{dir}\\{fileName}";
    return file;
}
```

6. We have called the `Log()` method and a log file is generated. Now, all we need is the text to be logged to the file. According to our pseudocode, we need to log the main exception and all inner exceptions. Let's write a test that checks whether the log file contains the message "Exception: Inner Exception Inner Exception":

```
[Test]
public void ContainsMessage()
{
    var logger = new Logger();
    var logFile = logger.Log(GetException());
    var msg = File.ReadAllText(logFile);
    Assert.IsTrue(msg.Contains("Exception: Inner Exception Inner
Exception"));
```

```
Exception"));
}
```

7. Now, we know that the test will fail because the string builder is *empty*, so we will add the method to the `Logger` class that will take an exception, log the message, and check whether the exception has an inner exception. If it has, then it will call itself with the parameter `isInnerException`:

```
private void BuildExceptionMessage(Exception ex, bool
isInnerException)
{
    if (isInnerException)
        _stringBuilder.Append("Inner Exception:
").AppendLine(ex.Message);
    else
        _stringBuilder.Append("Exception:
").AppendLine(ex.Message);
    if (ex.InnerException != null)
        BuildExceptionMessage(ex.InnerException, true);
}
```

8. Finally, update the `Log()` method of the `Logger` class to call our `BuildExceptionMessage()` method:

```
public string Log(Exception ex)
{
    _stringBuilder = new StringBuilder();
    _stringBuilder.AppendLine("-----
-----");
    BuildExceptionMessage(ex, false);
    _stringBuilder.AppendLine("-----
-----");
    return SaveLog();
}
```

All our tests now pass and we have a fully functioning program that does what's expected of it, but there is an opportunity here for some refactoring. The method called `BuildExceptionMessage()` is a candidate for reuse as it is very useful for debugging purposes, especially when you have an exception with an inner exception, so we are going to move that method into its own method. Notice that the `Log()` method is also building the opening and closing portions of the text to be logged.

We can and will move this into the `BuildExceptionMessage()` method:

1. Create a new class and call it `Text`. Add a private `StringBuilder` member variable and instantiate it in the constructor. Then, update the class by adding the following code:

```
public string ExceptionMessage => _stringBuilder.ToString();

public void BuildExceptionMessage(Exception ex, bool
isInnerException)
{
    if (isInnerException)
    {
        _stringBuilder.Append("Inner Exception:
") .AppendLine(ex.Message);
    }
    else
    {
        _stringBuilder.AppendLine("-----
-----");
        _stringBuilder.Append("Exception:
") .AppendLine(ex.Message);
    }
    if (ex.InnerException != null)
        BuildExceptionMessage(ex.InnerException, true);
    else
        _stringBuilder.AppendLine("-----
-----");
}
```

2. We've now got a useful `Text` class that returns a useful exception message from an exception with inner exceptions, but we can also refactor the code in the `SaveLog()` method. We can extract the code that generates a unique hashed filename into its own method. So, let's add the following method to the `Text` class:

```
public string GetHashedTextFileName(string name, SpecialFolder
folder)
{
    var fileName = $"{name}-{DateTime.UtcNow.GetHashCode()}.txt";
    var dir = Environment.GetFolderPath(folder);
    return $"{dir}\\{fileName}";
}
```

3. The `GetHashedTextFileName()` method accepts a name for the file specified by the user and a special folder. It then adds a hyphen and the current UTC date's hash code to the end of the filename. It then adds the `.txt` file extension and assigns the text to the `fileName` variable. The absolute path of the special folder requested by the caller is then assigned to the `dir` variable and the path and filename are then returned to the user. This method is guaranteed to return unique filenames.
4. Replace the body of the `Logger` class with the following code:

```
private Text _text;

public string Log(Exception ex)
{
    BuildMessage(ex);
    return SaveLog();
}

private void BuildMessage(Exception ex)
{
    _text = new Text();
    _text.BuildExceptionMessage(ex, false);
}

private string SaveLog()
{
    var filename = _text.GetHashedTextFileName("Log",
        Environment.SpecialFolder.MyDocuments);
    File.WriteAllText(filename, _text.ExceptionMessage);
    return filename;
}
```

The class is still doing the same thing, but it is cleaner and smaller as the message and filename generation has been moved to a separate class. If you run the code, it behaves in the same way. If you run the tests, they will all pass.

In this section, we have written unit tests that failed, and then modified them so that they passed. Then, we refactored the code to make it cleaner, which resulted in us writing code that can be reused in the same project or other projects. Let's now take a very brief look at redundant tests.

Removing redundant tests, comments, and dead code

As the book states, we are interested in writing clean code. As our programs and tests grow and we start to refactor, some code will become redundant. Any code that is redundant and does not get called is known as **dead code**. Dead code should always be removed as soon as it is identified. Dead code will not be executed in compiled code, but it is still part of the code base that needs to be maintained. Code files with dead code are longer than they need to be. Apart from the unnecessary fact that it makes your files bigger, it can also make reading source code harder, as it may cut through the natural flow of the code and add confusion and delay to the programmer reading it. Not only that, but the last thing any programmer new to the project needs is to waste valuable time trying to understand dead code that will never be used. So it is best to get rid of it.

As for comments, they can be really useful if done right, and API commenting is particularly beneficial for API documentation generation. But some comments just add noise to the code file, and a surprising number of programmers can become really irritated by them. There is one group of programmers that will comment on everything. Another group won't comment on anything as they believe the code should read like a book. And then there are those who take a balanced approach, and only comment on code when it is deemed necessary for people to understand the code.

When you see comments like this—*“This generates a random bug every so often. Don’t know why. But you’re welcome to fix it!”*—alarm bells should start ringing. First of all, the programmer who wrote the comment should have stuck with the code and not moved on until the conditions that generate the bug were identified, and then the bug should have been fixed. If you know who the programmer is who wrote the comment, then return the code to them to fix and remove the comment. I have seen code like that on more than one occasion, and I’ve seen comments on the web expressing these strong sentiments about such comments. I suppose it is a way to deal with lazy programmers. Should they not be lazy, but rather simply inexperienced, then it is a good learning task in the art of problem diagnosis and resolution.

If code has been checked in and approved, and you come across blocks of code that have been commented out, then delete them. The code will still exist in the version control history and you will be able to retrieve it from there if you have to.

Code should be read like a book, and so you should not aim to make your code cryptic just to look good and impress your colleagues, because I guarantee that when you come back to your own code in a few weeks’ time, you will scratch your head wondering what your own code does and why. I’ve seen many juniors make this mistake.

Redundant tests should also be removed. You only need to run the tests that are necessary. Tests for redundant code have no value and can waste considerable time. Also, if your company has CI/CD pipelines that also run tests in the cloud, then the redundant tests and dead code add business costs to the build, test, and deploy pipelines. This means that the fewer lines of code you upload, build, test, and deploy, the less your company has to fork out on running costs. Remember, running processes in the cloud costs money and the aim of a business is to spend as little money as possible, but rake in plenty of money.

So now that we’ve finished the chapter, let’s summarize what we’ve learned.

Summary

We started by looking at why it is important for developers to write unit tests to develop quality-assured code. Theoretical problems were identified that could arise from bugs in the software. These include loss of life and expensive lawsuits. Unit testing and what makes a good unit test was then discussed. We identified that a good unit test must be atomic, deterministic, repeatable, and fast.

Next, we went on to look at the tools available to developers that assist with TDD and BDD. MSTest and NUnit were discussed with examples that showed how to implement TDD. Then, we looked at using a mocking framework called Moq in conjunction with NUnit for testing mock objects. Our look at tools then concluded with SpecFlow—a BDD tool that allows us to write features in a business language that both techies and non-techies can understand—to make sure that what the business wants is what the business gets.

NUnit was then put to work as we worked through a very simple TDD example using the *fail, pass, and refactor* methodology, before finally looking at why we should remove unnecessary comments, redundant tests, and dead code.

At the end of this chapter, you will find further resources on testing software programs. In the next chapter, we are going to look at end-to-end testing. But before that, you might as well have a go at the following questions and see how much knowledge on unit testing you have retained.

Questions

1. What makes a good unit test?
2. What should a good unit test not be?
3. What does TDD stand for?
4. What does BDD stand for?
5. What is a unit test?
6. What is a mock object?
7. What is a fake object?
8. Name some unit testing frameworks.
9. Name some mocking frameworks.
10. Name a BDD framework.
11. What should be removed from source code files?

Further reading

- A brief overview of unit testing, with links to further information on different types of unit testing including integration testing, acceptance testing, and tester job descriptions, can be found at <http://softwaretestingfundamentals.com/unit-testing>.
- The Rhino Mocks homepage can be found at <http://hibernatingrhinos.com/oss/rhino-mocks>.

7

End-to-End System Testing

End-to-end (E2E) system testing is the automated testing of a system in its entirety. As a programmer, the unit tests for your piece of code are just a small factor in the bigger picture of the whole system. So in this chapter, we will be looking at the following topics:

- Performing E2E testing
- Coding and testing factories
- Coding and testing dependency injection
- Testing modularization

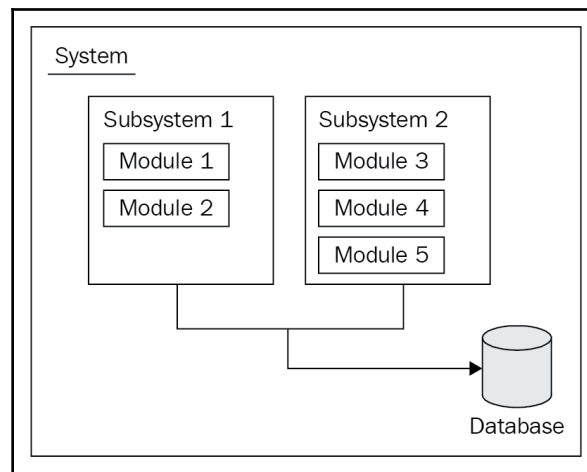
By the end of this chapter, you will have gained the following skills:

- Be able to define E2E testing
- Be able to perform E2E testing
- Be able to explain what factories are and how to use them
- Be able to understand what dependency injection is and how to use it
- Be able to understand what modularization is and how to utilize it

E2E testing

So, you've finished your project and all the unit tests pass. However, your project is a part of a larger system. This larger system will need to be tested to make sure that your code, and the other code it interfaces with, both work together as expected. Code tested in isolation can break when integrated into larger systems, and existing systems can break with the addition of new code, so it is important to perform E2E testing, also known as **integration testing**.

Integration testing is responsible for testing the complete program flow from beginning to end. Integration testing usually starts at the *requirements gathering stage*. You start by gathering and documenting the various requirements of the system. You then design all the components and devise tests for each subsystem, and then the E2E tests for the whole system. Then, you write your code according to the requirements and implement your own unit tests. Once your code is complete and the tests all pass, then the code is integrated into the overall system within the test environment and the E2E tests are executed. Often, E2E tests are carried out manually, although where possible, they can be automated as well. The following diagram shows a system that comprises two subsystems with modules and a database. In E2E testing, all these modules will be tested either manually, using automation, or by both methods:



The input to and output from each system are the main focus of the tests. You have to ask yourself, *is the correct information passed in and passed out of each system?*

Additionally, there are three things to consider when building your E2E tests:

- What *user functions* will there be, and what steps will each function perform?
- What *conditions* will there be for each function and each of its steps?
- What are the *different scenarios* that we will have to build test cases for?

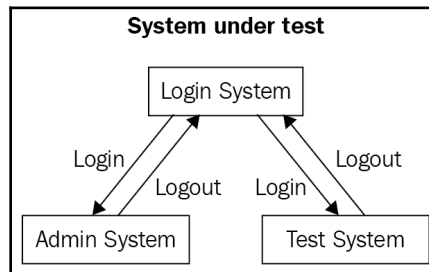
Each subsystem will have one or more features that it will provide, and each feature will have a number of actions that will be executed in a particular order. Those actions will receive inputs and provide outputs. There will also be relationships between features and functions that you must identify, after which you will need to determine whether the function is *reusable* or *independent*.

Consider the scenario of an online testing product. Teachers and students will log in to the system. If the teacher logs in, they will be taken to an admin console, and if a student logs in, they will be taken to the test menu to carry out one or more tests. In this scenario, we effectively have three subsystems:

- The login system
- The admin system
- The test system

There are two flows of execution in the aforementioned system. We have the admin flow and the test flow. Conditions and test cases will have to be established for each flow. We will use this very simple assessment system login scenario for our E2E example. In the real world, E2E will be more involved than in this chapter. The main aim of this chapter is to get you thinking about E2E testing and how you can best implement it, so we will keep things as simple as we can so that complexity does not get in the way of what we are trying to accomplish, which is to manually test three modules that must interact with each other.

The aim of this section is to build three console applications that make up the complete system: the login module, the admin module, and the test module. Then once they are built, we will go through testing them manually. The diagram that follows displays the interaction between systems. We will start with the login module:



The login module (subsystem)

The first part of our system requires both teachers and students to log in to the system using a username and password. The task list is as follows:

1. Enter the username.
2. Enter the password.
3. Press **Cancel** (this resets username and password).
4. Press **OK**.

5. If the username is invalid, then display an error message on the login page.
6. If the user is valid, then do the following:
 - If the user is a teacher, load the admin console.
 - If the user is a student, load the test console.

Let's start by creating a console application. Call it `CH07_Logon`. In the `Program.cs` class, replace the existing code with the following:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;

namespace CH07_Logon
{
    internal static class Program
    {
        private static void Main(string[] args)
        {
            DoLogin("Welcome to the test platform");
        }
    }
}
```

The `DoLogin()` method will take the passed-in string and use it for the title. Since we will not have logged in yet, the title will be set to "Welcome to the test platform". We need to add the `DoLogin()` method. The code for this method is as follows:

```
private static void DoLogin(string message)
{
    Console.WriteLine("-----");
    Console.WriteLine(message);
    Console.WriteLine("-----");
    Console.Write("Enter your username: ");
    var usr = Console.ReadLine();
    Console.Write("Enter your password: ");
    var pwd = ReadPassword();
    ValidateUser(usr, pwd);
}
```

The previous code accepts a message. The message is used as the title in the console window. The user is then prompted to enter their username and password. The `ReadPassword()` method reads all inputs and replaces filtered letters with an *asterisk* to hide the user's input. The username and password are then validated by calling the `ValidateUser()` method.

The next thing we must do is add the `ReadPassword()` method as in the code that follows:

```
public static string ReadPassword()
{
    return ReadPassword('*');
}
```

This method is really simple. It calls an overloaded method of the same name and passes in the password mask character. Let's implement the overloaded `ReadPassword()` method:

```
public static string ReadPassword(char mask)
{
    const int enter = 13, backspace = 8, controlBackspace = 127;
    int[] filtered = { 0, 27, 9, 10, 32 };
    var pass = new Stack<char>();
    char chr = (char)0;
    while ((chr = Console.ReadKey(true).KeyChar) != enter)
    {
        if (chr == backspace)
        {
            if (pass.Count > 0)
            {
                Console.Write("\b \b");
                pass.Pop();
            }
        }
        else if (chr == controlBackspace)
        {
            while (pass.Count > 0)
            {
                Console.Write("\b \b");
                pass.Pop();
            }
        }
        else if (filtered.Count(x => chr == x) <= 0)
        {
            pass.Push((char)chr);
            Console.Write(mask);
        }
    }
    Console.WriteLine();
    return new string(pass.Reverse().ToArray());
}
```


The overloaded `ReadPassword()` method accepts a password mask. This method adds each character to the stack. Unless the key being pressed is the *Enter* key, the key being pressed is checked to see if the user is performing a *Delete* keypress. If the user is performing a *Delete* keypress, then the last character entered is removed from the stack. If the character entered is not in the filtered list, then it is pushed onto the stack. The password mask is then written to the screen. As soon as the *Enter* key is pressed, a blank line is written to the console window, and the contents of the stack are reversed, returning it as a string.

The final method we need to write for this subsystem is the `ValidateUser()` method:

```
private static void ValidateUser(string usr, string pwd)
{
    if (usr.Equals("admin") && pwd.Equals("letmein"))
    {
        var process = new Process();
        process.StartInfo.FileName =
@"..\..\..\CH07_Admin\bin\Debug\CH07_Admin.exe";
        process.StartInfo.Arguments = "admin";
        process.Start();
    }
    else if (usr.Equals("student") && pwd.Equals("letmein"))
    {
        var process = new Process();
        process.StartInfo.FileName =
@"..\..\..\CH07_Test\bin\Debug\CH07_Test.exe";
        process.StartInfo.Arguments = "test";
        process.Start();
    }
    else
    {
        Console.Clear();
        DoLogin("Invalid username or password");
    }
}
```

The `ValidateUser()` method checks the username and password. If they validate as an admin, then the admin page is loaded. If they validate as a student, then the student page is loaded. Otherwise, the console is cleared, the user is informed the credentials are wrong, and they are prompted to reenter their credentials.

Upon a successful login operation being performed, the relevant subsystem is loaded and the login subsystem then terminates. Now that we have written our login module, we will write our admin module.

The admin module (subsystem)

The admin subsystem is where all the system administration is carried out. This includes the following:

- Importing students
- Exporting students
- Adding students
- Deleting students
- Editing students' profiles
- Assigning tests to students
- Changing the administrator password
- Backing up data
- Restoring data
- Erasing all data
- Viewing reports
- Exporting reports
- Saving reports
- Printing reports
- Logging out

For this exercise, we will not be implementing any of these features. I will leave you to do that as a fun exercise. All we are interested in is that the admin module loads on a successful login. If the admin module is loaded without logging in, then an error message is displayed. Then when the user presses a key, they are taken to the login module. Successful login is accomplished when a user successfully logs in as an administrator, and the admin executable is called with the *admin argument*.

Create a console application in Visual Studio and call it CH07_Admin. Update the `Main()` method as follows:

```
private static void Main(string[] args)
{
    if ((args.Count() > 0) && (args[0].Equals("admin")))
    {
        DisplayMainScreen();
    }
}
```

```
    }  
    else  
    {  
        DisplayMainScreenError();  
    }  
}
```

The `Main()` method checks that the argument count is greater than 0 and that the first argument in the array is `admin`. If it is, then the main screen is displayed by calling the `DisplayMainScreen()` method. Otherwise, the `DisplayMainScreenError()` method is called that warns the user that they must log in to access the system. It's time to write the `DisplayMainScreen()` method:

```
private static void DisplayMainScreen()  
{  
    Console.WriteLine("-----");  
    Console.WriteLine("Test Platform Administrator Console");  
    Console.WriteLine("-----");  
    Console.WriteLine("Press any key to exit");  
    Console.ReadKey();  
    Process.Start(@"..\..\..\CH07_Logon\bin\Debug\CH07_Logon.exe");  
}
```

As you can see, the `DisplayMainScreen()` method is really simple. It displays a title with a message to press any key to exit, then waits for a keypress. Upon keypress, the program shells out to the login module and exits. Now, for the `DisplayMainScreenError()` method:

```
private static void DisplayMainScreenError()  
{  
    Console.WriteLine("-----");  
    Console.WriteLine("Test Platform Administrator Console");  
    Console.WriteLine("-----");  
    Console.WriteLine("You must login to use the admin module.");  
    Console.WriteLine("Press any key to exit");  
    Console.ReadKey();  
    Process.Start(@"..\..\..\CH07_Logon\bin\Debug\CH07_Logon.exe");  
}
```

From this method, you can see that the module was started without logging in. This is not permitted. So when the user presses any key, the user is redirected to the login module, where they can log in to use the admin module. Our final module is the test module. Let's get to work and write it.

The test module (subsystem)

The test system consists of a menu. This menu displays a list of tests the student must perform, and also provides the option to exit the test system. The functions of this system include the following:

- Display a menu of tests to be completed.
- From the menu, select an item to start a test.
- On test completion, save results and return to the menu.
- When a test has been completed, remove it from the menu.
- When the user exits the test module, they are returned to the login module.

As with the previous module, I will let you have a play and add the aforementioned functionality. The main thing we are interested in here is to make sure the test module can only be run when the user has logged in. When the module is exited, the login module is loaded.

The test module is more or less a rehash of the admin module, so we will rush through this section to get to where we need to be. Update the `Main()` method as follows:

```
private static void Main(string[] args)
{
    if ((args.Count() > 0) && (args[0].Equals("test")))
    {
        DisplayMainScreen();
    }
    else
    {
        DisplayMainScreenError();
    }
}
```

Now add the `DisplayMainScreen()` method:

```
private static void DisplayMainScreen()
{
    Console.WriteLine("-----");
    Console.WriteLine("Test Platform Student Console");
    Console.WriteLine("-----");
    Console.WriteLine("Press any key to exit");
    Console.ReadKey();
    Process.Start(@"..\..\..\CH07_Logon\bin\Debug\CH07_Logon.exe");
}
```

And finally, write the `DisplayMainScreenError()` method:

```
private static void DisplayMainScreenError()
{
    Console.WriteLine("-----");
    Console.WriteLine("Test Platform Student Console");
    Console.WriteLine("-----");
    Console.WriteLine("You must login to use the student module.");
    Console.WriteLine("Press any key to exit");
    Console.ReadKey();
    Process.Start(@"..\..\..\CH07_Logon\bin\Debug\CH07_Logon.exe");
}
```

Now that we have written all three modules, we will test them in the next section.

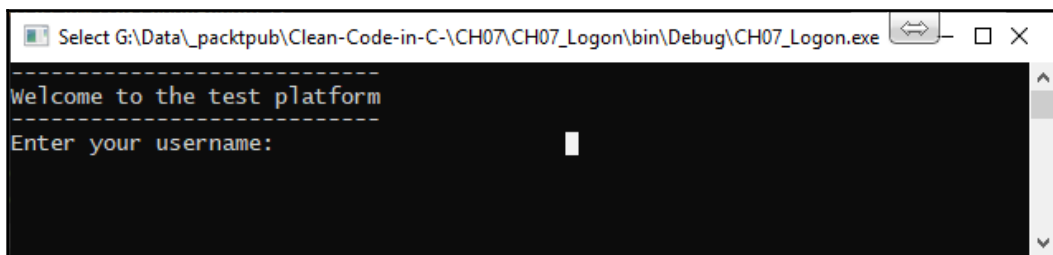
Testing our three-module system using E2E

In this section, we are going to perform a manual E2E test of our three-module system. We will test the login module to ensure that it only allows valid logins access to either the admin module or the test module. When a valid admin logs into the system, they should see the admin module, and the login module should be unloaded. When a valid student logs into the system, then they should see the test module, and the login module should be unloaded.

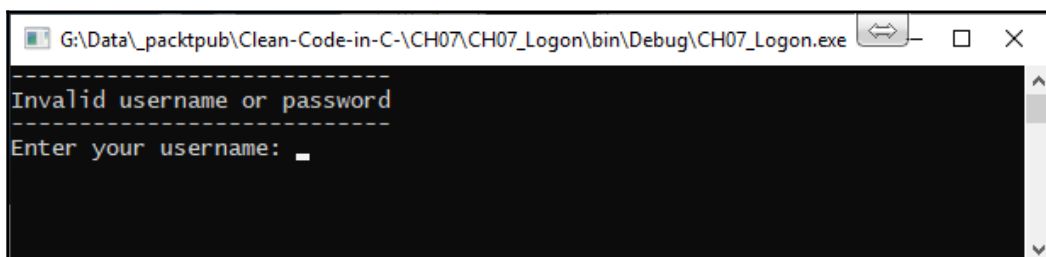
If we then try and load the admin module without first logging in, we should be warned that we must log in. Pressing any key should unload the admin module and load the login module. Trying to use the test module without logging in should behave in the same way as the admin module. We should be warned that we can't use the test module unless we log in, and pressing any key should load the login module and unload the test module.

Let's now go through the manual testing process:

1. Make sure that all the projects are built, then run the login module. You should see the screen that follows:

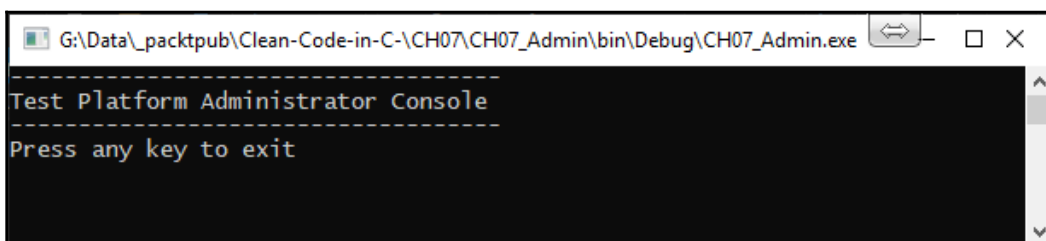


2. Enter an incorrect username and/or password, then press *Enter*, and you will see the following screen:



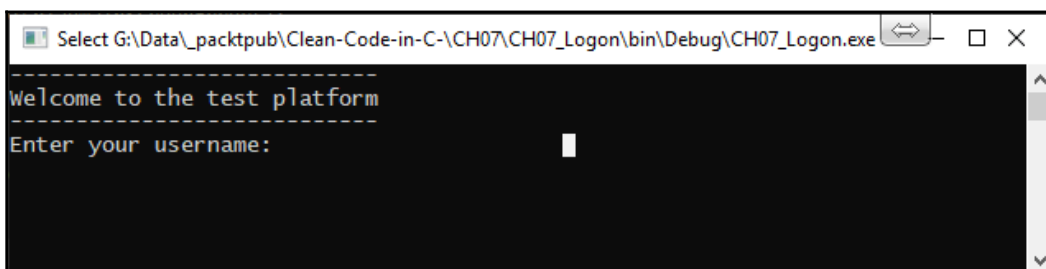
A screenshot of a Windows console window titled "G:\Data_packtpub\Clean-Code-in-C-\CH07\CH07_Logon\bin\Debug\CH07_Logon.exe". The console output shows a dashed border around the text "Invalid username or password", followed by the prompt "Enter your username: " with a cursor.

3. Now, enter `admin` as the username and `letmein` as the password, and then press *Enter*. You should see the admin module screen for a successful login:



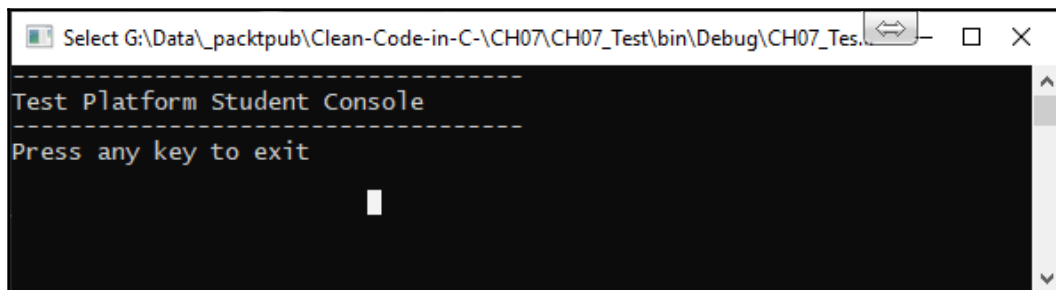
A screenshot of a Windows console window titled "G:\Data_packtpub\Clean-Code-in-C-\CH07\CH07_Admin\bin\Debug\CH07_Admin.exe". The console output shows a dashed border around the text "Test Platform Administrator Console", followed by the prompt "Press any key to exit".

4. Press any key to exit, and you should see the login module again:



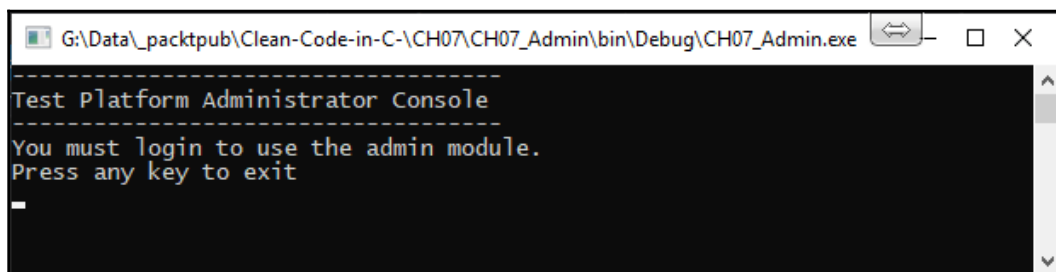
A screenshot of a Windows console window titled "Select G:\Data_packtpub\Clean-Code-in-C-\CH07\CH07_Logon\bin\Debug\CH07_Logon.exe". The console output shows a dashed border around the text "Welcome to the test platform", followed by the prompt "Enter your username: " with a cursor.

5. Enter `student` as your username and `letmein` as your password. Press `Enter` and you should be shown the student module:



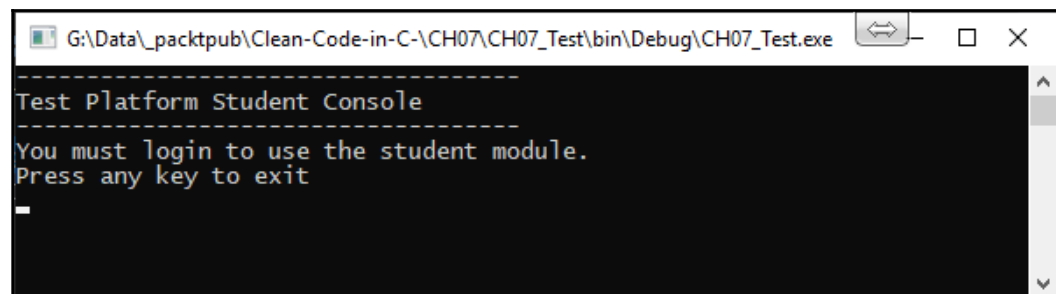
```
-----  
Test Platform Student Console  
-----  
Press any key to exit  
_
```

6. Now load the admin module without logging in, and you should see the following:



```
-----  
Test Platform Administrator Console  
-----  
You must login to use the admin module.  
Press any key to exit  
_
```

7. Pressing any key will take you back to the login module. Now load the test module without logging in, and you should see the following:



```
-----  
Test Platform Student Console  
-----  
You must login to use the student module.  
Press any key to exit  
_
```

We have now successfully manually carried out E2E testing of our system that consists of three modules. This is by far the best way to run through a system when E2E testing. Your unit tests will be very useful in making this stage fairly straightforward. By the time you get to this stage, your bugs should have been caught and dealt with. But as always, there is always the possibility of problems being encountered, which is why it is good to manually run through the system as a whole manually. That way, you can visually see through your interactions that the system behaves as expected.

Larger systems employ factories and dependency injection. In the following sections of this chapter, we will look at them both, starting with factories.

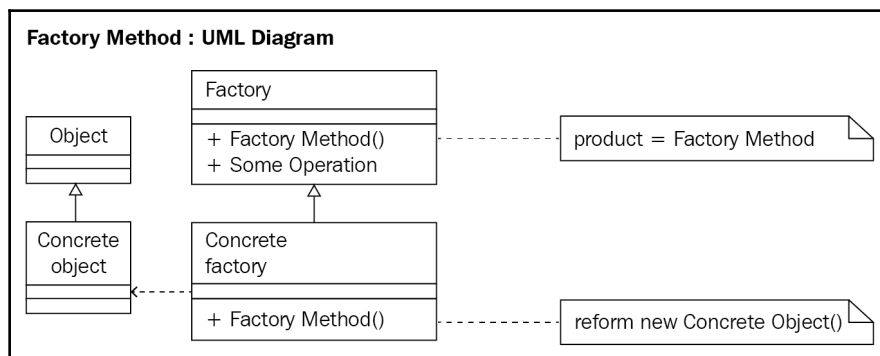
Factories

Factories are implemented using the **factory method pattern**. The intent of this pattern is to allow the creation of objects without specifying their classes. This is accomplished by invoking a factory method. The main goal of a factory method is to create an instance of a class.

You use the factory method pattern for the following scenarios:

- When the class is unable to anticipate the type of object that must be instantiated
- When the subclass must specify the type of object to instantiate
- When the class controls the instantiation of its objects

Consider the following diagram:



As you can see from the preceding diagram, you have the following items:

- `Factory`, which provides the interface for the `FactoryMethod()` that returns a type
- `ConcreteFactory`, which overrides or implements the `FactoryMethod()` to return a concrete type
- `ConcreteObject`, which inherits or implements the base class or interface

Now is a good time for a demonstration. Imagine that you have three different customers. Each customer requires using a different relational database as the backend data source. The databases used by your customers will be Oracle Database, SQL Server, and MySQL.

As a part of your E2E testing, you will need to test against each of these data sources. But how can you write the program once and have it work against any of those databases? This is where the `Factory` method pattern comes in.

Either during the installation process or via the initial configuration of your application, you can have the user specify the database that they wish to use as the data source. This information can be stored in a configuration file as an encrypted database connection string. When your application starts up, it will read the database connection string and decrypt it. The database connection string will then be passed into the factory method. Lastly, an appropriate database connection object will be selected, instantiated, and returned for use by your application.

Now that you have some background, let's create a .NET Framework Console Application in Visual Studio and call it `CH07_Factories`. Replace the code in the `App.config` file with the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.8" />
  </startup>
  <connectionStrings>
    <clear />
    <add name="SqlServer"
          connectionString="Data Source=SqlInstanceName;Initial
Catalog=DbName;Integrated Security=True"
          providerName="System.Data.SqlClient"
        />
    <add name="Oracle"
          connectionString="Data Source=OracleInstance;User
Id=usr;Password=pwd;Integrated Security=no;"
          providerName="System.Data.OracleClient"
        />
  </connectionStrings>
</configuration>
```

```
<add name="MySQL"
connectionString="Server=MySqlInstance;Database=MySqlDb;Uid=usr;Pwd=pwd; "
providerName="System.Data.MySqlClient"
/>
</connectionStrings>
</configuration>
```

As you can see, the preceding code has added the `connectionStrings` element to the configuration file. Within that element, we clear any existing connection strings and then add the three database connection strings we will be using for the application. To simplify the contents of this section, we have unencrypted connection strings, but in the production environment, make sure that your connection strings are encrypted!

In this project, we will not be using the `Main()` method in the `Program` class. We will start the `Factory` class, as follows:

```
namespace CH07_Factories
{
    public abstract class Factory
    {
        public abstract IDatabaseConnection FactoryMethod();
    }
}
```

The preceding code is our abstract factory with a single abstract `FactoryMethod()` that returns a type of `IDatabaseConnection`. Since it does not exist, we'll add that next:

```
namespace CH07_Factories
{
    public interface IDatabaseConnection
    {
        string ConnectionString { get; }
        void OpenConnection();
        void CloseConnection();
    }
}
```

In this interface, we have a read-only connection string, a method called `OpenConnection()` to open a database connection, and a method called `CloseConnection()` to close an open database connection. So far, we have our abstract `Factory` and our `IDatababaseConnection` interface. Next, we will create our concrete database connection classes. Let's start with the SQL Server database connection class:

```
public class SqlServerDbConnection : IDatabaseConnection
{
    public string ConnectionString { get; }
```

```
public SqlServerDbConnection(string connectionString)
{
    ConnectionString = connectionString;
}
public void CloseConnection()
{
    Console.WriteLine("SQL Server Database Connection Closed.");
}
public void OpenConnection()
{
    Console.WriteLine("SQL Server Database Connection Opened.");
}
}
```

As you can see, the `SqlServerDbConnection` class fully implements the `IDatabaseConnection` interface. The constructor takes `connectionString` as a single parameter. The read-only `ConnectionString` property is then assigned to `connectionString`. The `OpenConnection()` method only prints to the console.

In a real implementation, however, the connection string would be used to connect to the valid data source specified in the string. Once a database connection is open, it must be *closed*. The closing of the database connection would be carried out by the `CloseConnection()` method. Next, we repeat the preceding process for the Oracle database connection and the MySQL database connection:

```
public class OracleDbConnection : IDatabaseConnection
{
    public string ConnectionString { get; }
    public OracleDbConnection(string connectionString)
    {
        ConnectionString = connectionString;
    }
    public void CloseConnection()
    {
        Console.WriteLine("Oracle Database Connection Closed.");
    }
    public void OpenConnection()
    {
        Console.WriteLine("Oracle Database Connection Closed.");
    }
}
```

We now have the `OracleDbConnection` class in place. So, the last class we need to implement is the `MySqlDbConnection` class:

```
public class MySqlDbConnection : IDatabaseConnection
{
    public string ConnectionString { get; }
    public MySqlDbConnection(string connectionString)
    {
        ConnectionString = connectionString;
    }
    public void CloseConnection()
    {
        Console.WriteLine("MySQL Database Connection Closed.");
    }
    public void OpenConnection()
    {
        Console.WriteLine("MySQL Database Connection Closed.");
    }
}
```

With that, we have added our concrete classes. The only thing left to do is to create our `ConcreteFactory` class that inherits the abstract `Factory` class. You will need to reference the `System.Configuration.ConfigurationManager` NuGet packet:

```
using System.Configuration;

namespace CH07_Factories
{
    public class ConcreteFactory : Factory
    {
        private static ConnectionStringSettings _connectionStringSettings;

        public ConcreteFactory(string connectionStringName)
        {
            GetDbConnectionSettings(connectionStringName);
        }

        private static ConnectionStringSettings
        GetDbConnectionSettings(string connectionStringName)
        {
            return
            ConfigurationManager.ConnectionStrings[connectionStringName];
        }
    }
}
```

As we can see, the class uses the `System.Configuration` namespace. The `ConnectionStringSettings` values are stored in the `_connectionStringSettings` member variable. This is set in the constructor that takes `connectionStringName`. The name is passed into the `GetDbConnectionSettings()` method. The quick among you will see an obvious mistake in the constructor.

The method is getting called but the member variable is not being set. However, we will pick up this oversight and fix it when we come to run the tests that we have yet to write. The `GetDbConnectionSettings()` methods uses `ConfigurationManager` to read the required connection string from the `ConnectionStrings[]` array.

Now, it is time to complete our `ConcreteClass` by adding `FactoryMethod()`:

```
public override IDatabaseConnection FactoryMethod()
{
    var providerName = _connectionStringSettings.ProviderName;
    var connectionString = _connectionStringSettings.ConnectionString;
    switch (providerName)
    {
        case "System.Data.SqlClient":
            return new SqlServerDbConnection(connectionString);
        case "System.Data.OracleClient":
            return new OracleDbConnection(connectionString);
        case "System.Data.MySqlClient":
            return new MySqlDbConnection(connectionString);
        default:
            return null;
    }
}
```

Our `FactoryMethod()` returns a concrete class of type `IDatabaseConnection`. At the start of the class, the member variable is read and the values are stored locally for `providerName` and `connectionString`. A switch is then used to determine what type of database connection to build and pass back.

We are now in a position to test our factory to see whether it works with the different types of databases used by our customers. This test can be done manually, but for the purpose of this exercise, we are going to write automation tests.

Create a new NUnit test project. Add a reference to the `CH07_Factories` project. Then, add the `System.Configuration.ConfigurationManager` NuGet package. Rename the class to `UnitTests.cs`. Now, add the first test, as shown:

```
[Test]
public void IsSqlServerDbConnection()
{
    var factory = new ConcreteFactory("SqlServer");
    var connection = factory.FactoryMethod();
    Assert.IsInstanceOf<SqlServerDbConnection>(connection);
}
```

This test is for a SQL Server database connection. It creates a new `ConcreteFactory()` instance and passes in the `connectionStringName` value of `"SqlServer"`. The factory then instantiates and returns the correct database connection object via `FactoryMethod()`. Finally, the connection object is asserted to test that it is indeed an instance of type `SqlServerDbConnection`. We need to write the previous test twice more for the other database connections, so let's now add the Oracle database connection test:

```
[Test]
public void IsOracleDbConnection()
{
    var factory = new ConcreteFactory("Oracle");
    var connection = factory.FactoryMethod();
    Assert.IsInstanceOf<OracleDbConnection>(connection);
}
```

The test passes in the `connectionStringName` value of `"Oracle"`. An assertion is made to test whether the connection object returned is of type `OracleDbConnection`. Last of all, we have our MySQL database connection test:

```
[Test]
public void IsMySqlDbConnection()
{
    var factory = new ConcreteFactory("MySQL");
    var connection = factory.FactoryMethod();
    Assert.IsInstanceOf<MySqlDbConnection>(connection);
}
```

The test passes in the `connectionStringName` value of "MySQL". An assertion is made to test whether the connection object returned is of type `MySqlDbConnection`. If we run our tests now, they will all fail because the `_connectionStringSettings` variable is not getting set, so let's fix this. Modify your `ConcreteFactory` constructor as follows:

```
public ConcreteFactory(string connectionStringName)
{
    _connectionStringSettings =
        GetDbConnectionSettings(connectionStringName);
}
```

If you run all your tests now, they should work. If your connection string is not getting picked up by NUnit, then it will be looking in a different `App.config` file to what you are expecting. Add the following line before the line that reads the connection string:

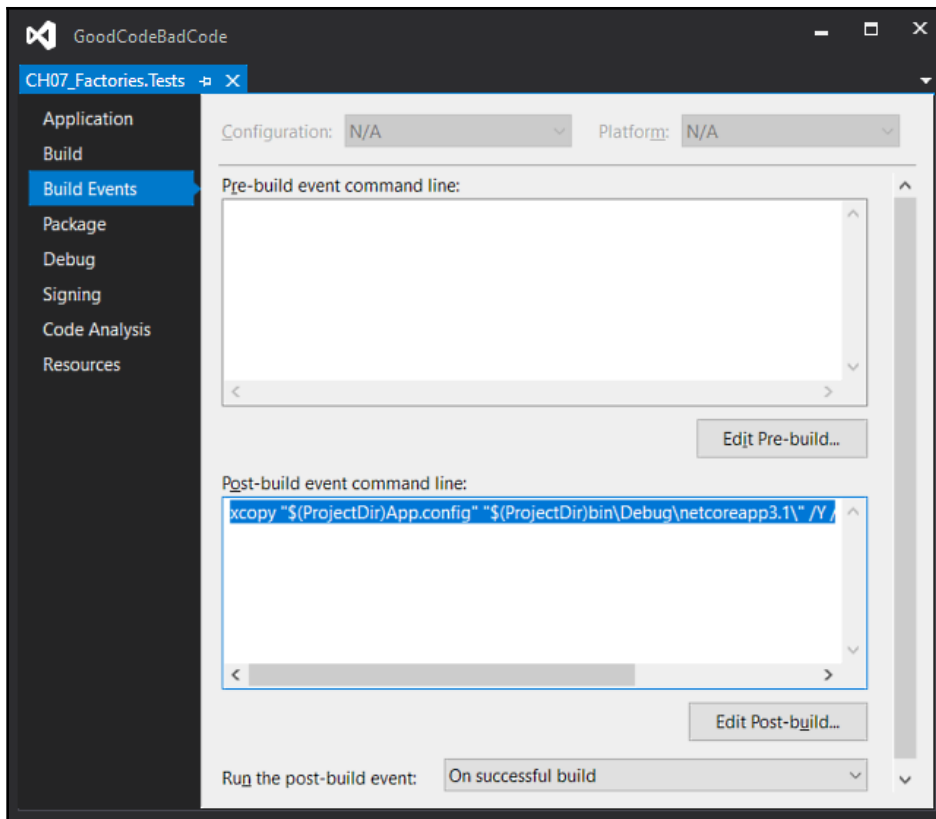
```
var filepath =
    ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None).File
    Path;
```

This will inform you where NUnit is looking for your connection string settings. If the file does not exist, you can create it manually and duplicate the contents from your main `App.config` file. But the problem with this is that the file will more than likely get deleted upon the next build. So to make the change permanent, you can add a post-build event command line to your test project.

To do this, right-click on your test project and select **Properties**. Then on the **Properties** tab, select **Build Events**. In the post-build event command line, add the following command:

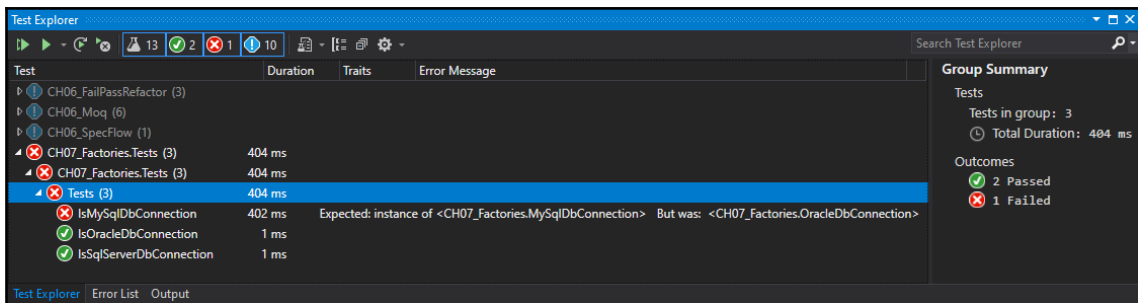
```
xcopy "$(ProjectDir)App.config" "$(ProjectDir)bin\Debug\netcoreapp3.1\" /Y
/I /R
```

The following screenshot shows the **Build Events** page of the **Project Properties** dialog with the **Post-build event command line** in place:



This will create the missing file in the test project output folder. The file on your system may be named `testhost.x86.dll.config`, since it is on my system. Now, your builds should be working.

If you change the return type of one of the cases in `FactoryMethod()`, you will see that your test fails, as shown in the following screenshot:



Change the code back to the correct type so that your code now passes.

We have seen how to manually E2E test a system, along with how to employ software factories, and how we can automatically test whether our factories function as expected. Now we will look at dependency injection and how this can be E2E tested.

Dependency injection

Dependency Injection (DI) helps you to produce code that is loosely coupled by separating the code's behavior from its dependencies, which leads to more readable code that is easy to test, extend, and maintain. Code is more readable because you follow the single responsibility principle. This also leads to much smaller code. Smaller code is easier to maintain and test, and because we rely upon abstractions instead of on implementations, we can extend the code more easily according to our needs.

The following are the types of DI that you can implement:

- Constructor injection
- Property/setter injection
- Method injection

Poor man's DI is composed without a container. However, the recommended and best practice is to use a DI container. In simple terms, a DI container is a registration framework that instantiates dependencies and injects them when requested.

We are now going to write our own dependency container, interface, services, and client for our DI example. Then we will write our tests for the dependency project. Bear in mind that even though tests should be written first, in most business situations I have encountered, they are written once the software has been written! So in this scenario, we will write our tests after the software we want has been coded. This can often happen when you employ multiple teams where some utilize TDD and some don't, or you use third-party code for which no tests exist.

We mentioned earlier that E2E is best done manually and that automation is hard, but you can automate tests of the system, as well as performing manual testing. This is particularly useful if you target multiple data sources.

The first thing you need to have in place is a dependency container. The dependency container keeps a register of types and instances. You register types before you use them. When it is time to use an instance of an object, you resolve it into a variable and inject (pass it) into the constructor, method, or property.

Create a new class library and call it `CH07_DependencyInjection`. Add a new class called `DependencyContainer`, and add the following code:

```
public static readonly IDictionary<Type, Type> Types = new Dictionary<Type,
Type>();
public static readonly IDictionary<Type, object> Instances = new
Dictionary<Type, object>();

public static void Register<TContract, TImplementation>()
{
    Types[typeof(TContract)] = typeof(TImplementation);
}

public static void Register<TContract, TImplementation>(TImplementation
instance)
{
    Instances[typeof(TContract)] = instance;
}
```

In this code, we have two dictionaries that house the types and the instances. We also have two methods. One is used to register our types, and the second is used to register our instances. Now that we have the code to register and store our types and instances, we need a way to resolve them at runtime. Add the following code to the `DependencyContainer` class:

```
public static T Resolve<T>()
{
    return (T)Resolve(typeof(T));
}
```

This method is passed in a type. It calls the method to resolve the type and returns an instance of that type. So, let's add that method now:

```
public static object Resolve(Type contract)
{
    if (Instances.ContainsKey(contract))
    {
        return Instances[contract];
    }
    else
    {
        Type implementation = Types[contract];
        ConstructorInfo constructor = implementation.GetConstructors()[0];
        ParameterInfo[] constructorParameters =
constructor.GetParameters();
        if (constructorParameters.Length == 0)
        {
```

```
        return Activator.CreateInstance(implementation);
    }
    List<object> parameters = new
List<object>(constructorParameters.Length);
    foreach (ParameterInfo parameterInfo in constructorParameters)
    {
        parameters.Add(Resolve(parameterInfo.ParameterType));
    }
    return constructor.Invoke(parameters.ToArray());
}
}
```

The `Resolve()` method checks to see whether the `Instances` dictionary contains an instance whose key matches the contract. If it does, then that instance is returned. Otherwise, a new instance is created and returned.

Now, we need an interface that our services to be injected will implement. We'll call it `IService`. It will have a single method that will return a string, and the method will be called `WhoAreYou()`:

```
public interface IService
{
    string WhoAreYou();
}
```

Our services to be injected will implement the aforementioned interface. Our first class will be named `ServiceOne`, and the method will return the string `"CH07_DependencyInjection.ServiceOne()"`:

```
public class ServiceOne : IService
{
    public string WhoAreYou()
    {
        return "CH07_DependencyInjection.ServiceOne()";
    }
}
```

The second service is the same except it is called `ServiceTwo`, and the method returns the string `"CH07_DependencyInjection.ServiceTwo()"`:

```
public class ServiceTwo : IService
{
    public string WhoAreYou()
    {
        return "CH07_DependencyInjection.ServiceTwo()";
    }
}
```

The dependency container, interface, and service classes are now in place. Finally, we are going to add the client that will be used as the demonstration object that will consume our services via DI. Our class will demonstrate constructor injection, property injection, and method injection. Add the following code to the top of the class:

```
private IService _service;

public Client() { }
```

The `_service` member variable will be used to store our injected service. We have a default constructor so that we can test our property and method injection. Add the constructor that accepts and sets the `IService` member:

```
public Client (IService service)
{
    _service = service;
}
```

Next, we will add our property to test property injection and constructor injection:

```
public IService Service
{
    get { return _service; }
    set
    {
        _service = value;
    }
}
```

Then we'll add a method that calls `WhoAreYou()` on the injected object. The `Service` property allows the `_service` member variable to be set and retrieved. Finally, we will add our `GetServiceName()` method:

```
public string GetServiceName(IService service)
{
    return service.WhoAreYou();
}
```

The `GetServiceName()` method is called on the injected instance of the `IService` class. This method returns the fully qualified name of the service passed in. Now we will write the unit tests to test the functionality. Add a test project and reference the dependency project. Call the test project `CH07_DependencyInjection.Tests` and rename `UnitTest1` to `UnitTests`.

We will write tests to check that our registration and resolving of instances works, and that the correct classes are injected by constructor injection, setter injection, and method injection. Our tests will test the injection of `ServiceOne` and `ServiceTwo`. Let's start by writing our `Setup()` method as follows:

```
[TestInitialize]
public void Setup()
{
    DependencyContainer.Register<ServiceOne, ServiceOne>();
    DependencyContainer.Register<ServiceTwo, ServiceTwo>();
}
```

In our `Setup()` method, we register both our implementations of the `IService` class, these being `ServiceOne()` and `ServiceTwo()`. Now we will write our two test methods to test the dependency container:

```
[TestMethod]
public void DependencyContainerTestServiceOne()
{
    var serviceOne = DependencyContainer.Resolve<ServiceOne>();
    Assert.IsInstanceOfType(serviceOne, typeof(ServiceOne));
}

[TestMethod]
public void DependencyContainerTestServiceTwo()
{
    var serviceTwo = DependencyContainer.Resolve<ServiceTwo>();
    Assert.IsInstanceOfType(serviceTwo, typeof(ServiceTwo));
}
```

Both these methods call the `Resolve()` method. The method checks for an instance of a type. If an instance exists, it returns it. Otherwise, one is instantiated and returned. It's time to write the constructor injection tests for `serviceOne` and `serviceTwo`:

```
[TestMethod]
public void ConstructorInjectionTestServiceOne()
{
    var serviceOne = DependencyContainer.Resolve<ServiceOne>();
    var client = new Client(serviceOne);
    Assert.IsInstanceOfType(client.Service, typeof(ServiceOne));
}

[TestMethod]
public void ConstructorInjectionTestServiceTwo()
{
    var serviceTwo = DependencyContainer.Resolve<ServiceTwo>();
    var client = new Client(serviceTwo);
}
```

```
        Assert.IsInstanceOfType(client.Service, typeof(ServiceTwo));  
    }
```

In both of these constructor test methods, we resolve the relevant service from the container registry. Then we pass the service into the constructor. Finally, using the `Service` property, we assert that the service passed in via the constructor is an instance of the expected service. Let's write the test to show that the property setter injection works as expected:

```
[TestMethod]  
public void PropertyInjectTestServiceOne()  
{  
    var serviceOne = DependencyContainer.Resolve<ServiceOne>();  
    var client = new Client();  
    client.Service = serviceOne;  
    Assert.IsInstanceOfType(client.Service, typeof(ServiceOne));  
}  
  
[TestMethod]  
public void PropertyInjectTestServiceTwo()  
{  
    var serviceTwo = DependencyContainer.Resolve<ServiceTwo>();  
    var client = new Client();  
    client.Service = serviceTwo;  
    Assert.IsInstanceOfType(client.Service, typeof(ServiceOne));  
}
```

To test that the setter injection resolves the class we are after, create a client using the default constructor, then assign the resolved instance to the `Service` property. Next, we assert whether the service is an instance of the expected type or not. Finally, for our tests, we just need to test our method injection:

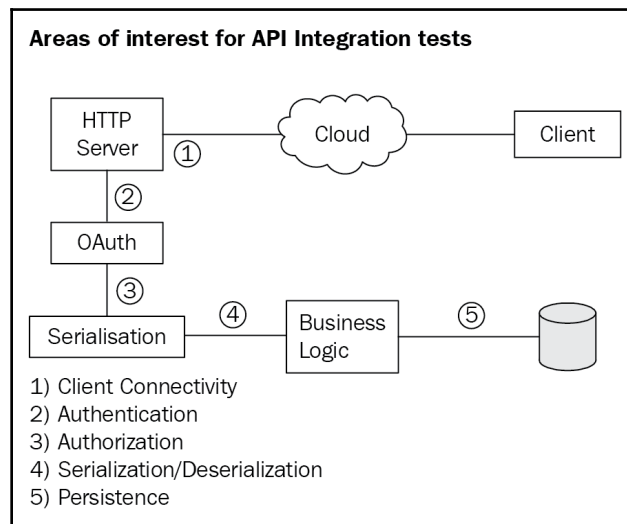
```
[TestMethod]  
public void MethodInjectionTestServiceOne()  
{  
    var serviceOne = DependencyContainer.Resolve<ServiceOne>();  
    var client = new Client();  
    Assert.AreEqual(client.GetServiceName(serviceOne),  
        "CH07_DependencyInjection.ServiceOne()");  
}  
  
[TestMethod]  
public void MethodInjectionTestServiceTwo()  
{  
    var serviceTwo = DependencyContainer.Resolve<ServiceTwo>();  
    var client = new Client();  
    Assert.AreEqual(client.GetServiceName(serviceTwo),
```

```
"CH07_DependencyInjection.ServiceTwo()");  
}
```

Here, we again resolve our instance. Create a new client using the default constructor and assert passing in the resolved instance and that calling the `GetServiceName()` method returns the correct identity of the passed-in instance.

Modularization

A system consists of one or more modules. When a system con two or more modules, you need to test the interaction between them to make sure they work together as expected. Let's consider the system for an API shown in the following diagram:



As you can see from the previous diagram, we have a client that accesses a data store in the cloud via an API. The client sends a request to the HTTP server. The request is authenticated. Once it has been authenticated, the request is then authorized to access the API. The data sent by the client is deserialized and then passed on to the business layer. The business layer then performs either a read, insert, update, or delete operation on the data store. The data is then passed back to the client from the database via the business layer, followed by the serialization layer, and then back to the client.

As you can see, we have a number of modules that interact with each other. We have the following:

- Security (**Authentication** and **Authorization**) interacting with serialization (**Serialization** and **Deserialization**)
- Serialization interacting with the business layer that contains all the business logic
- The **Business Logic** layer interacting with the data store

If we look at these three preceding points, we can see that a number of tests can be written to automate the E2E testing process. Many tests are essentially unit tests that become incorporated into our suite of integration tests. Let's consider some now. We are able to test the following:

- Correct login
- Incorrect login
- Authorized access
- Unauthorized access
- Serialization of data
- Deserialization of data
- Business logic
- Database read
- Database update
- Database insert
- Database delete

As you can see from these tests, they are unit tests over integration tests. So, what integration tests could we write? Well, we could write the following tests:

- Send a read request.
- Send an insert request.
- Send an edit request.
- Send a delete request.

Those four tests could be written using the correct username and password and well-formed data requests, and they could also be written for invalid usernames or passwords and malformed data requests.

So, you can perform integration testing by using unit tests to test the code in each module, then using tests that only test the interaction between two modules at a time. You can also write tests that perform a full E2E operation.

But despite being able to test all this with code, the one thing you *must* do is run through the system manually to verify that everything works as expected.

With all these tests completed successfully, you can have the confidence to release your code to the production environment.

Now that we have covered E2E testing (also known as **integration testing**), let's take some time to summarize what we have learned.

Summary

In this chapter, we looked at what E2E testing is. We saw that we can write automated tests, but we've also come to understand the importance of manually testing the complete application from an end user perspective.

When we looked at factories, we saw an example of their use when it comes to database connectivity. We considered a scenario where our app will enable users to use a database of their choice. We load in a connection string, and then based on that connection string, the relevant database connection object is instantiated and returned for use. We saw how we could test our factories for each use case for each different database. Factories can be used in a number of different scenarios, and now you know what they are, how to use them, and most importantly, you know how to test them.

DI enables a single class to work with multiple different implementations of an interface. We saw this in action when we wrote our own dependency container. The interface we created was implemented by two classes, added to the dependency register, and resolved when called upon by the dependency container. We implemented unit tests to test the different implementations for constructor injection, property injection, and method injection.

Then, we looked at modules. A simple application may consist of a single module, but the more an application grows in complexity, the more modules will make up that application. As the number of modules grows, so does the opportunity for something to go wrong. Therefore, it is very important to test the interaction between modules. The modules themselves can be tested using unit tests. The interaction between modules can be tested with more involved tests that run through a complete scenario from start to finish.

In the next chapter, we will be looking at best practices when working with threading and concurrency. But first, let's test your knowledge on the contents of this chapter.

Questions

1. What is E2E testing?
2. What is another term for E2E testing?
3. What methods should we employ during E2E testing?
4. What are factories, and why do we use them?
5. What is DI?
6. Why should we use a dependency container?

Further reading

- The book *Dependency Injection in .NET* by Manning will introduce you to .NET DI before guiding you through the various DI frameworks.

8

Threading and Concurrency

A process is essentially a program that is executing on an operating system. This process is made up of more than one thread of execution. A thread of execution is a set of commands issued by a process. The ability to execute more than one thread at a time is known as **multi-threading**. In this chapter, we are going to look at multi-threading and concurrency.

Multiple threads are allotted a set amount of time to execute, and each thread is executed on a rotational basis by a thread scheduler. The thread scheduler schedules the threads using a technique called **time slicing** and then passes each thread to the CPU to be executed at the scheduled time.

Concurrency is the ability to run more than one thread at exactly the same time. This can be accomplished on computers with more than one processor core. The more processor cores a computer has, the more threads of execution can be executed concurrently.

As we look at concurrency and threading in this chapter, we will encounter the problems of blocking, deadlocks, and race conditions. You will see how we can overcome these problems using clean coding techniques.

In the course of this chapter, we will cover each of the following topics:

- Understanding the thread life cycle
- Adding thread parameters
- Using a thread pool
- Using a mutual exclusion object with synchronous threads
- Working with parallel threads using semaphores
- Limiting the number of processors and threads in the thread pool
- Preventing deadlocks
- Preventing race conditions
- Understanding static constructors and methods
- Mutability, immutability, and thread safety

- Synchronized method dependencies
- Using the `Interlocked` class for simple state changes
- General recommendations

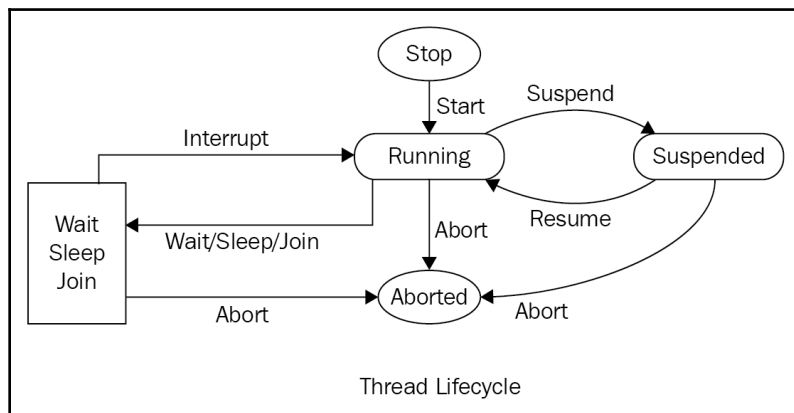
After working through this chapter and developing your threading and concurrency skills, you will have acquired the following skills:

- The ability to understand and discuss the thread life cycle
- An understanding of and ability to use foreground and background threads
- The ability to throttle threads and set the number of processors to use concurrently using a thread pool
- The ability to understand the effects of static constructors and methods in relation to multi-threading and concurrency
- The ability to take into account mutability and immutability and their impact on thread safety
- The ability to understand what causes race conditions and how to avoid them
- The ability to understand what causes deadlocks and how to avoid them
- The ability to perform simple state changes using the `Interlocked` class

To run through the code in this chapter, you will need a .NET Framework console application. Unless otherwise stated, all code will be placed in the `Program` class.

Understanding the thread life cycle

Threads in C# have an associated life cycle. The life cycle for threads is as follows:



When a thread starts, it enters the **running** state. When running, the thread can enter a **wait**, **sleep**, **join**, **stop**, or **suspended** state. Threads can also be aborted. Aborted threads enter the stop state. You can suspend and resume a thread by calling the `Suspend()` and `Resume()` methods, respectively.

A thread will enter the wait state when the `Monitor.Wait(object obj)` method is called. The thread will then continue when the `Monitor.Pulse(object obj)` method is called. Threads enter sleep mode by calling the `Thread.Sleep(int millisecondsTimeout)` method. Once the elapsed time has passed, the thread returns to the running state.

The `Thread.Join()` method causes a thread to enter the wait state. A joined thread will remain in the wait state until all dependent threads have finished running, upon which it will enter the running state. However, if any dependent threads are aborted, then this thread is also aborted and enters the stop state.



Threads that have completed or have been aborted cannot be restarted.

Threads can run in the foreground or the background. Let's look at both foreground and background threads, starting with foreground threads:

- **Foreground threads:** By default, threads run in the foreground. A process will continue to run while at least one foreground thread is currently running. Even if `Main()` completes but a foreground thread is running, the application process will remain active until the foreground thread terminates. Creating a foreground thread is really simple, as the following code shows:

```
var foregroundThread = new Thread(SomeMethodName);  
foregroundThread.Start();
```

- **Background threads:** You create a background thread in the same way that you create foreground threads, except that you also have to explicitly set a thread to run in the background, as shown:

```
var backgroundThread = new Thread(SomeMethodName);  
backgroundThread.IsBackground = true;  
backgroundThread.Start();
```

Background threads are used to carry out background tasks and keep the user interface responsive to the user. When the main process terminates, any background threads that are executing are also terminated. However, even if the main process terminates, any foreground threads that are running will run to completion.

In the next section, we will look at thread parameters.

Adding thread parameters

Methods that run in threads often have parameters. So, when executing a method within a thread, it is useful to know how to pass the method parameters into the thread.

Let's say that we have the following method, which adds two integers together and returns a result:

```
private static int Add(int a, int b)
{
    return a + b;
}
```

As you can see, the method is simple. There are two parameters called *a* and *b*. These two parameters will need to be passed into the thread for the `Add()` method to run properly. We will add an example method that will do just that:

```
private static void ThreadParametersExample()
{
    int result = 0;
    Thread thread = new Thread(() => { result = Add(1, 2); });
    thread.Start();
    thread.Join();
    Message($"The addition of 1 plus 2 is {result}.");
}
```

In this method, we declare an integer with an initial value of 0. We then create a new thread that calls the `Add()` method with the 1 and 2 parameter values, and then assign the result to the integer variable. The thread then starts and we wait for it to finish executing by calling the `Join()` method. Finally, we print the result to the console window.

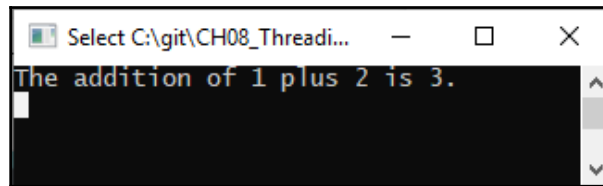
Let's add our `Message()` method:

```
internal static void Message(string message)
{
    Console.WriteLine(message);
}
```

The `Message()` method simply takes a string and outputs it to the console window. All we have to do now is update the `Main()` method, as follows:

```
static void Main(string[] args)
{
    ThreadParametersExample();
    Message("=== Press any Key to exit ===");
    Console.ReadKey();
}
```

In our `Main()` method, we call our example method and then wait for the user to press any key before exiting. You should see the following output:



As you can see, 1 and 2 were the method parameters passed into the addition method, and 3 was the value returned by the thread. The next topic we will look at is using a thread pool.

Using a thread pool

A thread pool improves performance by creating a collection of threads during application initialization. When a thread is required, it is assigned a single task. That task will be executed. Once executed, the thread is returned to the thread pool to be reused.

Since thread creation is expensive in .NET, we can improve performance by using a thread pool. Each process has a fixed number of threads based on the system resources available, such as memory and the CPU. However, we can increase or decrease the number of threads used by the thread pool. It is normally best to let the thread pool take care of how many threads to use, rather than manually setting these values.

The different ways to create a thread pool are as follows:

- Using the **Task Parallel Library (TPL)** (on .NET Framework 4.0 and higher)
- Using `ThreadPool.QueueUserWorkItem()`
- Using asynchronous delegates
- Using `BackgroundWorker`



As a rule of thumb, you should only use a thread pool for server-side applications. For client-side applications, use foreground and background threads, as necessary.

In this book, we will just look at the **TPL** and the `QueueUserWorkItem()` method. You can check out how to use the other two methods at <http://www.albahari.com/threading/>. We'll look at the TPL next.

Task Parallel Library

An asynchronous operation in C# is represented by a task. A task in C# is represented by the `Task` class in the TPL. As you will gather from the name, task parallelism enables multiple tasks to be executed concurrently, which we will learn about in the following subsections. The first `Parallel` class method we will look at is the `Invoke()` method.

Parallel.Invoke()

In our first example, we will invoke three separate methods using `Parallel.Invoke()`. Add the following three methods:

```
private static void MethodOne()
{
    Message($"MethodOne Executed: Thread
    Id({Thread.CurrentThread.ManagedThreadId})");
}

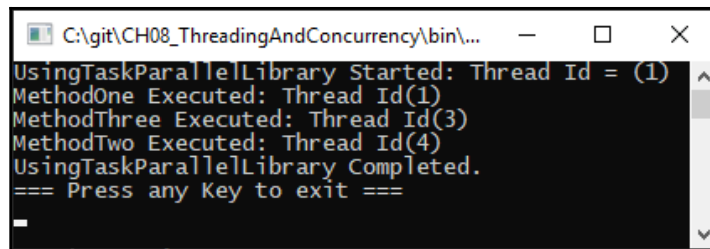
private static void MethodTwo()
{
    Message($"MethodTwo Executed: Thread
    Id({Thread.CurrentThread.ManagedThreadId})");
}

private static void MethodThree()
{
    Message($"MethodThree Executed: Thread
    Id({Thread.CurrentThread.ManagedThreadId})");
}
```


As you can see, these three methods are almost identical, apart from their names and the message printed to the console window via the `Message()` method we wrote earlier. Now, we'll add the `UsingTaskParallelLibrary()` method to execute these three methods in parallel:

```
private static void UsingTaskParallelLibrary()
{
    Message($"UsingTaskParallelLibrary Started: Thread Id =
    ({Thread.CurrentThread.ManagedThreadId})");
    Parallel.Invoke(MethodOne, MethodTwo, MethodThree);
    Message("UsingTaskParallelLibrary Completed.");
}
```

In this method, we write a message to the console window indicating the start of the method. We then invoke the `MethodOne`, `MethodTwo`, and `MethodThree` methods in parallel. Then, we write a message to the console window indicating that the method has reached its end, and then we wait for a key to be pressed before exiting the method. Run the code and you should see the following output:



```
C:\git\CH08_ThreadingAndConcurrency\bin\...
UsingTaskParallelLibrary Started: Thread Id = (1)
MethodOne Executed: Thread Id(1)
MethodThree Executed: Thread Id(3)
MethodTwo Executed: Thread Id(4)
UsingTaskParallelLibrary Completed.
=== Press any Key to exit ===
_
```

In the preceding screenshot, you can see that thread one is reused. Let's now move on to the `Parallel.For()` loop.

Parallel.For()

In our next TPL example, we will look at a simple `Parallel.For()` loop. Add the following method to the `Program` class of a new .NET Framework console application:

```
private static void Method()
{
    Message($"Method Executed: Thread
    Id({Thread.CurrentThread.ManagedThreadId})");
}
```

All this method does is output a string to the console window. We'll now create the method that executes the `Parallel.For()` loop:

```
private static void UsingTaskParallelLibraryFor()
{
    Message($"UsingTaskParallelLibraryFor Started: Thread Id =
    ({Thread.CurrentThread.ManagedThreadId})");
    Parallel.For(0, 1000, X => Method());
    Message("UsingTaskParallelLibraryFor Completed.");
}
```

In this method, we loop through 0 to 1000, calling `Method()`. You will see how the threads are reused with the different method calls, as in the following screenshot:

A screenshot of a Windows console window titled "C:\git\CH08_ThreadingAndConcurrenc...". The console displays the output of the `UsingTaskParallelLibraryFor` method. It shows 1000 lines of "Method Executed: Thread Id(6)" followed by "UsingTaskParallelLibraryFor Completed." at the bottom. The thread IDs are not all 6, but the screenshot shows a mix of IDs including 6, 5, 1, 4, 3, and 1, indicating thread reuse.

Now, we will look at using the `ThreadPool.QueueUserWorkItem()` method.

ThreadPool.QueueUserWorkItem()

The `ThreadPool.QueueUserWorkItem()` method accepts a `WaitCallback` method and queues it ready for execution. `WaitCallback` is a delegate that represents a callback method to be executed by a thread pool thread. When a thread becomes available, the method is executed. Let's add a simple example. We'll start by adding

`WaitCallbackMethod`:

```
private static void WaitCallbackMethod(Object _)
{
    Message("Hello from WaitCallBackMethod!");
}
```

This method accepts a type of object. However, since the parameter will be unused, we use the discard variable (`_`). A message is printed to the console window. Now, all we need is the code to call the method:

```
private static void ThreadPoolQueueUserWorkItem()
{
    ThreadPool.QueueUserWorkItem(WaitCallbackMethod);
    Message("Main thread does some work, then sleeps.");
    Thread.Sleep(1000);
    Message("Main thread exits.");
}
```

As you can see, we use the `ThreadPool` class to queue `WaitCallbackMethod()` in the thread pool via the call to the `QueueUserWorkItem()` method. We then do some work on the main thread. The main thread then goes to sleep. A thread becomes available from the thread pool and `WaitCallBackMethod()` is executed. The thread is then returned back to the thread pool to be reused. Execution returns to the main thread, which then completes and terminates.

In the next section, we will discuss thread-locking objects, known as **Mutual Exclusion Objects (mutexes)**.

Using a mutex with synchronous threads

In C#, a mutex is a thread-locking object that works across multiple processes. Only a process that can request or release a resource can modify the mutex. When a mutex is locked, the process will have to wait in a queue. When the mutex is unlocked, then it can be accessed. Multiple threads can use the same mutex, but only in a synchronous manner.

The benefits of using a mutex are that a mutex is a simple lock obtained before entering a critical piece of code. That lock is released when the critical piece of code is exited. Because only a single thread is in the critical piece of code at any one time, the data will remain in a consistent state as there will be no race conditions.

There are several disadvantages to using a mutex:

- Thread starvation occurs when a thread is unable to move forward as an existing thread has obtained a lock and has either gone to sleep or is pre-empted (prevented from completing its task).
- When a mutex is locked, only the thread that obtained the lock can unlock it. No other thread can lock or unlock it.
- Only one thread at a time is allowed to enter the critical piece of code. CPU time can be wasted as the normal implementation of a mutex may lead to a *busy waiting* state.

We will now write a program that demonstrates the use of a mutex. Start a new .NET Framework console application. Add the following line to the top of the class:

```
private static readonly Mutex _mutex = new Mutex();
```

Here, we have declared a primitive called `_mutex`, which we will use for inter-process synchronization. Now, add a method to demonstrate thread synchronization using a mutex:

```
private static void ThreadSynchronisationUsingMutex()
{
    try
    {
        _mutex.WaitOne();
        Message($"Domain Entered By: {Thread.CurrentThread.Name}");
        Thread.Sleep(500);
        Message($"Domain Left By: {Thread.CurrentThread.Name}");
    }
    finally
    {
        _mutex.ReleaseMutex();
    }
}
```

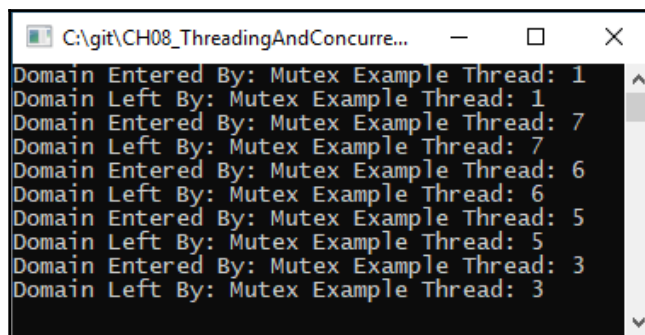
In this method, the current thread is blocked until the current wait handle receives a signal. Then, when the signal is given, it is safe for the next thread to enter. Upon completion, other threads are unblocked from trying to gain ownership of the mutex. Next, add the `MutexExample()` method:

```
private static void MutexExample()
{
    for (var i = 1; i <= 10; i++)
    {
        var thread = new Thread(ThreadSynchronisationUsingMutex)
        {
            Name = $"Mutex Example Thread: {i}"
        };
        thread.Start();
    }
}
```

In this method, we create 10 threads and start them. Each thread executes the `ThreadSynchronisationUsingMutex()` method. Now, finally, update the `Main()` method:

```
static void Main(string[] args)
{
    SemaphoreExample();
    Console.ReadKey();
}
```

The `Main()` method runs our mutex example. The output should be similar to the one in the following screenshot:



Run the example again and you may end up with different thread numbers. If they are the same numbers, then they may be in different orders.

Now that we have looked at mutexes, let's look at semaphores.

Working with parallel threads using semaphores

In multi-threaded applications, a non-negative number, known as a **semaphore**, is shared between threads that have a number of 1 or 2. In terms of synchronization, 1 specifies *wait* and 2 specifies *signal*. We can associate a semaphore with a number of buffers, which can each be worked on simultaneously by different processes.

So, essentially, semaphores are signaling mechanisms of the integer and binary primitive types that can be modified by wait and signal operations. If there are no free resources, then processes that require a resource should execute the wait operation until the semaphore value is *greater than 0*. Semaphores can have multiple program threads and they can be changed by any object, obtaining a resource or releasing it.

The advantages of using semaphores are down to the fact that more than one thread can access the critical piece of code. A semaphore is executed in the kernel and is machine-independent. The critical piece of code is protected from multiple processes if you use semaphores. Unlike a mutex, a semaphore never wastes processing time and resources.

Just like a mutex, semaphores also have their own set of disadvantages. Priority inversion is one of the biggest disadvantages and occurs when a high-priority thread is forced to wait for a semaphore to be released by its low-priority owning thread.

This can be further compounded if the low-priority thread is prevented from completing by mid-priority threads prior to their release. This is known as **unbounded priority inversion** because we can no longer predict the delay to the high-priority thread. With semaphores, the operating system must keep track of all wait and signal calls.

Semaphores are used by convention, but they are not forced. You need to execute wait and signal operations in the correct order; otherwise, you risk deadlocks in your code. Because of the complexity of using semaphores, there may be times when a mutual exclusion cannot be obtained. Loss of modularity in large systems is also another drawback and semaphores are prone to programming errors that result in deadlocks and mutual exclusion violation.

We're going to write a program now that demonstrates the use of semaphores:

```
private static readonly Semaphore _semaphore = new Semaphore(2, 4);
```

We have added a new semaphore variable. The first parameter states the initial number of requests for the semaphore that can be granted concurrently. The second parameter states the maximum number of requests for the semaphore that can be granted concurrently. Add the `StartSemaphore()` method:

```
private static void StartSemaphore(object id)
{
    Console.WriteLine($"Object {id} wants semaphore access.");
    try
    {
        _semaphore.WaitOne();
        Console.WriteLine($"Object {id} gained semaphore access.");
        Thread.Sleep(1000);
        Console.WriteLine($"Object {id} has exited semaphore.");
    }
    finally
    {
        _semaphore.Release();
    }
}
```

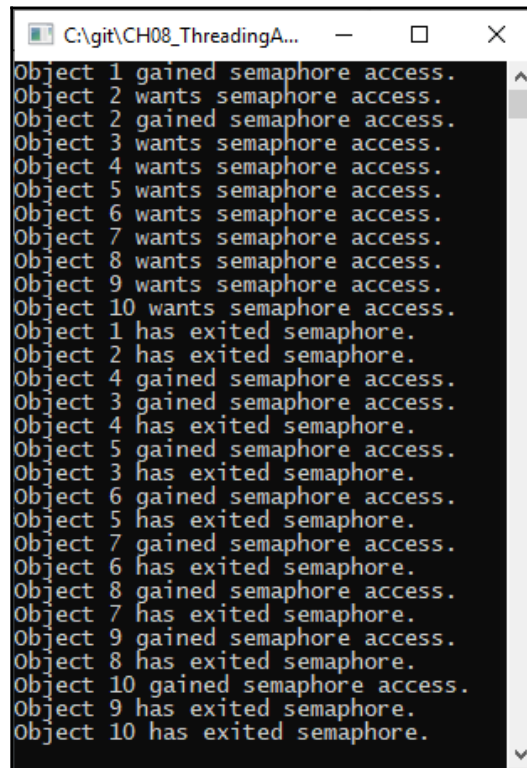
The current thread is blocked until the current wait handle receives a signal. The thread can then do its work. Finally, the semaphore is released and the count returns to the previous count. Now, add the `SemaphoreExample()` method:

```
private static void SemaphoreExample()
{
    for (int i = 1; i <= 10; i++)
    {
        Thread t = new Thread(StartSemaphore);
        t.Start(i);
    }
}
```

This example generates 10 threads, which execute the `StartSemaphore()` method. Let's update the `Main()` method to run the code:

```
static void Main(string[] args)
{
    SemaphoreExample();
    Console.ReadKey();
}
```

The `Main()` method calls `SemaphoreExample()` and then waits for a user keypress to exit. You should see the following output:



```
C:\git\CH08_ThreadingA...
Object 1 gained semaphore access.
Object 2 wants semaphore access.
Object 2 gained semaphore access.
Object 3 wants semaphore access.
Object 4 wants semaphore access.
Object 5 wants semaphore access.
Object 6 wants semaphore access.
Object 7 wants semaphore access.
Object 8 wants semaphore access.
Object 9 wants semaphore access.
Object 10 wants semaphore access.
Object 1 has exited semaphore.
Object 2 has exited semaphore.
Object 4 gained semaphore access.
Object 3 gained semaphore access.
Object 4 has exited semaphore.
Object 5 gained semaphore access.
Object 3 has exited semaphore.
Object 6 gained semaphore access.
Object 5 has exited semaphore.
Object 7 gained semaphore access.
Object 6 has exited semaphore.
Object 8 gained semaphore access.
Object 7 has exited semaphore.
Object 9 gained semaphore access.
Object 8 has exited semaphore.
Object 10 gained semaphore access.
Object 9 has exited semaphore.
Object 10 has exited semaphore.
```

Let's move on to look at how we limit the number of processors and threads in the thread pool.

Limiting the number of processors and threads in the thread pool

There may be times when you need to limit the number of processors and threads used by your computer program.

To reduce the number of processors that your program uses, you obtain the current process and set its processor affinity value. For example, say that we have a four-core computer and we want to limit our usage to the first two cores. The binary value for the first two cores is 11, which is 3 in integer form. Now, let's add a method to a new .NET Framework console application and call it `AssignCores()`:

```
private static void AssignCores(int cores)
{
    Process.GetCurrentProcess().ProcessorAffinity = new IntPtr(cores);
}
```

We pass in an integer to the method. This integer value will be converted into a binary value by .NET Framework. That binary value will use the processors identified by the value of 1. For binary values of 0, the processors will not be used. So, since machine code is represented by binary numbers, 0110 (6) will use cores 2 and 3, 1100 (3) will use cores 1 and 2, and 0011 (12) will use cores 3 and 4.



If you want a refresher on binary, refer to <https://www.computerhope.com/jargon/b/binary.htm>.

Now, to set the maximum number of threads, we call the `SetMaxThreads()` method on the `ThreadPool` class. This method takes two parameters, which are both integers. The first parameter is the maximum number of worker threads in the thread pool and the second parameter is the maximum number of asynchronous I/O threads in the thread pool. We'll now add our method to set the maximum number of threads:

```
private static void SetMaxThreads(int workerThreads, int asyncIoThreads)
{
    ThreadPool.SetMaxThreads(workerThreads, asyncIoThreads);
}
```



As you can see, it is pretty straightforward to set thread maximums and processors in your programs. Most of the time, you will not have to do this in your programs. The main reason for manually setting the number of threads and/or processors to use in your program is for if your programs run into performance issues. If your program does not experience performance issues, then it is best not to set the number of threads or the number of processors.

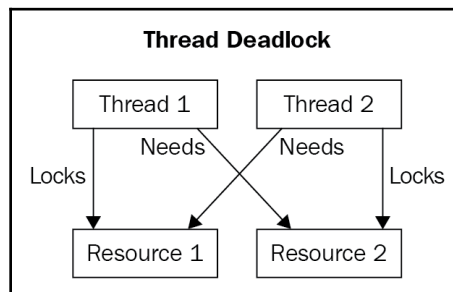
The next topic we will look at is deadlocks.

Preventing deadlocks

A **deadlock** occurs when two or more threads are executed and are waiting for each other to finish. This problem manifests in computer programs when they hang. For the end user, this can be very bad and can result in the loss or corruption of data. An example of this is executing two batches of data input that crash halfway through a transaction and cannot be rolled back. This is not good; let me explain why with an example.

Consider a major banking transaction that will take £1 million out of a customer's business bank account to pay their **Her Majesty's Revenue and Customs (HMRC)** tax bill. The money is taken from the business account, but before the money is deposited in the HMRC bank account, a deadlock occurs. There is no recovery option and so the application has to be terminated and restarted. As a result, the business bank account is reduced by £1 million but the HMRC tax bill has not been paid. The customer is still liable to pay the tax bill. But what happens to the money that has been taken out of the account? So, you can see the importance of removing the possibility of deadlocks occurring due to the problems they can cause.

To keep things simple, we will deal with two threads, shown in the following diagram:



We will call our threads **Thread 1** and **Thread 2** and our resources **Resource 1** and **Resource 2**. **Thread 1** obtains a lock on **Resource 1**. **Thread 2** obtains a lock on **Resource 2**. **Thread 1** requires access to **Resource 2** but has to wait because **Thread 2** has locked **Resource 2**. **Thread 2** requires access to **Resource 1** but has to wait because **Thread 1** has locked **Resource 1**. This results in both **Thread 1** and **Thread 2** being in a wait state. Since neither thread can continue until the other thread releases its resource, both threads are in a **deadlock situation**. When a computer program is in a deadlock situation, it *hangs*, forcing you to terminate the program.

A code example of a deadlock will be a nice way to illustrate this, and so in the next section, we will code a deadlock example.

Coding a deadlock example

The best way to understand this is with a working example. We are going to write some code consisting of two methods that have two different locks each. They will both lock objects that the other method needs. Because each thread locks the resources that the other thread needs, they will both enter a deadlock state. Once we have our example working, we will then modify it so that our code recovers from the deadlock situation and is able to continue.

Create a new .NET Framework console application and call it CH08_Deadlocks. We will need two objects as member variables, so let's add them:

```
static object _object1 = new object();
static object _object2 = new object();
```

These objects will be used as our lock objects. We will have two threads, and each thread will execute its own method. Now, add `Thread1Method()` to your code:

```
private static void Thread1Method()
{
    Console.WriteLine("Thread1Method: Thread1Method Entered.");
    lock (_object1)
    {
        Console.WriteLine("Thread1Method: Entered _object1 lock.
Sleeping...");
        Thread.Sleep(1000);
        Console.WriteLine("Thread1Method: Woke from sleep");
        lock (_object2)
        {
            Console.WriteLine("Thread1Method: Entered _object2 lock.");
        }
        Console.WriteLine("Thread1Method: Exited _object2 lock.");
    }
    Console.WriteLine("Thread1Method: Exited _object1 lock.");
}
```

`Thread1Method()` obtains a lock on `_object1`. It then sleeps for 1 second. When it awakes, a lock is obtained on `_object2`. The method then exits both locks and terminates.

`Thread2Method()` obtains a lock on `_object2`. It then sleeps for 1 second. When it awakes, a lock is obtained on `_object1`. The method then exits both locks and terminates:

```
private static void Thread2Method()
{
    Console.WriteLine("Thread2Method: Thread1Method Entered.");
    lock (_object2)
    {
        Console.WriteLine("Thread2Method: Entered _object2 lock.
Sleeping...");
        Thread.Sleep(1000);
        Console.WriteLine("Thread2Method: Woke from sleep.");
        lock (_object1)
        {
            Console.WriteLine("Thread2Method: Entered _object1 lock.");
        }
        Console.WriteLine("Thread2Method: Exited _object1 lock.");
    }
    Console.WriteLine("Thread2Method: Exited _object2 lock.");
}
```

Well, we now have our two methods in place to demonstrate a deadlock. We just need the code to call them in a way that will cause a deadlock. Let's add the `DeadlockNoRecovery()` method:

```
private static void DeadlockNoRecovery()
{
    Thread thread1 = new Thread((ThreadStart)Thread1Method);
    Thread thread2 = new Thread((ThreadStart)Thread2Method);

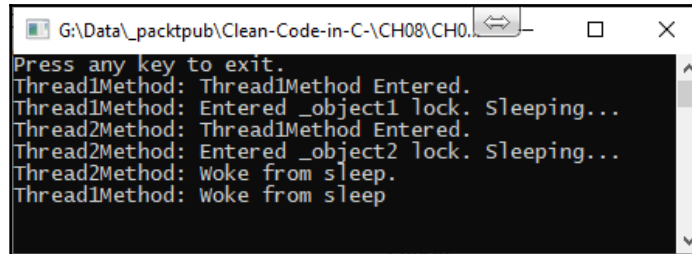
    thread1.Start();
    thread2.Start();

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
```

In the `DeadlockNoRecovery()` method, we create two threads. Each thread is assigned a different method. Then, each thread is started. The program is then paused until the user presses a key. Now, update the `Main()` method and run your code:

```
static void Main()
{
    DeadlockNoRecovery();
}
```

When you run your program, you should see the following output:



```
G:\Data\_packtpub\Clean-Code-in-C-\CH08\CH08
Press any key to exit.
Thread1Method: Thread1Method Entered.
Thread1Method: Entered _object1 lock. Sleeping...
Thread2Method: Thread1Method Entered.
Thread2Method: Entered _object2 lock. Sleeping...
Thread2Method: Woke from sleep.
Thread1Method: Woke from sleep
```

As you can see, because thread1 has locked `_object1`, thread2 is blocked from obtaining a lock on `_object1`. Also, because thread2 has locked `_object2`, thread1 is blocked from obtaining a lock on `_object2`. So, both threads are in deadlock and the program hangs.

We will now write some code that demonstrates how to avoid this deadlock situation from occurring. We will be using the `Monitor.TryLock()` method to try and obtain a lock within a certain number of milliseconds. We will then exit a successful lock with `Monitor.Exit()`.

Now, add the `DeadlockWithRecovery()` method:

```
private static void DeadlockWithRecovery()
{
    Thread thread4 = new Thread((ThreadStart)Thread4Method);
    Thread thread5 = new Thread((ThreadStart)Thread5Method);

    thread4.Start();
    thread5.Start();

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
```

The `DeadlockWithRecovery()` method creates two foreground threads. It then starts the threads, prints a message to the console, and waits for the user to press a key before exiting. We will now add the code for `Thread4Method()`:

```
private static void Thread4Method()
{
    Console.WriteLine("Thread4Method: Entered _object1 lock.
Sleeping...");
    Thread.Sleep(1000);
    Console.WriteLine("Thread4Method: Woke from sleep");
}
```

```
        if (!Monitor.TryEnter(_object1))
        {
            Console.WriteLine("Thread4Method: Failed to lock _object1.");
            return;
        }
        try
        {
            if (!Monitor.TryEnter(_object2))
            {
                Console.WriteLine("Thread4Method: Failed to lock _object2.");
                return;
            }
            try
            {
                Console.WriteLine("Thread4Method: Doing work with _object2.");
            }
            finally
            {
                Monitor.Exit(_object2);
                Console.WriteLine("Thread4Method: Released _object2 lock.");
            }
        }
        finally
        {
            Monitor.Exit(_object1);
            Console.WriteLine("Thread4Method: Released _object1 lock.");
        }
    }
```

`Thread4Method()` sleeps for 1 second. It then tries to get a lock on `_object1`. If it fails to get a lock on `_object1`, it returns from the method. If a lock on `_object1` is obtained, then it tries to get a lock on `_object2`. If a lock on `_object2` can't be obtained, then it returns from the method. If a lock is obtained on `_object2`, then it performs the necessary work on `_object2`. The lock on `_object2` is then released, and then the lock on `_object1` is released.

Our `Thread5Method()` method does exactly the same thing, except the objects—`_object1` and `_object2`—are locked in reverse order:

```
private static void Thread5Method()
{
    Console.WriteLine("Thread5Method: Entered _object2 lock.
Sleeping...");
    Thread.Sleep(1000);
    Console.WriteLine("Thread5Method: Woke from sleep");
    if (!Monitor.TryEnter(_object2))
    {

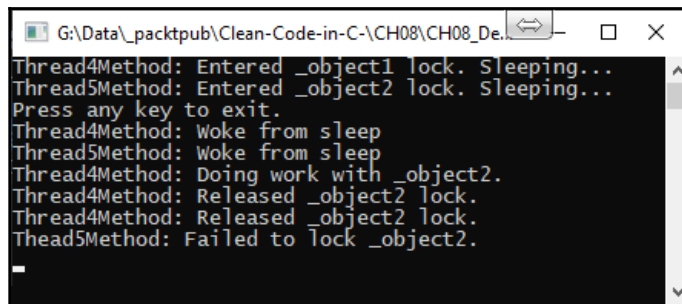
```

```
        Console.WriteLine("Thead5Method: Failed to lock _object2.");
        return;
    }
    try
    {
        if (!Monitor.TryEnter(_object1))
        {
            Console.WriteLine("Thread5Method: Failed to lock _object1.");
            return;
        }
        try
        {
            Console.WriteLine("Thread5Method: Doing work with _object1.");
        }
        finally
        {
            Monitor.Exit(_object1);
            Console.WriteLine("Thread5Method: Released _object1 lock.");
        }
    }
    finally
    {
        Monitor.Exit(_object2);
        Console.WriteLine("Thread5Method: Released _object2 lock.");
    }
}
```

Now, add the `DeadlockWithRecovery()` method call to your `Main()` method:

```
static void Main()
{
    DeadlockWithRecovery();
}
```

Then, run your code a few times. The majority of the time, you will see what is in the following screenshot, where all the locks have been successfully obtained:



```
G:\Data\_packtpub\Clean-Code-in-C-\CH08\CH08_De...
Thread4Method: Entered _object1 lock. Sleeping...
Thread5Method: Entered _object2 lock. Sleeping...
Press any key to exit.
Thread4Method: Woke from sleep
Thread5Method: Woke from sleep
Thread4Method: Doing work with _object2.
Thread4Method: Released _object2 lock.
Thread4Method: Released _object2 lock.
Thead5Method: Failed to lock _object2.
_
```

Then, press any key and the program will exit. If you keep running the program, you will eventually find a lock that fails. The program failed to get a lock on `_object2` in `Thread5Method()`. However, if you press any key, the program will exit. As you can see, by using `Monitor.TryEnter()`, you can try and lock an object. But if the lock is not obtained, then you are able to take another action without your program hanging.

In the next section, we look at preventing race conditions.

Preventing race conditions

When multiple threads using the same resource produce different outcomes due to the timings of each thread, this is known as a **race condition**. We will demonstrate this in action now.

In our demonstration, we will have two threads. Each thread will call a method to print the alphabet. One method will print the alphabet using *uppercase letters*. The second method will print the alphabet using *lowercase letters*. From the demonstration, we'll see how the output is wrong, and every time the program is run, the output will be wrong.

First, add the `ThreadingRaceCondition()` method:

```
static void ThreadingRaceCondition()
{
    Thread T1 = new Thread(Method1);
    T1.Start();
    Thread T2 = new Thread(Method2);
    T2.Start();
}
```

`ThreadingRaceCondition()` produces two threads and starts them. It also references two methods. `Method1()` prints out the alphabet in uppercase and `Method2()` prints out the alphabet in lowercase. Let's add `Method1()` and `Method2()`:

```
static void Method1()
{
    for (_alphabetCharacter = 'A'; _alphabetCharacter <= 'Z';
        _alphabetCharacter++)
    {
        Console.Write(_alphabetCharacter + " ");
    }
}

private static void Method2()
```



```

{
    for (_alphabetCharacter = 'a'; _alphabetCharacter <= 'z';
        _alphabetCharacter++)
    {
        Console.Write(_alphabetCharacter + " ");
    }
}

```

Both `Method1()` and `Method2()` reference the `_alphabetCharacter` variable. So, add the member to the top of the class:

```
private static char _alphabetCharacter;
```

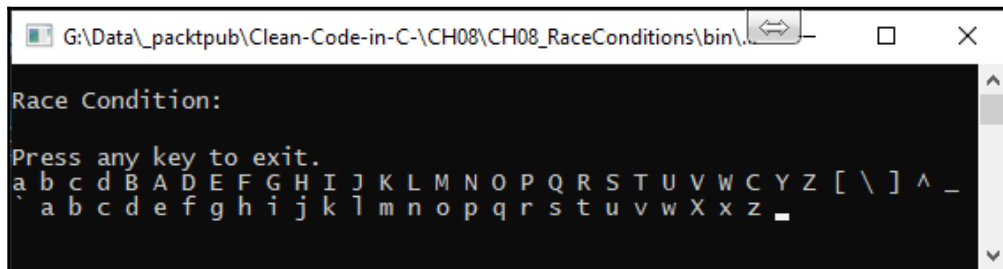
Now, update `MainMethod()`:

```

static void Main(string[] args)
{
    Console.WriteLine("\n\nRace Condition:");
    ThreadingRaceCondition();
    Console.WriteLine("\n\nPress any key to exit.");
    Console.ReadKey();
}

```

We now have our code in place to demonstrate the race condition. If you run the program multiple times, you will see that the results are not what we expect. You should even see characters that are not part of the alphabet:



Not exactly what we were expecting, is it?

We are going to solve this problem by using the TPL. The aim of the TPL is to simplify **parallelism** and **concurrency**. As most computers today have two or more processors, the TPL will scale the degree of concurrency dynamically to make the most efficient use of all the available processors.



The partitioning of work, the scheduling of threads in the thread pool, cancellation support, state management, and so on are also carried out by the TPL. A link to the official Microsoft TPL documentation can be found in the *Further reading* section of this chapter.

You will see just how simple the solution to the aforementioned problem can be. We have a task that runs `Method1()`. The task then continues with `Method2()`. We then call `Wait()` to wait for the task to complete execution. Now, add the `ThreadingRaceConditionFixed()` method to your source code:

```
static void ThreadingRaceConditionFixed()
{
    Task
        .Run(() => Method1())
        .ContinueWith(task => Method2())
        .Wait();
}
```

Modify your `Main()` method, as follows:

```
static void Main(string[] args)
{
    //Console.WriteLine("\n\nRace Condition:");
    //ThreadingRaceCondition();
    Console.WriteLine("\n\nRace Condition Fixed:");
    ThreadingRaceConditionFixed();
    Console.WriteLine("\n\nPress any key to exit.");
    Console.ReadKey();
}
```

Run the code now. If you run it multiple times, you will see that the output is always the same, as in the following screenshot:

So far, we have seen what a thread is and how to use them in the foreground and background. We have also looked at deadlocks and how to solve them with `Monitor.TryEnter()`. Finally, we looked at what race conditions are and how to solve them using the TPL.

Now, we will move on to looking at static constructors and methods.

Understanding static constructors and methods

If multiple classes require access to a property instance simultaneously, then one of the threads will be requested to run the **static constructor** (also known as the **type initializer**). While waiting for the type initializer to run, all the other threads will be locked. Once the type initializer has run, the locked threads are unlocked and are able to access the `Instance` property.

Static constructors are thread-safe as they are guaranteed to run only once per application domain. They are executed before accessing any static members and before any class instantiation is performed.



Should an exception be raised in and escape from a static constructor, then `TypeInitializationException` is generated, which causes the CLR to exit your program.

Before any threads can access a class, static initializers and static constructors must finish executing.

Static methods only keep a single copy of the method and its data at the type level. This means that the same method and its data will be shared between different instances. Each thread in an application has its own stack. Value types passed into static methods are created on the calling thread's stack, and so they are thread-safe. This means that if two threads call the same code and pass the same value in, there will be two copies of that value—one on each thread's stack. So, multiple threads will not affect each other.

However, if you have a static method that accesses a member variable, then it is not thread-safe. Two different threads call the same method and so both will have access to the member variable. A process or context-switching occurs between threads; each thread will access and modify the member variable. This leads to race conditions, as you saw earlier in this chapter.

You also run into problems if you pass reference types into a static method, as different threads will have access to the same reference type. This also causes a race condition.



When working with static methods that will be used across threads, avoid member variable access and do not pass reference types in. Static methods are thread-safe as long as you pass in primitive types and don't modify the state.

Now that we've discussed static constructors and methods, we will run through some example code.

Adding static constructors to our sample code

Start a new .NET Framework console application. Add a class called `StaticConstructorTestClass` to the project. Then, add a read-only static string variable called `_message`:

```
public class StaticConstructorTestClass
{
    private readonly static string _message;
}
```

The `_message` variable is returned to the caller by the `Message()` method. Let's write the `Message()` method now:

```
public static string Message()
{
    return $"Message: {_message}";
}
```

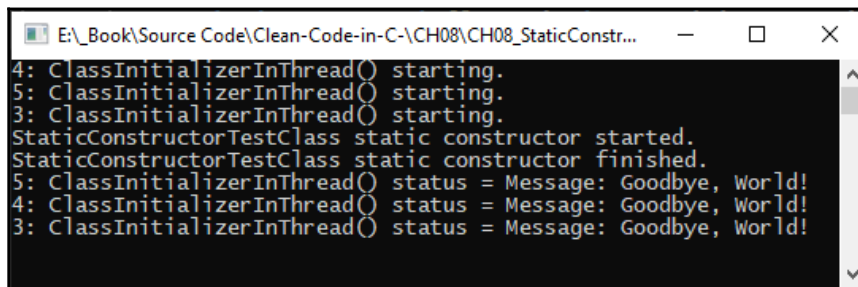
This method returns the message stored in the `_message` variable. Now, we need to write our constructor:

```
static StaticConstructorTestClass()
{
    Console.WriteLine("StaticConstructorTestClass static constructor started.");
    _message = "Hello, World!";
    Thread.Sleep(1000);
    _message = "Goodbye, World!";
    Console.WriteLine("StaticConstructorTestClass static constructor finished.");
}
```

In our constructor, we write a message to the screen. We then set the member variable and let the thread sleep for a second. Then, we set the message again and write another message to the console. Now, in the `Program` class, update the `Main()` method, as follows:

```
static void Main(string[] args)
{
    var program = new Program();
    program.StaticConstructorExample();
    Thread.CurrentThread.Join();
}
```

Our `Main()` method instantiates the `Program` class. The `StaticConstructorExample()` method is then called. When the program halts and we can see the result, we join threads. You can see the output in the following screenshot:



```
4: ClassInitializerInThread() starting.
5: ClassInitializerInThread() starting.
3: ClassInitializerInThread() starting.
StaticConstructorTestClass static constructor started.
StaticConstructorTestClass static constructor finished.
5: ClassInitializerInThread() status = Message: Goodbye, World!
4: ClassInitializerInThread() status = Message: Goodbye, World!
3: ClassInitializerInThread() status = Message: Goodbye, World!
```

We'll now take a look at examples of static methods.

Adding static methods to our sample code

We are now going to look at thread-safe static methods and non-thread-safe methods in action. Add a new class called `StaticExampleClass` to a new .NET Framework console application. Then, add the following code:

```
public static class StaticExampleClass
{
    private static int _x = 1;
    private static int _y = 2;
    private static int _z = 3;
}
```

At the top of our class, we add three integers—`_x`, `_y`, and `_z`—with values of 1, 2, and 3, respectively. These variables can be modified between threads. Now, we will add a static constructor to print out the values of these variables:

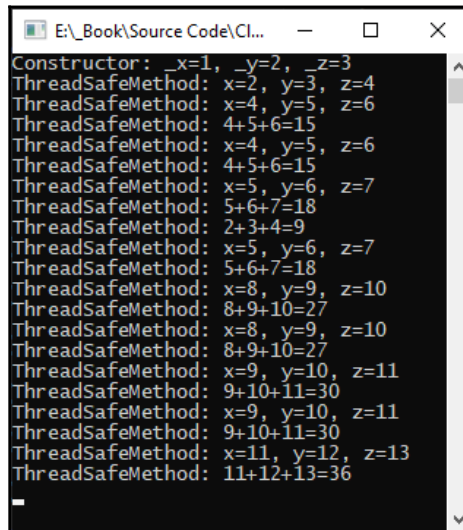
```
static StaticExampleClass()
{
    Console.WriteLine($"Constructor: _x={_x}, _y={_y}, _z={_z}");
}
```

As you can see, the static constructor simply prints out the values of the variables to the console window. Our first method will be a thread-safe method called `ThreadSafeMethod()`:

```
internal static void ThreadSafeMethod(int x, int y, int z)
{
    Console.WriteLine($"ThreadSafeMethod: x={x}, y={y}, z={z}");
    Console.WriteLine($"ThreadSafeMethod: {x}+{y}+{z}={x+y+z}");
}
```

This method is thread-safe because it only operates on by value parameters. It does not interact with the member variables and does not include any by reference values. So, no matter what values are passed in, you will always get the expected result.

This means that regardless of whether only a single thread or even millions of threads are accessing the method, the output for each thread will be what you expect when you pass in the input values, even despite context switching. The following screenshot shows the output:



```
Constructor: _x=1, _y=2, _z=3
ThreadSafeMethod: x=2, y=3, z=4
ThreadSafeMethod: x=4, y=5, z=6
ThreadSafeMethod: 4+5+6=15
ThreadSafeMethod: x=4, y=5, z=6
ThreadSafeMethod: 4+5+6=15
ThreadSafeMethod: x=5, y=6, z=7
ThreadSafeMethod: 5+6+7=18
ThreadSafeMethod: 2+3+4=9
ThreadSafeMethod: x=5, y=6, z=7
ThreadSafeMethod: 5+6+7=18
ThreadSafeMethod: x=8, y=9, z=10
ThreadSafeMethod: 8+9+10=27
ThreadSafeMethod: x=8, y=9, z=10
ThreadSafeMethod: 8+9+10=27
ThreadSafeMethod: x=9, y=10, z=11
ThreadSafeMethod: 9+10+11=30
ThreadSafeMethod: x=9, y=10, z=11
ThreadSafeMethod: 9+10+11=30
ThreadSafeMethod: x=11, y=12, z=13
ThreadSafeMethod: 11+12+13=36
```

Now that we have looked at thread-safe methods, it is only right that we look at non-thread-safe methods. By now, you know that a static method that operates on by reference values or static member variables is not thread-safe.

In our next example, we will use a method with the same three parameters as `ThreadSafeMethod()`, but this time, we will set the member variables, output a message, go to sleep for a while, and then awake to print the values out again. Add the following `NotThreadSafeMethod()` method to `StaticExampleClass`:

```
internal static void NotThreadSafeMethod(int x, int y, int z)
{
    _x = x;
    _y = y;
    _z = z;
    Console.WriteLine(
        $"{Thread.CurrentThread.ManagedThreadId}-NotThreadSafeMethod:
_x={_x}, _y={_y}, _z={_z}"
    );
    Thread.Sleep(300);
    Console.WriteLine(
        $"{Thread.CurrentThread.ManagedThreadId}-ThreadSafeMethod:
{_x}+{_y}+{_z}={_x + _y + _z}"
    );
}
```

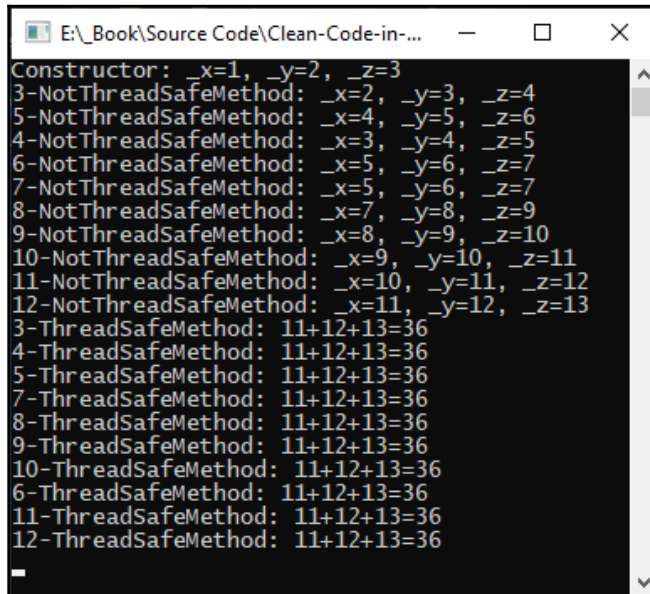
In this method, we set the member variables to the values passed into the method. We then output those values to the console windows and go to sleep for 300 milliseconds. Then, upon waking from our sleep, we print the values out again. In the `Program` class, update the `Main()` method, as shown:

```
static void Main(string[] args)
{
    var program = new Program();
    program.ThreadUnsafeMethodCall();
    Console.ReadKey();
}
```

In the `Main()` method, we instantiate the program class, call `ThreadUnsafeMethodCall()`, and then wait for the user to press a key before exiting. So, let's add `ThreadUnsafeMethodCall()` to the `Program` class:

```
private void ThreadUnsafeMethodCall()
{
    for (var index = 0; index < 10; index++)
    {
        var thread = new Thread(() =>
        {
            StaticExampleClass.NotThreadSafeMethod(index + 1, index + 2,
index + 3);
        });
        thread.Start();
    }
}
```

This method produces 10 threads that call `NotThreadSafeMethod()` of `StaticExampleClass`. If you run the code, you will see an output similar to that in the following screenshot:



As you can see, the output is not what we would expect. This is because of the pollution from different threads. This leads us nicely to the next section on mutability, immutability, and thread safety.

Mutability, immutability, and thread safety

Mutability is a source of bugs in multi-threaded applications. A mutable bug is normally a data bug caused by values being updated and shared between threads. To remove the risk of mutability bugs, it is best to use **immutable types**. The guaranteed safe execution of a body of code by multiple threads at the same time is called **thread safety**. When working with multi-threaded programs, it is important that your code is thread-safe. Your code is thread-safe if it removes race conditions and deadlocks, along with problems caused by mutability.

An object that cannot be modified after it has been created is an **immutable object**. Once created, if passed between threads using correct thread synchronization, all threads will see the same valid state of an object. Immutable objects allow you to share data safely between threads.

An object that can be modified after it has been created is a mutable object. Mutable objects can have their data values changed between threads. This can lead to some serious data corruption. So, even if the program does not crash, it can leave the data in an invalid state. Therefore, when working with multiple threads of execution, it is important that your objects are immutable. In *Chapter 3, Classes, Objects, and Data Structures*, we went through creating and using immutable data structures for your immutable objects.



To ensure thread safety, do not use mutable objects, pass parameters by reference, or modify member variables—only pass parameters by value and only operate on parameter variables. Do not access member variables. Immutable structures are a good and thread-safe way to pass data between objects.

We will take a brief look at mutability, immutability, and thread safety with the following examples. We'll start with mutability in terms of thread safety.

Writing code that is mutable and not thread-safe

To demonstrate mutability within a multi-threaded application, we will write a new .NET Framework console application. Add a new class to the application called `MutableClass` with the following code:

```
internal class MutableClass
{
    private readonly int[] _intArray;

    public MutableClass(int[] intArray)
```

```
{
    _intArray = intArray;
}

public int[] GetIntArray()
{
    return _intArray;
}
}
```

In our `MutableClass` class, we have a constructor that takes an integer array as an argument. A member integer array is then assigned the array passed into the constructor. The `GetIntArray()` method returns the integer array member variable. If you look at this class, you would not think it is mutable because once the array is passed into the constructor, the class provides no way to modify it. Yet, the integer array passed into the constructor is mutable. The `GetIntArray()` method returns a reference to the mutable array.

In our `Program` class, we will add the `MutableExample()` method to show that the integer array is mutable:

```
private static void MutableExample()
{
    int[] iar = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    var mutableClass = new MutableClass(iar);

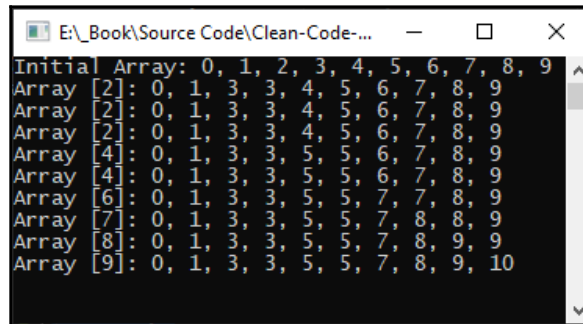
    Console.WriteLine($"Initial Array: {iar[0]}, {iar[1]}, {iar[2]},
{iar[3]}, {iar[4]}, {iar[5]}, {iar[6]}, {iar[7]}, {iar[8]}, {iar[9]}");

    for (var x = 0; x < 9; x++)
    {
        var thread = new Thread(() =>
        {
            iar[x] = x + 1;
            var ia = mutableClass.GetIntArray();
            Console.WriteLine($"Array [{x}]: {ia[0]}, {ia[1]}, {ia[2]},
{ia[3]}, {ia[4]}, {ia[5]}, {ia[6]}, {ia[7]}, {ia[8]}, {ia[9]}");
        });
        thread.Start();
    }
}
```

In our `MutableExample()` method, we have declared and initiated an integer array of items from 0 to 9. We then declare a new instance of `MutableClass` and pass in the integer array. Next, we print out the contents of the initial array before it is modified. Then, we loop nine times. For each iteration, we increase the array at the index specified by the current loop count value of `x` so that it equals `x + 1`. After that, we start the thread. Now, update the `Main()` method, as follows:

```
static void Main(string[] args)
{
    MutableExample();
    Console.ReadKey();
}
```

Our `Main()` method simply calls `MutableExample()` and then waits for a keypress. Run the code and you should see something as in the following screenshot:



As you can see, even though we only created one instance of `MutableClass` before creating and running our threads, changing the local array modifies the array in the instance of `MutableClass`. This proves that the arrays are mutable, and so they are not thread-safe.

We will now look at immutability in terms of thread safety.

Writing code that is immutable and thread-safe

In our immutability example, we will again create a .NET Framework console application and we'll use the same array. Add a class called `ImmutableStruct` and modify the code, as shown:

```
internal struct ImmutableStruct
{
```

```

private ImmutableArray<int> _immutableArray;

public ImmutableStruct(ImmutableArray<int> immutableArray)
{
    _immutableArray = immutableArray;
}

public int[] GetIntArray()
{
    return _immutableArray.ToArray<int>();
}
}

```

Instead of using a normal integer array, we employ `ImmutableArray`. An immutable array is passed into the constructor and assigned to the `_immutableArray` member variable. Our `GetIntArray()` method returns the immutable array as a normal integer array.

Add the `ImmutableExample()` array to the `Program` class:

```

private static void ImmutableExample()
{
    int[] iar = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    var immutableStruct = new ImmutableStruct(iar.ToImmutableArray<int>());

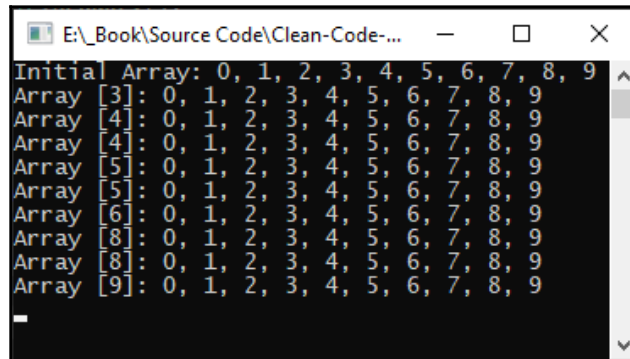
    Console.WriteLine($"Initial Array: {iar[0]}, {iar[1]}, {iar[2]},
{iar[3]}, {iar[4]}, {iar[5]}, {iar[6]}, {iar[7]}, {iar[8]}, {iar[9]}");

    for (var x = 0; x < 9; x++)
    {
        var thread = new Thread(() =>
        {
            iar[x] = x + 1;
            var ia = immutableStruct.GetIntArray();
            Console.WriteLine($"Array [{x}]: {ia[0]}, {ia[1]}, {ia[2]},
{ia[3]}, {ia[4]}, {ia[5]}, {ia[6]}, {ia[7]}, {ia[8]}, {ia[9]}");
        });
        thread.Start();
    }
}

```

In our `ImmutableExample()` method, we create an array of integers that we pass into the constructor of `ImmutableStruct` as an immutable array. We then print the content of the local array before modification. Then, we loop nine times. In each iteration, we access the location of the count of the current iteration in the array and add the count of the current iteration plus one to the variable at that position in the array.

We then assign a copy of the `immutableStruct` array to a local variable via a call to `GetIntArray()`. Then, we proceed to print out the values of the returned array. Finally, we start the thread. Call the `ImmutableExample()` method from your `Main()` method and then run the code. You should see the following output:

A screenshot of a Windows console window with a black background and white text. The window title bar shows the file path 'E:_Book\Source Code\Clean-Code-...'. The output text is as follows:

```
Initial Array: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Array [3]: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Array [4]: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Array [4]: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Array [5]: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Array [5]: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Array [6]: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Array [8]: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Array [8]: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Array [9]: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

As you can see, the array's content is not modified by updating the local array. This version of our program shows that our program is thread-safe.

Let's briefly run through what we've learned about thread safety so far in the next section.

Understanding thread safety

As you saw in the previous two sections, it is very important to be careful when writing multi-threaded code. Writing thread-safe code can be very difficult, especially in larger projects. You have to be particularly careful with collections, passing parameters by reference, and when accessing member variables within static classes. The best practices for multi-threaded applications are to only pass immutable types, not to access static member variables, and if any code that is not thread-safe must be executed, then to lock the code using a lock, mutex, or semaphore. Although you have already seen code like this in action in this chapter, we will quickly refresh our memory on this with some code snippets.

The following code snippet shows how to write an immutable type using `readonly struct`:

```
public readonly struct ImmutablePerson
{
    public ImmutablePerson(int id, string firstName, string lastName)
    {
        _id = id;
        _firstName = firstName;
        _lastName = lastName;
    }

    public int Id { get; }
    public string FirstName { get; }
    public string LastName { get { return _lastName; } }
}
```

In our `ImmutablePerson` structure, we have a public constructor that takes an integer for the ID and strings for the first and last name. We assign the `id`, `firstName`, and `lastName` parameters to member read-only variables. The only access to the data is via read-only properties. This means that there is no way to modify the data. Since the data cannot be modified once it has been created, it is classed as thread-safe. Because it is thread-safe, it cannot be modified by different threads. The only way to modify the data would be to create a new struct with the new data.



Structs can be mutable, just like classes. However, to pass data around that you don't want to be modified, then read-only structs are a good, lightweight choice. They are faster to create and destroy than classes as they are added to the stack—that is, unless they are part of a class that is added to the heap.

Earlier on, we saw how collections are mutable. However, there is also a namespace of immutable collections called `System.Collections.Namespace`. The following table lists various items from this namespace:

Classes	Structs	Interfaces
ImmutableArray	ImmutableArray<T>.Enumerator	IImmutableDictionary<TKey,TValue>
ImmutableArray<T>.Builder	ImmutableArray<T>	IImmutableList<T>
ImmutableDictionary	ImmutableDictionary<TKey,TValue>.Enumerator	IImmutableQueue<T>
ImmutableDictionary<TKey,TValue>.Builder	ImmutableHashSet<T>.Enumerator	IImmutableSet<T>
ImmutableDictionary<TKey,TValue>	ImmutableList<T>.Enumerator	IImmutableStack<T>
ImmutableHashSet	ImmutableQueue<T>.Enumerator	
ImmutableHashSet<T>.Builder	ImmutableSortedDictionary<TKey,TValue>.Enumerator	
ImmutableHashSet<T>	ImmutableSortedSet<T>.Enumerator	
ImmutableInterlocked	ImmutableStack<T>.Enumerator	
ImmutableList		
ImmutableList<T>.Builder		
ImmutableList<T>		
ImmutableQueue		
ImmutableQueue<T>		
ImmutableSortedDictionary		
ImmutableSortedDictionary<TKey,TValue>.Builder		
ImmutableSortedList		
ImmutableSortedList<T>.Builder		
ImmutableStack		
ImmutableStack<T>		



The `System.Collections.Immutable` namespace contains a number of immutable collections that you can use safely between threads. Refer to <https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable?view=netcore-3.1> for more details.

Using a lock object in C# is really straightforward, as the following code snippet shows:

```
public class LockExample
{
    public object _lock = new object();

    public void UnsafeMethod()
    {
        lock(_lock)
        {
```

```
        // Execute unsafe code.
    }
}
```

We create and instantiate the `_lock` member variable. Then, when it comes to executing code that is not thread-safe, we wrap the code in the lock and pass in the `_lock` variable to use as the lock object. When a thread enters the lock, all other threads are barred from executing the code until the thread leaves the lock. One problem with using this code is that threads can enter a deadlock situation. One way around this is to use a mutex.

You can use a synchronization primitive for interprocess synchronization. Start by adding the following code to the top of the class that has code that needs protection:

```
private static readonly Mutex _mutex = new Mutex();
```

Then, to use the mutex, you will need to wrap the code that needs protection with the following `try/catch` block:

```
try
{
    _mutex.WaitOne();
    // ... Do work here ...
}
finally
{
    _mutex.ReleaseMutex();
}
```

In the preceding code, the `WaitOne()` method blocks the current thread until the wait handle receives a signal. As soon as the mutex is signaled, the `WaitOne()` method returns `true`. The calling thread then assumes ownership of the mutex. Protected resources can then be accessed by the calling thread. When the work is finished on the protected resource, the mutex is released by calling `ReleaseMutex()`. `ReleaseMutex()` is called in the `finally` block because you don't want a thread to keep a resource locked if it raises an exception for whatever reason. So, always release a mutex in a `finally` block.

Another mechanism for protecting access to resources is using a semaphore. Semaphores are coded much like a mutex and they perform the same role of protecting resources. The main difference between a semaphore and a mutex is that a mutex is a locking mechanism and a semaphore is a signaling mechanism. To use semaphores instead of locks and mutexes, add the following line to the top of a class:

```
private static readonly Semaphore _semaphore = new Semaphore(2, 4);
```

We have now added a new semaphore variable. The first parameter states the initial number of requests for the semaphore that can be granted concurrently. The second parameter states the maximum number of requests for the semaphore that can be granted concurrently. You will then protect access to a resource in your methods, as follows:

```
try
{
    _semaphore.WaitOne();
    // ... Do work here ...
}
finally
{
    _semaphore.Release();
}
```

The current thread is blocked until the current wait handle receives a signal. The thread can then do its work. Finally, the semaphore is released.

You have seen, in this chapter, how to use locks, mutexes, and semaphores to lock code that is not thread-safe. Also remember that background threads terminate when the process completes and terminates, whereas foreground threads will continue executing until completion. If you have any code that must run to completion without the thread being terminated halfway through what it is doing, then you are better off using foreground threads over background threads.

The next section covers synchronized method dependencies.

Synchronized method dependencies

To synchronize your code, use a lock statement as we did previously. You can also reference the `System.Runtime.CompilerServices` namespace in your projects. Then, you can add the `[MethodImpl(MethodImplOptions.Synchronized)]` annotation to methods and properties.

Here is an example of the `[MethodImpl(MethodImplOptions.Synchronized)]` annotation applied to a method:

```
[MethodImpl(MethodImplOptions.Synchronized)]
public static void ThisIsASynchronisedMethod()
{
    Console.WriteLine("Synchronised method called.");
}
```

Here is an example of using `[MethodImpl(MethodImplOptions.Synchronized)]` with a property:

```
private int i;
public int SomeProperty
{
    [MethodImpl(MethodImplOptions.Synchronized)]
    get { return i; }
    [MethodImpl(MethodImplOptions.Synchronized)]
    set { i = value; }
}
```

As you can see, it is easy to encounter a deadlock or a race condition, but it is just as easy to overcome deadlocks by using `Monitor.TryEnter()` and race conditions with `Task.ContinueWith()`.

In the next section, we look at the `Interlocked` class.

Using the Interlocked class

In multi-threaded applications, errors can creep in during the thread scheduler context-switching process. One of the main problems that arises is the update of the same variables by different threads. The methods of the `System.Threading.Interlocked` class in the `mscorlib` assembly help to protect against these kinds of errors. The methods of the `Interlocked` class do not throw exceptions, and so they are very helpful in applying simple state changes in a more performant way than using the `lock` statement that we've seen previously.

The methods available in the `Interlocked` class are as follows:

- **CompareExchange:** Compares two variables and stores the results in a different variable
- **Add:** Adds two `Int32` or `Int64` integer variables together and stores the result in the first integer
- **Decrement:** Decrements the `Int32` and `Int64` integer variable values and stores their results
- **Increment:** Increments the `Int32` and `Int64` integer variable values and stores their results
- **Read:** Reads integer variables of the `Int64` type
- **Exchange:** Exchanges values between variables

We are now going to write a simple console application that demonstrates these methods. Start by creating a new .NET Framework console application. Add the following lines to the top of the `Program` class:

```
private static long _value = long.MaxValue;
private static int _resourceInUse = 0;
```

The `_value` variable will be used to demonstrate the update of variables using the interlocking methods. The `_resourceInUse` variable is used to indicate whether a resource is in use. Add the `CompareExchangeVariables()` method:

```
private static void CompareExchangeVariables()
{
    Interlocked.CompareExchange(ref _value, 123, long.MaxValue);
}
```

In our `CompareExchangeVariables()` method, we call the `CompareExchange()` method to compare `_value` with `long.MaxValue`. If the two values are equal, then `_value` is replaced with the value of 123. We'll now add our `AddVariables()` method:

```
private static void AddVariables()
{
    Interlocked.Add(ref _value, 321);
}
```

The `AddVariables()` method calls the `Add()` method to access the `_value` member variable and update it with the value of `_value` plus 321. Next, we'll add our `DecrementVariable()` method:

```
private static void DecrementVariable()
{
    Interlocked.Decrement(ref _value);
}
```

This method calls the `Decrement()` method, which decrements the `_value` member variable by 1. Our next method is `IncrementValue()`:

```
private static void IncrementVariable()
{
    Interlocked.Increment(ref _value);
}
```

In our `IncrementVariable()` method, we increment the `_value` member variable by calling the `Increment()` method. The next method we will write is the `ReadVariable()` method:

```
private static long ReadVariable()
{
    // The Read method is unnecessary on 64-bit systems, because 64-bit
    // read operations are already atomic. On 32-bit systems, 64-bit read
    // operations are not atomic unless performed using Read.
    return Interlocked.Read(ref _value);
}
```

Since 64-bit read operations are atomic, calling the `Interlocked.Read()` method is unnecessary. However, on 32-bit systems, for 64-bit reads to be atomic, you need to call the `Interlocked.Read()` method. Add the `PerformUnsafeCodeSafely()` method:

```
private static void PerformUnsafeCodeSafely()
{
    for (int i = 0; i < 5; i++)
    {
        UseResource();
        Thread.Sleep(1000);
    }
}
```

The `PerformUnsafeCodeSafely()` method loops five times. Each iteration of the loop calls the `UseResource()` method, and then the thread goes to sleep for 1 second. Now, we'll add the `UseResource()` method:

```
static bool UseResource()
{
    if (0 == Interlocked.Exchange(ref _resourceInUse, 1))
    {
        Console.WriteLine($"{Thread.CurrentThread.Name} acquired the
lock");
        NonThreadSafeResourceAccess();
        Thread.Sleep(500);
        Console.WriteLine($"{Thread.CurrentThread.Name} exiting lock");
        Interlocked.Exchange(ref _resourceInUse, 0);
        return true;
    }
    else
    {
        Console.WriteLine($"{Thread.CurrentThread.Name} was denied the
lock");
        return false;
    }
}
```

The `UseResource()` method prevents a lock from being obtained if the resource is in use, as identified by the `_resourceInUse` variable. We start by setting the `_resourceInUse` member variable value to 1 by calling the `Exchange()` method. The `Exchange()` method returns an integer, which we compare against 0. If the value returned by `Exchange()` is 0, then the method is not in use.

If the method is in use, then we output a message informing the user that the current thread was denied the lock.

If the method is not in use, then we output a message informing the user that the current thread has obtained a lock. We then call the `NonThreadSafeResourceAccess()` method and then send the thread to sleep for half a second to simulate work.

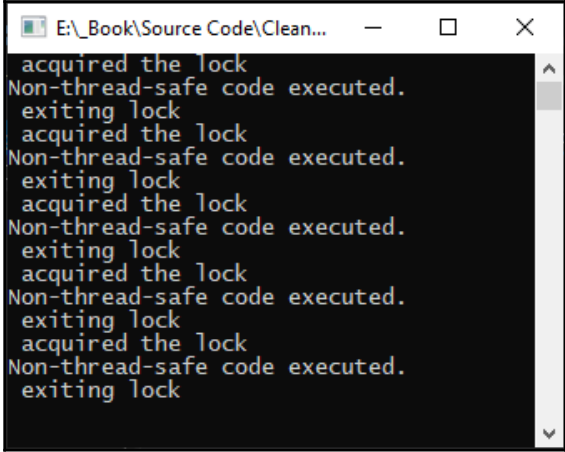
When the thread awakes, we output a message informing the user that the current thread has exited the lock. Then, we release the lock by calling the `Exchange()` method and setting the value of `_resourceInUse` to 0. Add the `NonThreadSafeResourceAccess()` method:

```
private static void NonThreadSafeResourceAccess()
{
    Console.WriteLine("Non-thread-safe code executed.");
}
```

`NonThreadSafeResourceAccess()` is where non-thread-safe code is executed in the safety of the lock. In our method, we simply inform the user with a message. The last job to do before we run our code is to update our `Main()` method, as follows:

```
static void Main(string[] args)
{
    CompareExchangeVariables();
    AddVariables();
    DecrementVariable();
    IncrementVariable();
    ReadVariable();
    PerformUnsafeCodeSafely();
}
```

Our `Main()` method calls the methods that test the `Interlocked` methods. Run the code and you should see something similar to the following:



```
E:\_Book\Source Code\Clean...
acquired the lock
Non-thread-safe code executed.
exiting lock
acquired the lock
Non-thread-safe code executed.
exiting lock
acquired the lock
Non-thread-safe code executed.
exiting lock
acquired the lock
Non-thread-safe code executed.
exiting lock
acquired the lock
Non-thread-safe code executed.
exiting lock
```

We'll now go over some general recommendations.

General recommendations

In this final section, we will look at some general recommendations from Microsoft for working on multi-threaded applications. They include the following:

- Avoid using `Thread.Abort` to terminate other threads.
- Use a mutex, `ManualResetEvent`, `AutoResetEvent`, and `Monitor` to synchronize activities between multiple threads.

- Where possible, use a thread pool for your worker threads.
- If you have any worker threads that gets blocked, then use `Monitor.PulseAll` to notify all the threads of a change in the worker thread's state.
- Avoid using `this`, type instances, and string instances including string literals as lock objects. Avoid using types of the lock objects.
- Instance locks can result in deadlocks, so exercise caution when using them.
- Use the `try/finally` block with threads that enter a monitor so that in the `finally` block, you ensure that the thread leaves the monitor by calling `Monitor.Exit()`.
- Use different threads for different resources.
- Avoid assigning multiple threads to the same resource.
- I/O tasks should have their own thread as they block when performing I/O operations. This way, you allow other threads to run.
- User input should have its own dedicated thread.
- Improve performance for simple state changes by using the methods of the `System.Threading.Interlocked` class instead of the lock statement.
- For heavily used code, avoid synchronization as it can lead to deadlocks and race conditions.
- Make static data thread-safe by default.
- Instance data must not be thread-safe by default; otherwise, you decrease performance, increase lock contention, and introduce the possibility of race conditions and deadlocks occurring.
- Avoid using static methods that alter state as they lead to threading bugs.

That concludes our look at threading and concurrency. Let's run through a summary of what we have learned.

Summary

In this chapter, we covered what threading is and how to use it. We looked at the problems of deadlocks and race conditions in action, and we saw how to prevent these exceptional circumstances using a lock statement and the TPL library. We also discussed the thread safety of static constructors, static methods, immutable objects, and mutable objects. We saw why using immutable objects is a thread-safe way of transferring data between threads, and we reviewed some general recommendations for working with threads.

We also saw how making your code thread-safe can have a lot of benefits. In the next chapter, we will look at designing effective APIs. But for now, you can test your knowledge by answering the following questions and you can further your reading by referring to the links provided.

Questions

1. What is a thread?
2. How many threads are there in a single-threaded application?
3. What types of threads are there?
4. What thread terminates as soon as the program is exited?
5. What thread continues through to completion, even if the program is exited?
6. What code makes a thread sleep for half a millisecond?
7. How do you instantiate a thread that calls a method named `Method1`?
8. How do you make a thread a background thread?
9. What is a deadlock?
10. How do you exit a lock obtained using `Monitor.TryEnter(objectName)`?
11. How can you recover from a deadlock?
12. What is a race condition?
13. What is one way to prevent race conditions?
14. What makes static methods unsafe?
15. Are static constructors thread-safe?
16. What is responsible for managing groups of threads?
17. What is an immutable object?
18. Why are immutable objects preferred to mutable objects in threaded applications?

Further reading

- <https://www.c-sharpcorner.com/blogs/mutex-and-semaphore-in-thread> provides examples of using a mutex and a semaphore.
- <https://www.guru99.com/mutex-vs-semaphore.html> explains the differences between a mutex and a semaphore.

- <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-constructors> is the official Microsoft documentation on static constructors.
- <https://docs.microsoft.com/en-us/dotnet/standard/threading/managed-threading-best-practices> is the official Microsoft guidance on Microsoft's managed threading best practices.
- <https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl> is the official Microsoft API documentation for the TPL.
- <https://www.c-sharpcorner.com/UploadFile/1d42da/interlocked-class-in-C-Sharp-threading/> covers the Interlocked class in C# threading.
- <http://geekswithblogs.net/BlackRabbitCoder/archive/2012/08/23/c.net-little-wonders-interlocked-read-and-exchange.aspx> provides a discussion on `System.Threading.Interlocked` with examples.
- <http://www.albahari.com/threading/> is a link to a free eBook by Joseph Albahari about threading in C#.
- <https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable?view=netcore-3.1> is the official Microsoft documentation on the immutable collections available in the `System.Collections.Immutable` namespace.

9 Designing and Developing APIs

Application Programming Interfaces (APIs) have never been so vital in so many ways as they are these days. APIs are used to connect governments and institutions in the sharing of data and in a collaborative manner for business and governmental issues. They are used between doctors' surgeries and hospitals to share patient data in real time. You use APIs every day when you connect to your emails and collaborate with your colleagues and clients through platforms such as Microsoft Teams, Microsoft Azure, Amazon Web Services, and Google Cloud Platform.

Every time you chat with someone or have a video call with them using your computers or phones, you are using APIs. When streaming video conferences, entering a website technical support chat, or streaming your favorite music and videos, you are using APIs. So, as a programmer, it is imperative that you are well versed in what APIs are and how to design, develop, secure, and deploy them.

In this chapter, we will talk about what APIs are, how they benefit you, and why it is necessary to learn about them. We will also be discussing API proxies, design and development guidelines, how to design APIs using RAML, and how to document APIs using Swagger.

The following topics are covered in this chapter:

- What is an API?
- API proxies
- API design guidelines
- API design using RAML
- Swagger API development

This chapter will assist you in gaining the following skills:

- Understanding APIs and why you need to learn about them
- Understanding API proxies and why we use them
- Being aware of design guidelines when designing your own APIs
- Using RAML to design your own APIs
- Using Swagger to document your APIs

By the end of this chapter, you will understand the basics of good API design and you will be armed with the knowledge needed to push your API abilities forward. It is important to understand what an API is, and so that is how we shall start this chapter. But first, make sure that you implement the following technical requirements to get the most out of this chapter.

Technical requirements

We will be using the following technologies in this chapter to create an API:

- Visual Studio 2019 Community edition or higher
- Swashbuckle.AspNetCore 5 or higher
- Swagger (<https://swagger.io>)
- Atom (<http://atom.io>)
- API Workbench by MuleSoft

What is an API?

APIs are reusable libraries that can be shared between different applications and can be made available via REST services (in which case, they are referred to as **RESTful APIs**).



Representational State Transfer (REST) was introduced by Roy Fielding in 2000.

REST is an architectural style that is made up of *constraints*. Altogether there are six constraints that should be considered when writing REST services. These constraints are as follows:

- **Uniform interface:** This is used to identify resources, and it manipulates these resources through *representation*. Messages use hypermedia and are self-descriptive. **Hypermedia as the Engine of Application State (HATEOAS)** is utilized to contain information about what operation can be carried out next by the client.
- **Client-server:** This constraint utilizes information hiding through *encapsulation*. So, only the API calls that are to be used by clients will be visible and all the other APIs will be kept hidden. A RESTful API should be independent of other parts of the system, making it loosely coupled.
- **Stateless:** This states that the RESTful API has no session nor history. If a session or history is required by the client, then the client must provide all the relevant information in the request to the server.
- **Cacheable:** This constraint means that resources must declare themselves cacheable. This means that resources can be accessed quickly. As a result, our RESTful API gains speed and our server load is reduced.
- **Layered system:** The layered system constraint dictates that each layer must do one and only one thing. Each component should only know what it needs to use in order to function and perform its tasks. A component should not know about the parts of the system that it does not use.
- **Optional executable code:** The executable code constraint is optional. This constraint determines that servers can, on a temporary basis, extend or customize the functionality of a client by transferring executable code.

So, when designing an API, it would be prudent to assume that the end user will be a programmer with any level of experience. They should be able to easily obtain the API, read up on it, and put it to work straight away.

Don't worry about creating the perfect API. APIs usually evolve over time anyway, and if you have ever worked with Microsoft APIs, you will know that they regularly upgrade them. APIs with features that will be removed in the future are often marked with an annotation that informs the user not to use a particular property or method as they will be removed in a future release. Then, when they will no longer be used, they usually get marked with an obsolete annotation before they are finally removed. This tells the users of the API to upgrade any apps using deprecated features.

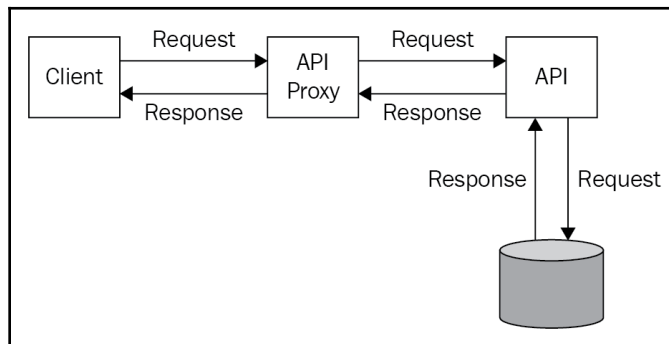
Why use REST services for API access? Well, many companies make large profits from making their APIs available online and charging for them. So, RESTful APIs can be a very valuable asset. Rapid API (<https://rapidapi.com/>) has free and paid APIs available for use.

Your APIs can remain permanently in place. If you use a cloud provider, your APIs can be highly scalable and you can make them generally available either for free or via a subscription. You can encapsulate all the complicated workings and expose what you need to via a simple interface, and because your APIs will be small and cacheable, they are very fast. Let's look now at API proxies and why you would use them.

API proxies

An **API proxy** is a class that sits between the client and your API. It is, in essence, an API contract between you and the developers who will be using your API. So, rather than giving developers direct access to your API's backend services, which may break over time as you refactor and extend them, you provide assurance to the consumers of your API that the API contract will be honored, even when the backend services change.

The following diagram displays the communication between the client, an API proxy, the actual API being accessed, and the API's communication with the data source:



A console application that shows how easy it is to implement the proxy pattern will be programmed in this section. Our example will have an interface that will be implemented by the API and the proxy. The API will return the actual message and the proxy will obtain the message from the API and pass it to the client. Proxies can also do much more than simply call the API method and return a response. They can perform authentication, authorization, routing based on credentials, and much more. However, our example will be kept to the absolute minimum so that you can see the simplicity in the proxy pattern.

Start a new .NET Framework console application. Add the `Apis`, `Interfaces`, and `Proxies` folders and place the `HelloWorldInterface` interface into the `Interfaces` folder:

```
public interface HelloWorldInterface
{
    string GetMessage();
}
```

Our interface method, `GetMessage()`, returns a message as a string. Both the proxy and API class will implement this interface. The `HelloWorldApi` class implements `HelloWorldInterface`, so add it to the `Apis` folder:

```
internal class HelloWorldApi : HelloWorldInterface
{
    public string GetMessage()
    {
        return "Hello World!";
    }
}
```

As you can see, our API class implements the interface and returns a "Hello World!" message. We have also made the class an internal class. This prevents external callers from being able to access the contents of this class. Now, we'll add our `HelloWorldProxy` class to the `Proxies` folder:

```
public class HelloWorldProxy : HelloWorldInterface
{
    public string GetMessage()
    {
        return new HelloWorldApi().GetMessage();
    }
}
```

Our proxy class is set to public as this class will be called by clients. The proxy class will call the `GetMessage()` method within the API class and return the response to the caller. All that's left to do now is to modify our `Main()` method:

```
static void Main(string[] args)
{
    Console.WriteLine(new HelloWorldProxy().GetMessage());
    Console.ReadKey();
}
```

Our `Main()` class calls the `GetMessage()` method of the `HelloWorldProxy` proxy class. Our proxy class calls the API class and the returned method is printed in the console window. The console then waits for a keypress before it exits.

Run the code and view the output; you have successfully implemented an API proxy class. You can make your proxies as simple or as complicated as they need to be, but what you have done here is the foundation for success.

In this chapter, we will be building an API. So, let's discuss what we will be building, and then get going with working on it. Once you've completed the project, you will have a working API that generates a monthly dividend payment calendar in JSON format.

API design guidelines

There are some basic guidelines to follow to write an effective API—for example, your resources should use nouns in plural form. So, for example, if you had a wholesale website, then your URLs would look something like the following dummy links:

- `http://wholesale-website.com/api/customers/1`
- `http://wholesale-website.com/api/products/20`

The preceding URLs will follow the controller routes of `api/controller/id`. In terms of relationships within the business domain, these should also be reflected in URLs such as `http://wholesale-website.com/api/categories/12/products`—this call will return a list of products for category 12.

If you need to use a verb as a resource, then you can do so. When making an HTTP request, use `GET` to retrieve items, `HEAD` to retrieve only headers, `POST` to insert or save a new resource, `PUT` to replace a resource, and `DELETE` to remove a resource. Keep resources lean by using query parameters.

When paginating results, a ready-made set of links should be made available to the client. RFC 5988 introduced **link headers**. In the specification, an **International Resource Identifier (IRI)** is a typed connection between two resources. For more information, refer to <https://www.greenbytes.de/tech/webdav/rfc5988.html>. The format of link header requests is as follows:

- `<https://wholesale-website.com/api/products?page=10&per_page=100>; rel="next"`
- `<https://wholesale-website.com/api/products?page=11&per_page=100>; rel="last"`

The versioning of your API can be done in the URL. So, each resource will have a different URL for the same resource, as in the following examples:

- <https://wholesale-website.com/api/v1/cart>
- <https://wholesale-website.com/api/v2/cart>

This form of versioning is very simple and makes it easy to find the correct version of the API.

JSON is the preferred resource representation. It is much more human-readable than XML and is also lighter in size. When you are using the `POST`, `PUT`, and `PATCH` verbs, you should also require the content-type header to be set to `application/JSON`, or throw the 415 HTTP status code (which means unsupported media type). Gzip is a single-file/stream lossless data compression utility. Use Gzip by default to save a good percentage in bandwidth, and always set the `HTTP Accept-Encoding` header to `gzip`.

Always use HTTPS (TLS) for your APIs. The identification of the caller should always be done in the header. We saw this with our API when we set the `x-api-key` header with our API access key. Each request should be authenticated and authorized. Unauthorized access should result in an HTTP 403 Forbidden response. Also, use the correct HTTP response codes. So, if a request is successful, use the 200 status code, for a resource that is not found, use 404, and so on. For an exhaustive list of HTTP status codes, visit <https://httpstatuses.com/>. OAuth 2.0 is the industry-standard protocol for authorization. You can read all about it at <https://oauth.net/2/>.

An API should provide documentation on its usage with examples. Documentation should always be up to date with the current version, and it should be visually appealing and easy to read. We'll look at Swagger to help us create documentation later in this chapter.

You never know when your API is going to need to scale. So, this should be factored in from the start. In our *Dividend Calendar API* project in the next chapter, you will see how we implement throttling to just one API call per month on a specific day of the month. However, you can effectively come up with 1,001 different ways to throttle your APIs depending on your own needs, but this should be done at the start of a project. So, as soon as you start a new project, think *scalability*.

For security and performance reasons, you may decide to implement an API proxy. An API proxy disconnects the client from accessing your API directly. A proxy can access an API in the same project or on an external API. By using a proxy, you can avoid exposing your database schema.

Responses to the client should never match the structure of your database. This can act as a green light for hackers. So, avoid one-to-one mappings between database structures and the responses you send back to clients. You should also hide identifiers from your clients as they can be used to manually access data by the client.

An API contains resources. A **resource** is an item that can be operated on in some way. Resources can be files or data. For example, students in a school database are resources that can be added, edited, or deleted. Video files can be retrieved and played, as can audio files. Images are also resources, as are report templates that will be opened, manipulated, and filled with data before they are presented to the user.

Often, resources form collections of items, such as students in a school database. `Students` is the name of a collection of the `Student` type. Resources are accessed via a URL. A URL contains the path to a resource.

URLs are known as **API endpoints**. An API endpoint is an address of a resource. This resource may be accessed by an URL with one or more parameters or an URL without any parameters. An URL should only contain plural nouns (names of resources) and should not contain verbs or actions. Parameters can be used to identify a single resource within a collection. Pagination should be employed if the dataset is going to be very large. For requests with parameters that break the URI length limit, you can place the parameters in the body of a `POST` request.

Verbs form part of the HTTP request. The `POST` verb is used to add a resource. To retrieve one or more resources, you use the `GET` verb. `PUT` updates or replaces one or more resources, and `PATCH` updates or modifies a resource or collection. `DELETE` deletes a resource or collection.

You should always make sure you provide and respond to HTTP status codes appropriately. For a complete list of HTTP status codes, visit <https://httpstatuses.com/>.

As for field, method, and property names, you can use any convention you like, but it must be consistent and follow the company guidelines. Camel case convention is normally used in JSON. Since you will be developing APIs in C#, it is best to stick to the industry-standard C# naming conventions.

Since your API will evolve over time, it is best to employ some form of versioning. Versioning allows consumers to consume specific versions of your API. This can be very important for providing backward compatibility when new versions of your API implement breaking changes. It is normally a good idea to have the version number, such as `v1` or `v2`, included in the URL. Whatever method you use to version your APIs, just remember to be *consistent*.

If you will be consuming third-party APIs, you will need to keep the API keys secret. One way to accomplish this is to store the keys in a key vault, such as Azure Key Vault, which requires authentication and authorization. You should also secure your own APIs with a method of your choosing. A common method nowadays is through the use of API keys. You will see how to use API keys and Azure Key Vault to secure third-party keys and your own APIs in the next chapter.

Well-defined software boundaries

Nobody in their right mind likes spaghetti code. It is very hard to read, maintain, and extend. So, when designing an API, you can overcome this problem with well-defined software boundaries. A well-defined software boundary is known as a **bounded context** in **Domain-Driven Design (DDD)**. In business terms, a bounded context is a business operational unit, such as HR, finance, customer services, infrastructure, and so on. These business operational units are known as **domains**, and they can be broken down into smaller sub-domains. Then, these sub-domains can be broken down into even smaller sub-domains.

By breaking a business up into business operational units, domain experts can be employed in those specific areas. A common language can be determined at the start of a project so that the business understands the IT terms and the IT staff understands the business terms. If the business and IT staff are language-aligned, there is less margin for errors due to misunderstandings from both sides.

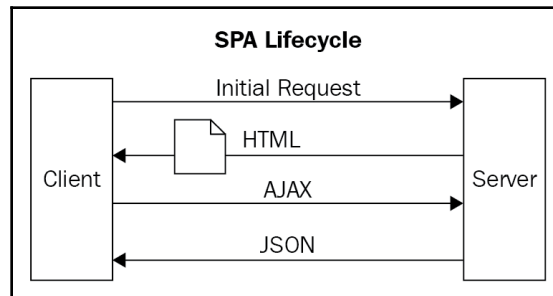
Having a major project broken down into sub-domains means that you can have smaller teams working independently on projects. So, large development teams can be grouped into smaller teams working concurrently on various projects.

DDD is a big subject in itself and is not covered here. However, links to more information are posted in the *Further reading* section of this chapter.

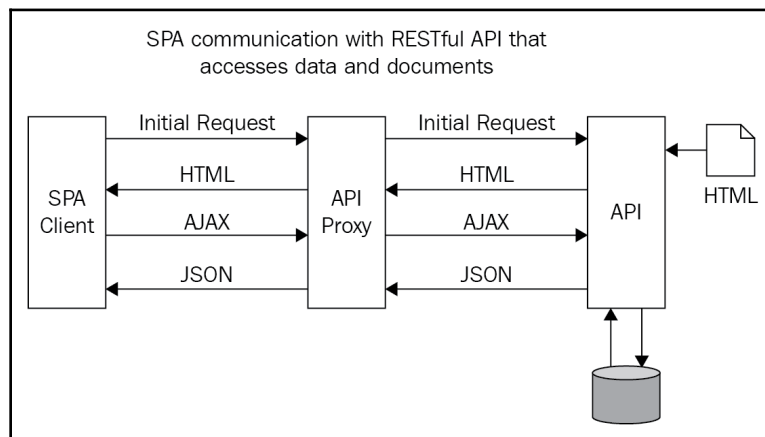
The only items that should be exposed by APIs are the interfaces that form contracts and API endpoints. Everything else should be hidden from the subscriber and consumer. This means that even large databases can be broken down so that each API has its own database. Given how large and complex websites can be by today's standards, we can even have micro-services with micro-databases and micro-frontends.

A micro-frontend is a small portion of a web page that is dynamically retrieved and modified according to user interactions. That frontend will interact with an API, which in turn will access a micro-database. This is ideal in terms of **Single-Page Applications (SPAs)**.

SPAs are websites consisting of a single page. When a user initiates an action, only the required portion of the web page is updated; the rest of the page remains the same. So, for example, say the web page has an aside. That aside displays adverts. Those adverts are stored in the database as portions of HTML. The aside is set to auto-update itself every 5 seconds. When 5 seconds is up, the aside requests the API to assign a new advert. The API then uses whatever algorithm is in place to obtain a new advert to display from the database. The HTML document is then updated, and the aside is updated with the new advert. The following diagram shows the typical SPA life cycle:



This aside is a well-defined software boundary. It does not need to know anything whatsoever about the rest of the page it is displayed in. All it is concerned with is displaying a new advert every 5 seconds:



The previous diagram shows an SPA communicating with a RESTful API via an API proxy, and the API is able to access documents and databases.

The only components that make up the aside are an HTML document fragment, a micro-service, and a database. These can be worked on by a small team in whatever technology that the team prefers and is comfortable with. The full SPA could be made up of hundreds of micro-documents, micros-services, and micro-databases. The key point is that these services could be made up of any technology and worked on independently by any team. Multiple projects could also be worked on concurrently.

Within our bounded context, we can use the following software methodologies to improve the quality of our code:

- **The Single Responsibility, Open/Closed, Liskov, Interface Segregation, and Dependency Inversion (SOLID) principles**
- **Don't Repeat Yourself (DRY)**
- **You Ain't Gonna Need It (YAGNI)**
- **Keep It Simple, Stupid (KISS)**

These methodologies work well together to eliminate duplicate code, prevent you from writing code that you don't need, and to keep objects and methods small. The reason why we develop for a class and a method is that they should both do only one thing and do it well.

Namespaces are used to perform logical groupings. We can use namespaces to define software boundaries. The more specific a namespace is, the more meaningful it is to the programmer. Meaningful namespaces help programmers to partition code and find what they are looking for with ease. Use namespaces to logically group interfaces, classes, structs, and enums.

In the next section, you will learn how to design an API using RAML. Then, you will generate a C# API from the RAML file.

Understanding the importance of good quality API documentation

When working on a project, it is necessary to understand all the APIs that are already used. The reason for this is that you can often end up writing code that already exists, which obviously leads to wasted effort. Not only that but by writing your own version of code that already exists, you now have two copies of code that do the same thing. This adds to the complexity of the software and increases the maintenance overhead as both versions of the code must be maintained. It also adds the potential for bugs.

On massive projects that are spread across multiple technologies and repositories, with teams that have a high staff turnaround, and especially where no documentation exists, code duplication becomes a real problem. Sometimes, there will only be one or two domain experts, with the majority of the team not knowing the system at all. I have worked on projects like this before and they are a real pain to maintain and expand.

That is why API documentation is vital for any project, no matter how large or how small it is. It is inevitable in the field of software development that people will move on, especially when more lucrative work is offered elsewhere. If the person moving on is the domain expert, then they will take their knowledge with them. If no documentation exists, then new developers to the project will have a steep learning curve in understanding the project by having to read the code. If the code is messy and complex, this can cause a real headache for onboarding new staff.

As a result, due to the lack of system knowledge, programmers will be inclined to more or less write the code they need from scratch to get the job done as they will be under pressure to deliver to the business on time. This will often lead to duplicate code and code reuse not being utilized. This causes the software to become complex and error-prone, and this kind of software ends up being hard to extend and maintain.

Now, you understand why APIs must be documented. A well-documented API will lead to greater understanding by programmers and is more inclined to get reused, thereby reducing the potential for code duplication and producing code that is hard to extend or maintain.

You should also be aware of any code that is marked as deprecated or obsolete. Deprecated code will be removed in future releases and obsolete code is no longer in use. If you are using APIs that are marked as deprecated or obsolete, then this code should be prioritized to address first.

Now that you understand the importance of good quality API documentation, we will look at a tool called Swagger. Swagger is an easy-to-use tool for producing nice-looking, high-quality API documentation.

Swagger API development

Swagger provides a powerful set of tools that are focused around API development. With Swagger, you can do the following things:

- **Design:** Design your API and model it to keep up with specification-based standards.
- **Build:** Build an API in C# that is stable and reusable.

- **Document:** Provide developers with documentation that they can interact with.
- **Test:** Easily test your API.
- **Standardize:** Apply constraints to your API architecture using your company guidelines.

We are going to get Swagger up and running in our ASP.NET Core 3.0+ project. So, start by creating the project in Visual Studio 2019. Select the **Web API** and **No Authentication** settings. Before we continue, it is worth noting that Swagger automatically generates aesthetically pleasing documentation that is functional. Very little code is required to set up Swagger, which is why many modern APIs use it.

Before we can use Swagger, we first need to install support for it in our project. To install Swagger, you must install version 5 or higher of the `Swashbuckle.AspNetCore` dependency package. As of the time of writing, the version available on NuGet is version 5.3.3. After the installation is complete, we need to add the Swagger services that we will be using to the services collection. In our case, we will only be using Swagger to document our API. In the `Startup.cs` class, add the following line to the `ConfigureServices()` method:

```
services.AddSwaggerGen(swagger =>
{
    swagger.SwaggerDoc("v1", new OpenApiInfo { Title = "Weather Forecast
API" });
});
```

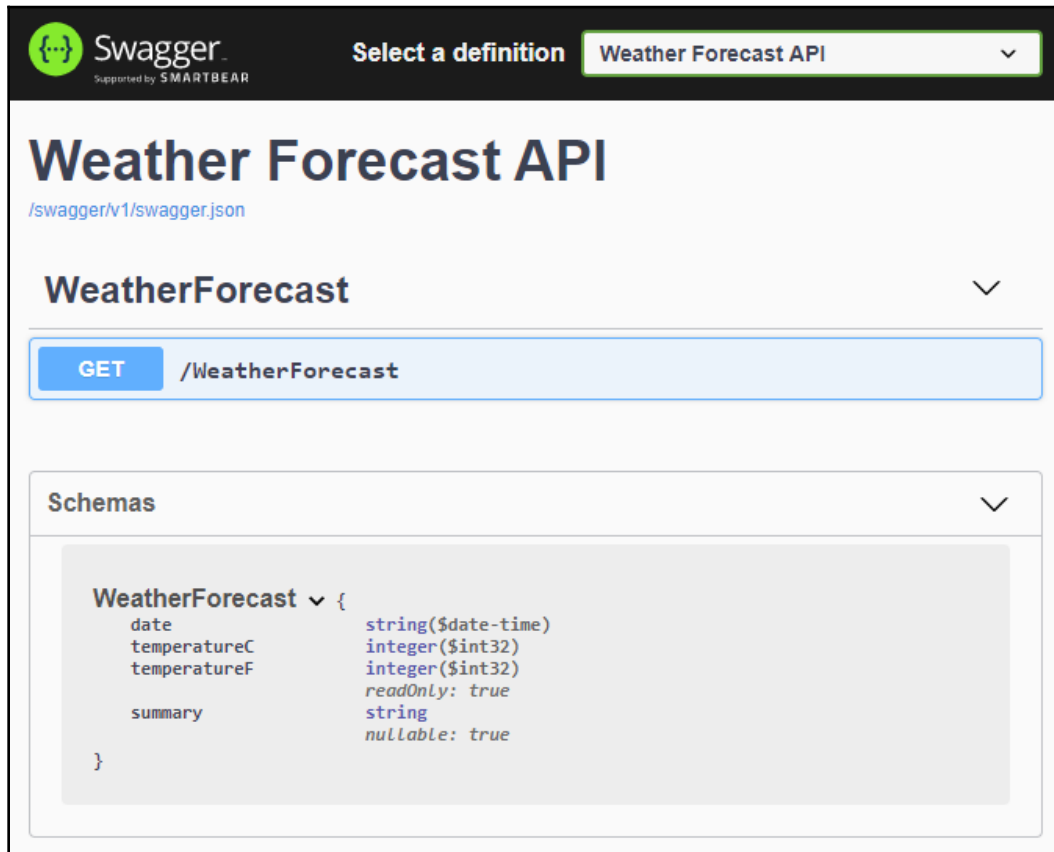
In the code we've just added, the Swagger documenting service has been assigned to the services collection. Our API version is `v1` and our API title is `Weather Forecast API`. We now need to update the `Configure()` method to add our Swagger middleware, as follows, immediately after the `if` statement:

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Weather Forecast API");
});
```

In our `Configure()` method, we are informing our app to use Swagger and the Swagger UI, and we assign our Swagger endpoint for `Weather Forecast API`. Next, you will need to install the `Swashbuckle.AspNetCore.Newtonsoft` NuGet dependency package (version 5.3.3, as of the time of writing). Then, add the following line to your `ConfigureServices()` method:

```
services.AddSwaggerGenNewtonsoftSupport();
```

We added Newtonsoft support for our Swagger documentation generation. That is all there is to getting Swagger up and running. So, run your project and navigate to `https://localhost:PORT_NUMBER/swagger/index.html`. You should see the following web page:



We will now take a look at why we should pass immutable structs instead of mutable objects.

Passing immutable structs instead of mutable objects

In this section, you are going to write a computer program that processes 1 million objects and 1 million immutable structs. You will see how much quicker, in terms of performance, structs are over objects. We will be writing some code that processes 1 million objects in 1,440 milliseconds and processes 1 million structs in 841 milliseconds. That is a difference of 599 milliseconds. Such a small unit of time might not sound like a lot, but when working with massive datasets, you will see big performance improvements when using immutable structs over mutable objects.

Values in mutable objects can also be modified between threads, which can be very bad for business. Imagine having £15,000 in your bank account and you pay your landlord £435 in rent. Your account has an overdraft limit that can be exceeded. Now, at the same time that you are paying £435, someone else is paying a car firm £23,000 for a new car. The value on your account is modified by the car purchaser's thread. So, you end up paying your landlord £23,000, leaving your bank balance £8,000 in debt. We won't code an example of mutable data being changed between threads as this was covered in [Chapter 8, *Threading and Concurrency*](#).



The key points of this section are that structs are faster than objects and immutable structs are thread-safe.

When creating and passing objects, structs are more performant than objects. You can also make structs immutable so that they are thread-safe. Here, we will write a small program. This program will have two methods—one will create 1 million person objects and the other will create 1 million person structures.

Add a new .NET Framework console application called `CH11_WellDefinedBoundaries` and the following `PersonObject` class:

```
public class PersonObject
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```


This object will be used to create 1 million people objects. Now, add the `PersonStruct`:

```
public struct PersonStruct
{
    private readonly string _firstName;
    private readonly string _lastName;

    public PersonStruct(string firstName, string lastName)
    {
        _firstName = firstName;
        _lastName = lastName;
    }

    public string FirstName => _firstName;
    public string LastName => _lastName;
}
```

This struct is immutable, with the `readonly` properties being set via the constructor and used to create our 1 million structs. Now, we can modify the program to show the performance between object and struct creation. Add the `CreateObject()` method:

```
private static void CreateObjects()
{
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    var people = new List<PersonObject>();
    for (var i = 1; i <= 1000000; i++)
    {
        people.Add(new PersonObject { FirstName = "Person", LastName =
        $"Number {i}" });
    }
    stopwatch.Stop();
    Console.WriteLine($"Object: {stopwatch.ElapsedMilliseconds}, Object
    Count: {people.Count}");
    GC.Collect();
}
```

As you can see, we start a stopwatch, create a new list, and add 1 million person objects to the list. We then stop the stopwatch, output the results to the window, and then call the garbage collector to clean up our resources. Let's now add our `CreateStructs()` method:

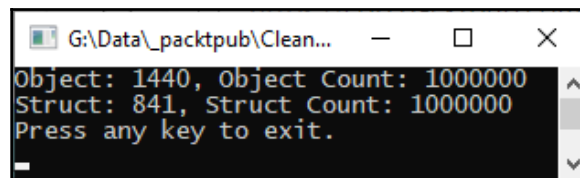
```
private static void CreateStructs()
{
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    var people = new List<PersonStruct>();
    for (var i = 1; i <= 1000000; i++)
```

```
{
    people.Add(new PersonStruct("Person", $"Number {i}"));
}
stopwatch.Stop();
Console.WriteLine($"Struct: {stopwatch.ElapsedMilliseconds}, Struct
Count: {people.Count}");
GC.Collect();
}
```

Our structure does a similar thing here as for the `CreateObjects()` methods, but creates a list of structs and adds 1 million structs to the list. Finally, modify the `Main()` method, as follows:

```
static void Main(string[] args)
{
    CreateObjects();
    CreateStructs();
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
```

We call both of our methods and then wait for the user to press any key before we exit. Run the program and you should see the following output:



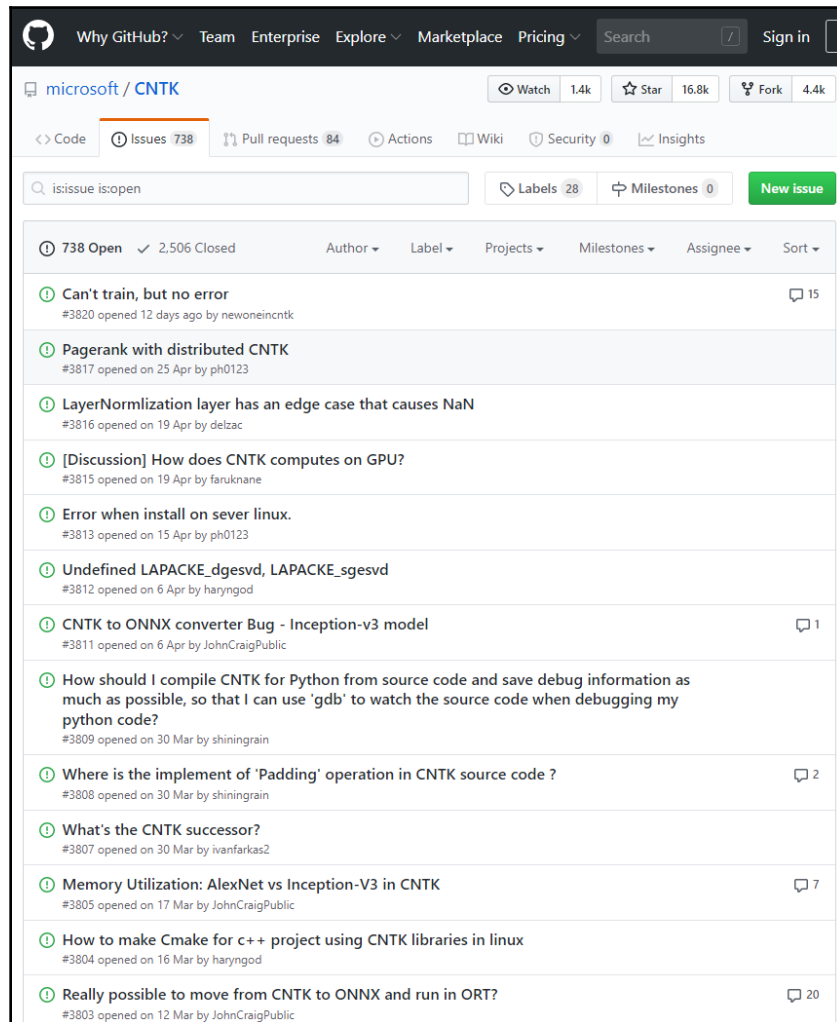
As you can see from the previous screenshot, it took 1,440 milliseconds to create 1 million objects and add them to a list of objects, and only 841 milliseconds to create 1 million structs and add them to a list of structs.

So, not only are you able to make structs immutable and thread-safe as they cannot be modified between threads, but they also perform a lot faster when compared to objects. Therefore, if you are dealing with large amounts of data, structs can save you a lot of processing time. Not only that but if you are charged per cycle of execution time by your cloud computing service, then using structs over objects is going to save you money.

Let's now have a look at writing third-party API tests for the APIs that you will be using.

Testing third-party APIs

You may ask "Why should I test third-party APIs?" Well, that is a good question. The reason why you should test third-party APIs is that just like your own code, third-party code is susceptible to programming errors. I remember once running into some real difficulty on a document processing website I was building for a law firm. After much investigation, I found the problem was down due to faulty JavaScript embedded in the Microsoft API that I was using. The following screenshot is of the GitHub **Issues** page for the Microsoft Cognitive Toolkit, which has **738** outstanding issues:



As you can see from the Microsoft Cognitive Toolkit, third-party APIs do have issues. That means as the programmer, the onus is on you to ensure that the third-party APIs that you employ work as expected. Should you encounter any bugs, then it is good practice to inform the third party of the bugs. If the API is open source and you have access to the source code, you can even check out the code and submit your own fixes.

Whenever you encounter bugs in third-party code that will not be addressed in time for you to meet your deadlines, then one option you have available to you is to write a **wrapper class** that has all the same constructors, methods, and properties and makes them call the same constructors, methods, and properties on the third-party class, with the exception that you write your own bug-free version of the third-party property or method that has the bug in it. Chapter 11, *Addressing Cross-Cutting Concerns*, provides sections on the proxy pattern and decorator pattern, which will help you write wrapper classes.

Testing your own APIs

In Chapter 6, *Unit Testing*, and Chapter 7, *End-to-End System Testing*, you saw, with code examples, how to test your own code. You should always test your own APIs as it is important to have complete trust in the quality of your APIs. Therefore, as a programmer, you should always unit test your code before you pass it on to quality assurance. Quality assurance should then run integration and regression testing on the API to ensure it meets the company's agreed level of quality.

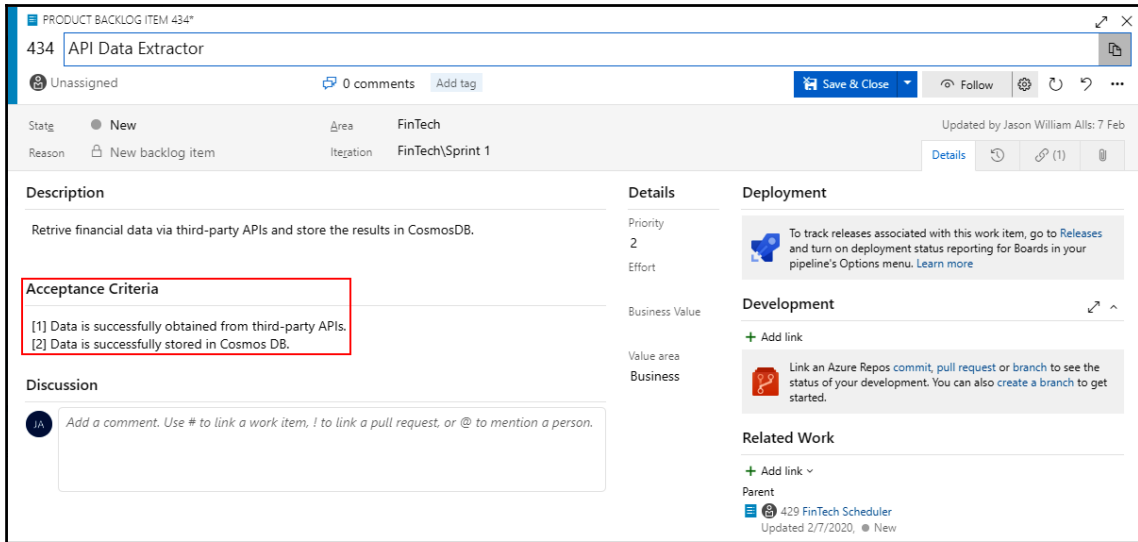
Your API may do exactly what the business has asked for and be perfect without bugs; but when it's integrated with the system, do peculiar things happen that you were unable to test for in certain situations? Often, I have encountered situations in development teams where code will work on one person's computer but not on other computers. Yet, there often seems to be no logical reason for this to be the case. These problems can be incredibly frustrating and even time-consuming to get to the bottom of. But you want these problems to be ironed out before you pass on your code for quality assurance, and most definitely before it is released into production. Having to deal with customer bugs is not always a pleasant experience.

Testing your programs should involve the following:

- When given the correct range of values, the method under test outputs the correct result.
- When given the incorrect range of values, the method provides the appropriate response without crashing.

Remember that your API should only include what the business has asked for and should not make the internal details accessible to clients. This is where the product backlog, which is part of the Scrum project management methodology, is useful.

The product backlog is the list of new features and technical debts that you and your team will be working on. Each item in the product backlog will have a description and acceptance criteria, as shown in the following screenshot:



You write your unit tests around the acceptance criteria. Your tests will include the normal path of execution and abnormal paths of execution. Using this screenshot as an example, we have two acceptance criteria:

- Data is successfully obtained from third-party APIs.
- Data is successfully stored in Cosmos DB.

In these two acceptance criteria, we know we will be calling APIs that obtain data. That data will be obtained from third parties. Once obtained, the data will then be stored in the database. On the face of it, this specification that we have to work with is quite vague. In real life, I have found that this is often the case.

Given the vagueness of the specification, we will make the assumption that the specification will be generic and will apply to different API calls, and we can assume that the data returned is JSON data. We will also make the assumption that the returned JSON data will be stored in its raw form in a Cosmos DB database.

So, what tests can we write for our first acceptance criteria? Well, we can write the following test cases:

1. When given an URL with a parameter list, assert that we receive a status of 200 and JSON returned for a GET request when all the correct information is supplied.
2. Assert that we receive a status of 401 when an unauthorized GET request has been made.
3. Assert that we receive a status of 403 when the authenticated user is forbidden from accessing the resource.
4. Assert that we receive a status of 500 when the server is down.

What tests can we write for our second acceptance criteria? Well, we can write the following test cases:

1. Assert that unauthorized access to the database is denied.
2. Assert that the API handles cases where the database is unavailable gracefully.
3. Assert that authorized access to the database is granted.
4. Assert that JSON insertion into the database succeeds.

So, even from such a vague specification, we have been able to obtain eight test cases. Between them, all these cases test for a successful roundtrip to the third-party server and back, and then into the database. They also test for various points at which the process could fail. If all of these tests pass, we have complete confidence in our code and that it will pass quality control when it leaves our hands as developers.

In the next section, we will look at how to design APIs using RAML.

API design using RAML

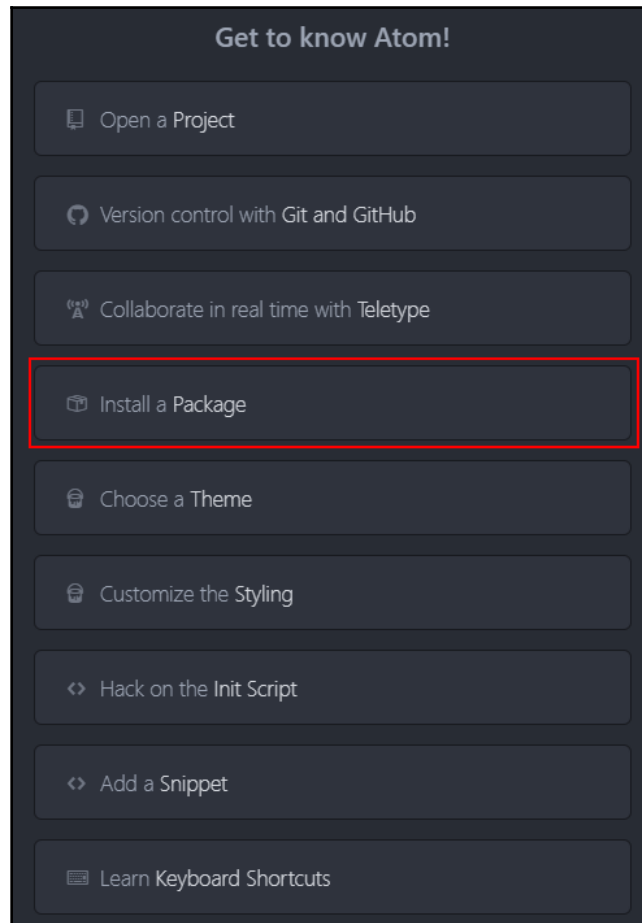
In this section, we will discuss designing an API using RAML. You can gain in-depth knowledge about all aspects of RAML from the RAML website (<https://raml.org/developers/design-your-api>). We are going to learn the basics of RAML by designing a really simple API using API Workbench in Atom. We'll start with the installation.

The first step is to install the packages.

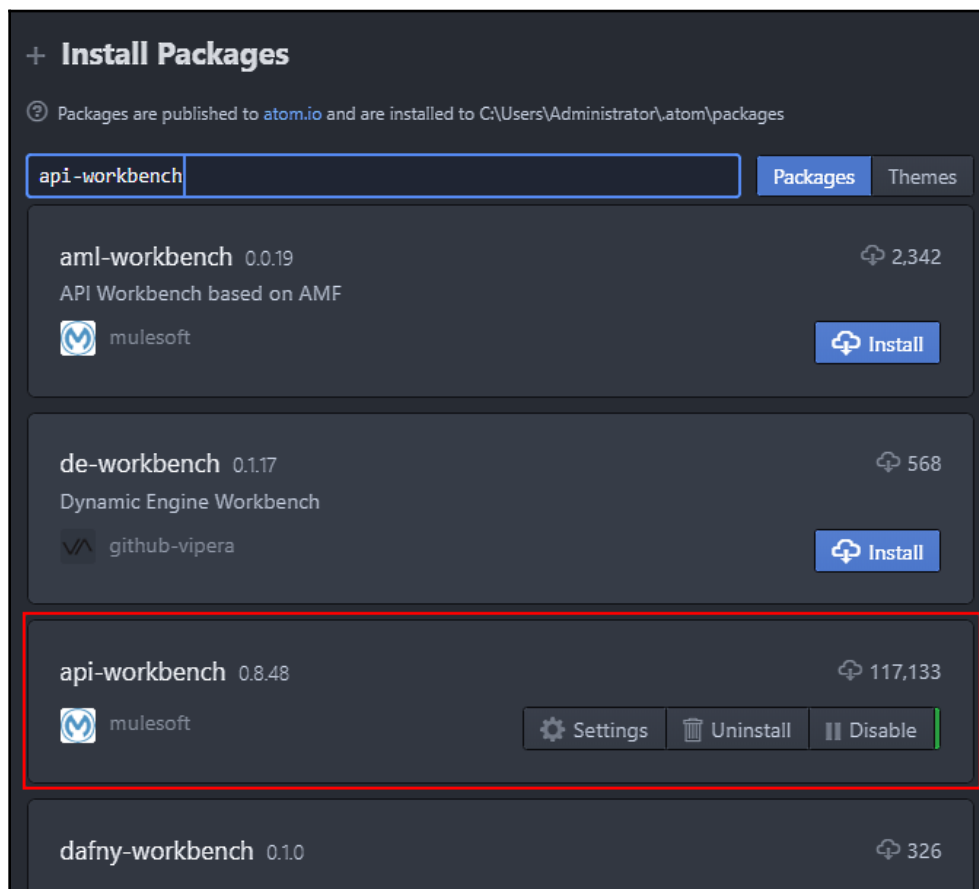
Installing Atom and API Workbench by MuleSoft

Let's see how to do that:

1. Start by installing Atom from <http://atom.io>.
2. Then, click on **Install a Package**:



3. Then, search for `api-workbench` by `mulesoft` and install it:



4. The installation is successful if you find it listed under **Packages | Installed Packages**.

Now that we've installed the packages, let's move onto creating the project.

Creating the project

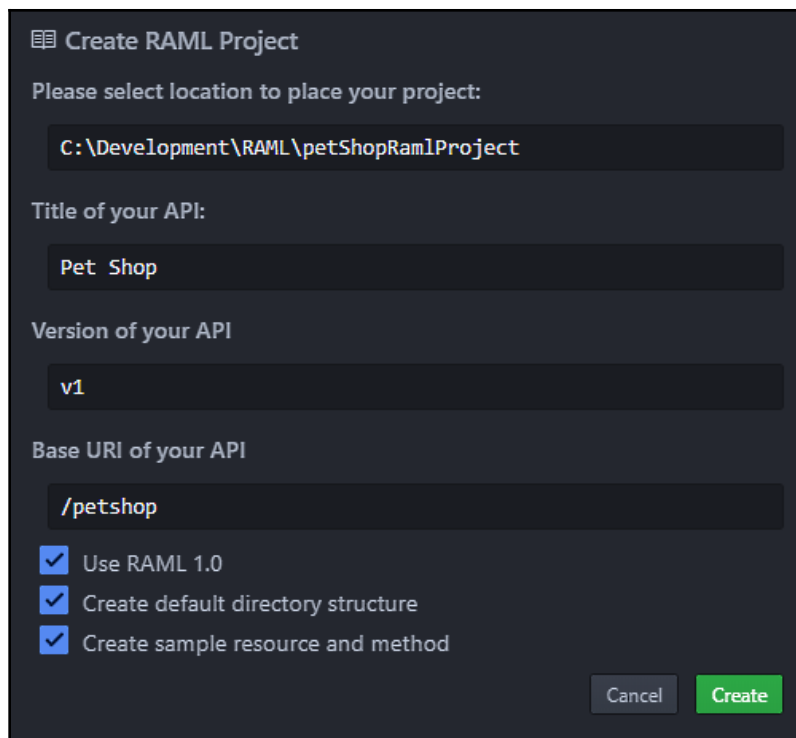
Let's see how to go about this:

1. Click **File | Add Project Folder**.
2. Create a new folder or select an existing one. I will create a new folder called `C:\Development\RAML` and open it.
3. Add a new file to your project folder called `Shop.raml`.
4. Right-click in the file and select **Add New | Create New API**.
5. Give it any name you want and then click on **Ok**. You have now just created your first API design.

If you look at the RAML file, you will see that its contents are in human-readable text. The API we've just created contains a simple `GET` command that returns a string that contains the words "Hello World":

```
#%RAML 1.0
title: Pet Shop
types:
  TestType:
    type: object
    properties:
      id: number
      optional?: string
      expanded:
        type: object
        properties:
          count: number
/helloWorld:
  get:
    responses:
      200:
        body:
          application/json:
            example: |
              {
                "message" : "Hello World"
              }
```

This is RAML code. You will see that it is pretty similar to JSON in that the code is simple, human-readable code that is indented. Delete the file. From the **Packages** menu, select **API Workbench | Create RAML Project**. Fill out the **Create RAML Project** dialog, as in the following screenshot:



Create RAML Project

Please select location to place your project:

C:\Development\RAML\petShopRamlProject

Title of your API:

Pet Shop

Version of your API

v1

Base URI of your API

/petshop

☒ Use RAML 1.0

☒ Create default directory structure

☒ Create sample resource and method

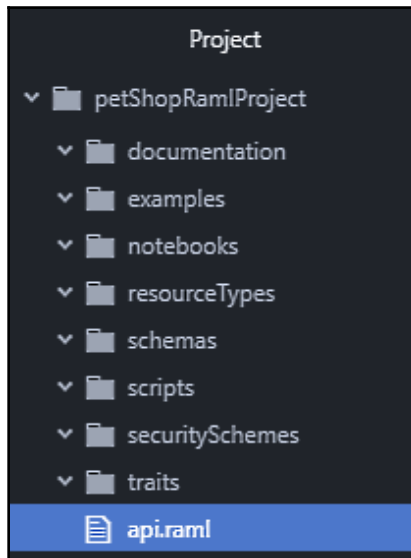
Cancel Create

The settings in this dialog will produce the following RAML code:

```
##RAML 1.0
title: Pet Shop
version: v1
baseUri: /petshop
types:
  TestType:
    type: object
    properties:
      id: number
      optional?: string
      expanded:
        type: object
        properties:
          count: number
/helloWorld:
  get:
    responses:
      200:
        body:
```

```
application/json:
  example: |
    {
      "message" : "Hello World"
    }
}
```

The main difference between the last RAML file and the first one you viewed is the insertion of the `version` and `baseUri` properties. These settings also update your **Project** folder's content, as follows:



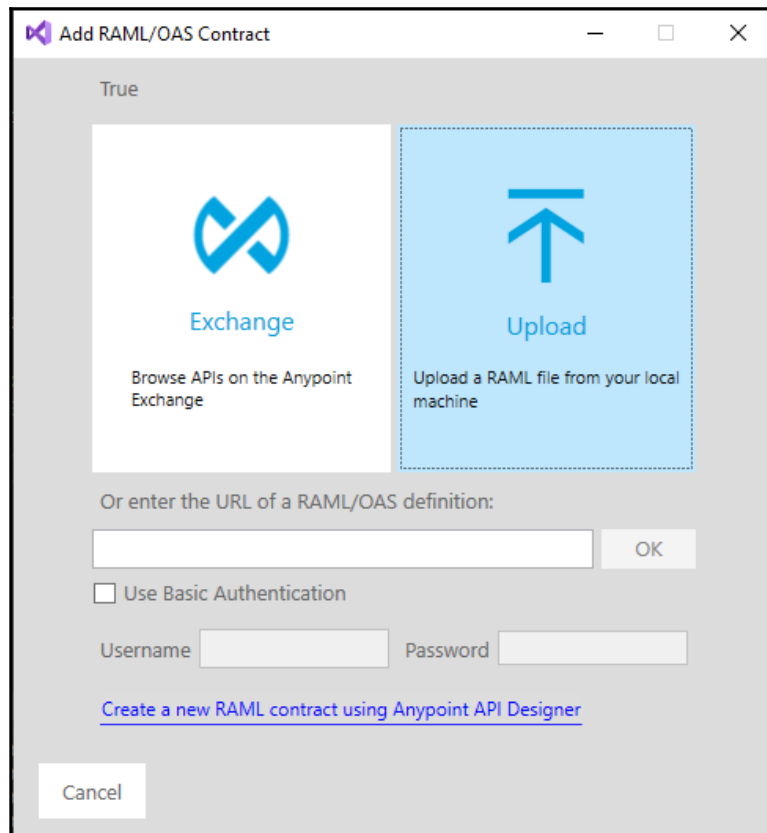
For a very detailed tutorial on this subject, head on over to <http://apiworkbench.com/docs/>. This URL also provides details on how to add resources and methods; fill method bodies and responses; add sub-resources; add examples and types; create and extract resource types; add resource type parameters, method parameters, and traits; reuse traits, resource types, and libraries; add more types and resources; extract libraries; and much more than we can cover in this chapter.

Now that we have a design that is language implementation-agnostic, how do we generate our API in C#?

Generating our C# API from our agnostic RAML design specification

You will need to have, as a minimum, Visual Studio 2019 Community edition installed. Then, make sure you close Visual Studio. Also, download and install the Visual Studio MuleSoft Inc. `RAMLToolsforNET` tool. With these tools installed, we will now proceed through the steps required to produce the skeleton framework of our previously specified API. This will be accomplished by adding a RAML/OAS contract and importing our RAML file:

1. In Visual Studio 2019, create a new .NET Framework console application.
2. Right-click on the project and select **Add RAML/OAS Contract**. This will open the following dialog:



3. Click on **Upload**, and then select your RAML file. The **Import RAML/OAS** dialog will then be presented. Fill the dialog out as shown and then click on **Import**:

Import RAML/OAS

Filename:

Namespace:

Generate Unit Tests: ☐

Use api version: ☒

Use async methods: ☒

Enable customization: ☒

Controllers Folder:

Controllers Namespace:

Models Folder:

Models Namespace:

Add '.generated.cs' suffix: ☐

Preview RAML/OAS

Cancel Import

Your project will now be updated with the required dependencies, and new folders and files will be added to your console application. You will notice three root folders, called *Contracts*, *Controllers*, and *Models*. In the *Contracts* folder, we have our RAML file and the *IV1HelloWorldController* interface. It contains one method: `Task<IHttpActionResult> Get()`. The *v1HelloWorldController* class implements the *Iv1HelloWorldController* interface. Let's have a look at the implemented `Get()` method in the controller class:

```
/// <summary>  
/// /helloWorld
```

```
/// </summary>
/// <returns>HelloWorldGet200</returns>
public async Task<IHttpActionResult> Get()
{
    // TODO: implement Get - route: helloWorld/helloWorld
    // var result = new HelloWorldGet200();
    // return Ok(result);
    return Ok();
}
```

In the preceding code, we can see that the code comments out the instantiation of the `HelloWorldGet200` class and the returned result. The `HelloWorldGet200` class is our model class. We can update our model to whatever data we want it to contain. In our simple example, we won't bother too much with this; we will just return the "Hello World!" string. Update the uncommented line to the following:

```
return Ok("Hello World!");
```

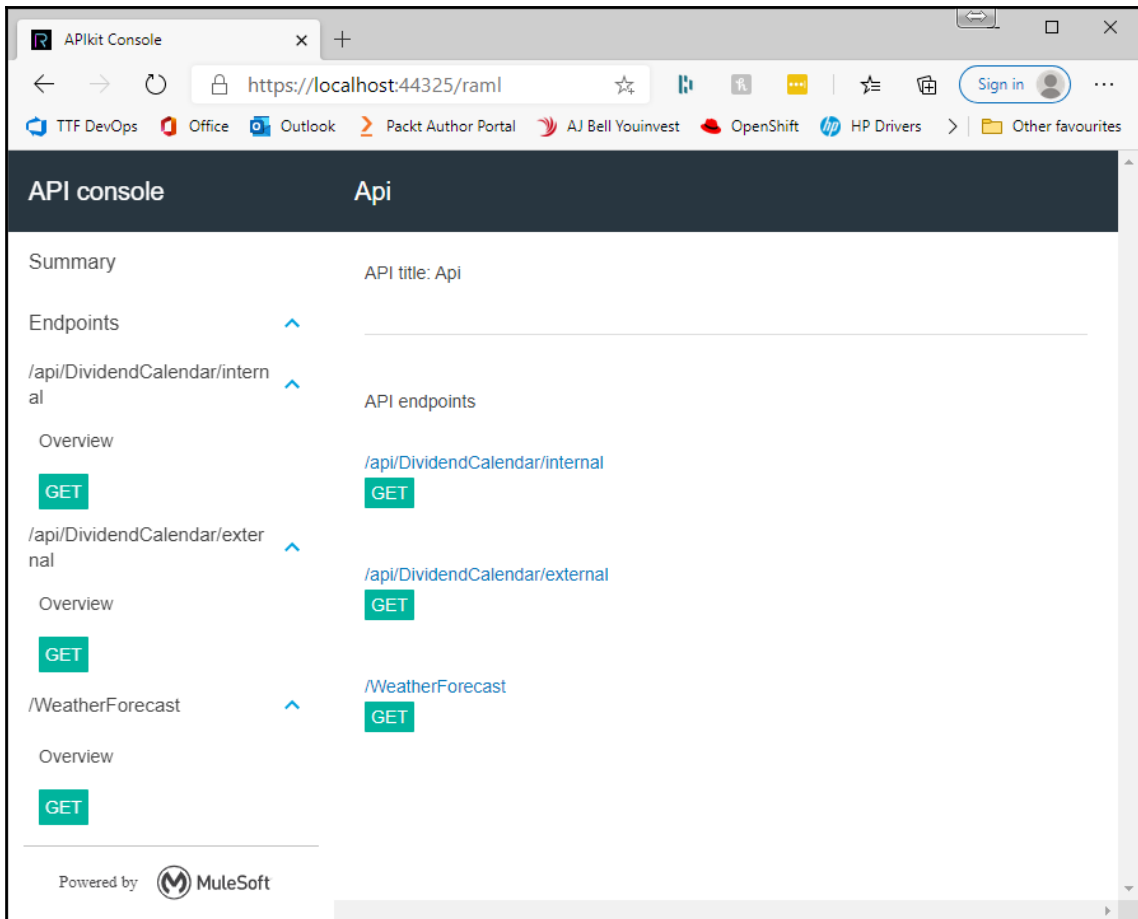
The `Ok()` method returns a type of `OkNegotiatedContentResult<T>`. We will call this `Get()` method from our `Main()` method in the `Program` class. Update the `Main()` method, as shown:

```
static void Main(string[] args)
{
    Task.Run(async () =>
    {
        var hwc = new v1HelloWorldController();
        var response = await hwc.Get() as
        OkNegotiatedContentResult<string>;
        if (response is OkNegotiatedContentResult<string>)
        {
            var msg = response.Content;
            Console.WriteLine($"Message: {msg}");
        }
    }).GetAwaiter().GetResult();
    Console.ReadKey();
}
```

As we are running asynchronous code in a static method, we must add the work to the thread pool queue. We then execute our code and wait for the result. Once the code returns, we simply wait for a keypress and then exit.

We have created an MVC API within a console app and executed API calls based on the RAML file that we imported. This same process works for the ASP.NET and ASP.NET Core websites. We will now extract RAML from an existing API.

Load the dividend calendar API project from earlier on in this chapter. Then, right-click on the project and select **Extract RAML**. Then, once the extraction has finished, run your project. Change the URL to `https://localhost:44325/raml`. When you extract RAML, the code generation process adds a `RamlController` class to your project, along with a RAML view. You will see that your API is now documented, as shown in the RAML view:



By using RAML, you can design an API and then generate the structure and you can reverse engineer an API. The RAML specification helps you design your API and make changes by modifying the RAML code. You can view the <http://raml.org> website for more information on how to get the most out of the RAML specification if you want to know more. We'll now have a look at Swagger and how to use it in ASP.NET Core 3+ projects.

Well, we've now reached the end of this chapter. Now, we will summarize what we have achieved and learned.

Summary

In this chapter, we discussed what an API is. Then, we looked at how we can use API proxies as contracts between ourselves and the consumers of our APIs. This protects our APIs from direct access by third parties. Next, we looked at a number of design guidelines for improving the quality of our APIs.

We then went on to discuss Swagger and saw how to document the Weather API with Swagger. Testing APIs was then covered, and we saw why it is good to test your code and any third-party code that you use in your projects. Finally, we looked at designing a language-agnostic API using RAML and translated it into a working project using C#.

In the next chapter, we will write a project to demonstrate securing keys using Azure Key Vault and securing our own API using API keys. But before then, let's put your brain to work to see what you have learned.

Questions

1. What does API stand for?
2. What does REST stand for?
3. What are the six constraints of REST?
4. What does HATEOAS stand for?
5. What is RAML?
6. What is Swagger?
7. What is meant by the term **well-defined software boundary**?
8. Why should you understand the APIs that you are using?

9. What performs better—structs or objects?
10. Why should you test third-party APIs?
11. Why should you test your own APIs?
12. How can you determine what tests to write for your code?
13. Name three ways to organize code into well-defined software boundaries.

Further reading

- <https://weblogs.asp.net/sukumarraju/asp-net-web-api-testing-using-nunit-framework> provides a complete example of using NUnit to test web APIs.
- <https://raml.org/developers/design-your-api> shows you how to design your API with RAML.
- <http://apiworkbench.com/docs/> provides documentation on using RAML in Atom to design your APIs.
- <https://dotnetcoretutorials.com/2017/10/19/using-swagger-asp-net-core/> is a good introduction to using Swagger.
- <https://swagger.io/about/> takes you to the Swagger **About** page.
- <https://httpstatuses.com/> is a list of HTTP status codes.
- <https://www.greenbytes.de/tech/webdav/rfc5988.html> is the web linking specification RFC 5988.
- <https://oauth.net/2/> takes you to the OAuth 2.0 home page.
- https://en.wikipedia.org/wiki/Domain-driven_design is the Wikipedia page for domain-driven design.
- <https://www.packtpub.com/gb/application-development/hands-domain-driven-design-net-core> provides information on the *Hands-On Domain-Driven Design with .NET Core* book.
- <https://www.packtpub.com/gb/application-development/test-driven-development-c-and-net-core-mvc-video> provides information on *Test-Driven Development with C# and .NET Core and MVC*.

10

Securing APIs with API Keys and Azure Key Vault

In this chapter, we are going to see how we can keep secrets in Azure Key Vault. We will also be looking at how we can use API keys to secure our own keys with authentication and role-based authorization. To gain first-hand experience with API security, we will build a fully functional FinTech API.

Our API will extract third-party API data using a private key (kept safe in Azure Key Vault). We will then secure our API with two API keys; one key will be used internally and a second key will be used by external users.

The following topics are covered in this chapter:

- Accessing the Morningstar API
- Storing the Morningstar API in Azure Key Vault
- Creating the dividend calendar ASP.NET Core web application in Azure
- Publishing our web application
- Using an API key to secure our dividend calendar API
- Testing our API key's security
- Adding the dividend calendar code
- Throttling our API

You will understand the basics of good API design and you will be armed with the knowledge needed to push your API abilities forward. This chapter will assist you in gaining the following skills:

- Securing an API with a client API key
- Storing and retrieving secrets using Azure Key Vault
- Using Postman to execute API commands that post and get data
- Applying for and using third-party APIs on RapidAPI.com
- Throttling API usage
- Writing FinTech APIs that leverage online financial data

Before we continue, make sure you implement the following technical requirements to get the most out of this chapter.

Technical requirements

We will be using the following technologies in this chapter to write an API:

- Visual Studio 2019 Community edition or higher
- Your own personal Morningstar API key from <https://rapidapi.com/integraatio/api/morningstar1>
- RestSharp (<http://restsharp.org/>)
- Swashbuckle.AspNetCore 5 or higher
- Postman (<https://www.postman.com/>)
- Swagger (<https://swagger.io>)

Undertaking the API project – dividend calendar

The best way to learn is by doing. So, we will build a working API and secure it. The API won't be perfect and there will be room for improvement. However, you are free to implement these improvements yourself and expand on the project as you wish. The main goal here is to have a fully functioning API that does one thing: return financial data that lists all the company dividends that will be paid in the current year.

Our dividend calendar API, which we will be building in this chapter, is an API that is authenticated with an **API key**. Depending on what key is used, authorization will determine whether the user is internal or external. The controller will then execute the appropriate method, depending on the type of user. Only the internal user method will be implemented, but you are free to implement the external user method yourself as a training exercise.



The internal method extracts an API key from Azure Key Vault and executes various API calls to a third-party API. The data is returned in **JavaScript Object Notation (JSON)** format, deserialized into objects and then processed to extract future dividend payments, which are added to a list of dividends. This list is then returned to the caller in JSON format. The end result is a JSON file that has all the scheduled dividend payments for the current year. The end user can then take this data and convert it into a list of dividends that can be queried using LINQ.

The project we will be building in this chapter is a web API that returns processed JSON from third-party financial APIs. Our project will obtain a list of companies from a given stock exchange. We will then loop through these companies to obtain their dividend data. The dividend data will then be processed for the current year. So, what we will end up returning to the API caller is JSON data. This JSON data will contain a list of companies and their dividend payment forecast for the current year. The JSON data can then be converted by the end user into C# objects, and LINQ queries can be performed on these objects. Queries can be carried out to get x-dividend payments for the next month or payments due this month, for example.

The APIs that we will be using will be part of the Morningstar API, which is available via RapidAPI.com. You can sign up for a free Morningstar API key. We will secure our API with a login system, where users log in using an email address and password. You will also need Postman, as we will be using it to fire the API's `POST` and `GET` requests to the dividend calendar API.

Our solution will contain a single project, which will be an ASP.NET Core application that targets .NET Framework Core 3.1 or higher. We will now discuss how to access the Morningstar API.

Accessing the Morningstar API

Go to <https://rapidapi.com/integraatio/api/morningstar1> and request an API access key. The API is a Freemium API. This means you are allowed a certain number of calls for free for a limited period, after which you need to pay for its usage. Take some time to look at the API and its documentation. Pay attention to the pricing plans and keep your key a secret when you receive it.

The APIs that we are interested in are as follows:

- `GET /companies/list-by-exchange`: This API returns a list of countries for the specified exchange.
- `GET /dividends`: This API gets all the historical and current dividend payment information for the specified company.

The first part of the API request is the `GET` HTTP verb, which is used to retrieve a resource. The second part of the API request is the resource to `GET`, which in this case is `/companies/list-by-exchange`. As we can see in the second bullet point of the preceding list, we are getting the `/dividends` resource.

You can test each API in the browser and see the data that is returned. I recommend you do this before you continue. This will help you get a feel for what we will be working on. The basic flow we will be using is getting the list of companies that belong to a specified exchange, then looping through them to obtain the dividend data. If the dividend data has a future payment date, then the dividend data will be added to the calendar; otherwise, it will be discarded. No matter how much dividend data exists for a company, we are only interested in the first record, which is the most current one.

Now that you have your API key (assuming you are following along with these steps), we will start to build our API.

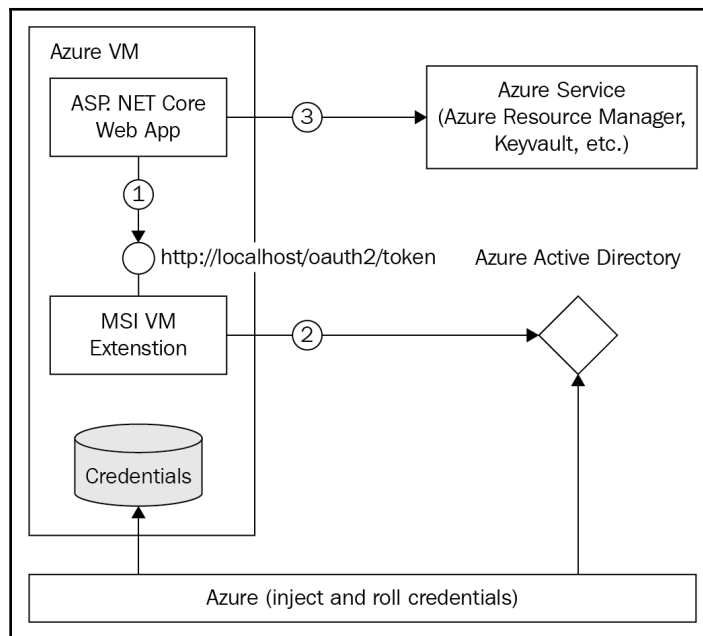
Storing the Morningstar API key in Azure Key Vault

We will be using Azure Key Vault and **Managed Service Identity (MSI)** with an ASP.NET Core web application. So, before you continue, you will need an Azure subscription. For new customers, there is a free 12-month offer available at <http://azure.microsoft.com/en-us/free>.

As web developers, it is important not to store secrets in code because code can be *reverse-engineered*. If code is open source, then there is the danger of uploading personal or enterprise keys to a public version control system. A way around this is to store secrets securely, but this gives rise to a dilemma. To access secret keys, we need to be authenticated. So, how do we overcome this dilemma?

We can overcome this dilemma by enabling MSI for our Azure service. As a result, a service principal is produced by Azure. This service principal is used by applications developed by the user to access resources on Microsoft Azure. For the service principal, you can use a certificate or a username and password, along with any role you choose that has the required set of permissions.

The person who controls the Azure account is in control of what specific tasks each service can perform. It is often best to start with full restrictions and only add capabilities as and when they are needed. The following diagram shows the relationships between our ASP.NET Core web applications, MSI, and our Azure service:



Azure Active Directory (Azure AD) is employed by MSI to inject the service principal for the service instance. An Azure resource known as a **local metadata service** is used to obtain an access token and will be used to authenticate service access to the Azure key vault.

The code then calls a local metadata service that is available on the Azure resource to get the access token. The access token extracted from the local MSI endpoint is then used by our code to authenticate to an Azure Key Vault service.

Open the Azure CLI and type `az login` to log in to Azure. Once we are logged in, we can create a resource group. Azure resource groups are logical containers into which Azure resources are deployed and managed. The following command creates a resource group in the East US location:

```
az group create --name "<YourResourceGroupName>" --location "East US"
```

Use this resource group throughout the rest of the chapter. We will now move on to creating our key vault. The creation of a key vault requires the following information:

- The name of the key vault, which is a string that is between 3 to 24 characters long and can only contain the 0–9, a–z, A–Z, and – (hyphen) characters
- The name of the resource group
- The location—for example, East US or West US

In the Azure CLI, enter the following command:

```
az keyvault create --name "<YourKeyVaultName>" --resource-group  
"<YourResourceGroupName>" --location "East US"
```

Only your Azure account is authorized to perform operations on your new vault at this stage. You can add other accounts if necessary.

The main key that we need to add to our project is `MorningstarApiKey`. To add the Morningstar API key to your key vault, type the following command:

```
az keyvault secret set --vault-name "<YourKeyVaultName>" --name  
"MorningstarApiKey" --value "<YourMorningstarApiKey>"
```

Your key vault now stores your Morningstar API key. To check that the value is stored correctly, type the following command:

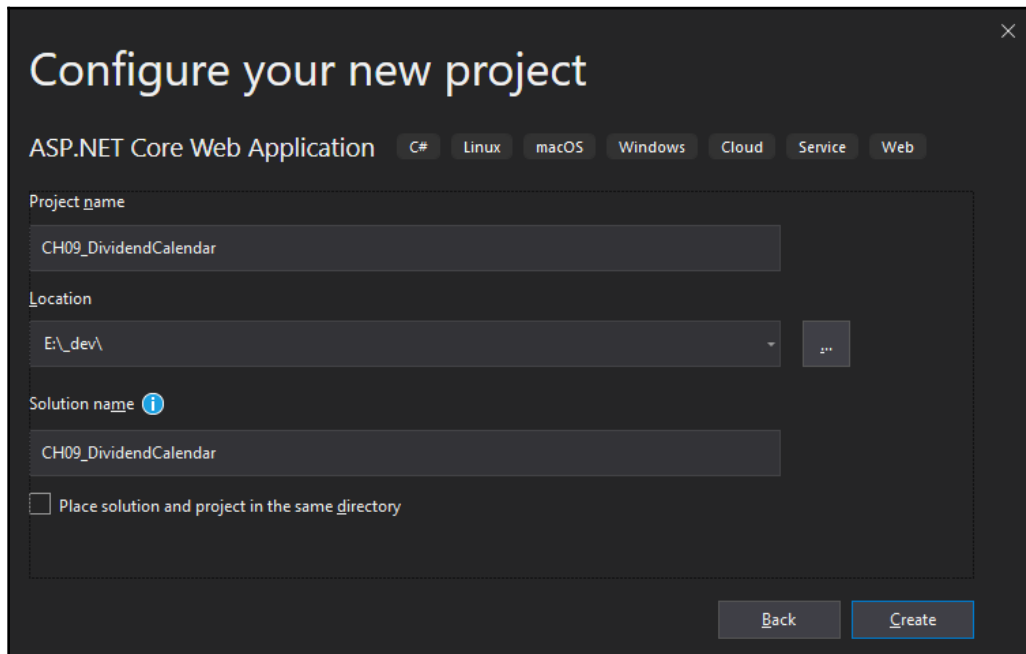
```
az keyvault secret show --name "MorningstarApiKey" --vault-name  
"<YourKeyVaultName>"
```

You should now see your secret displayed in the console window, which shows the key and value for the stored secret.

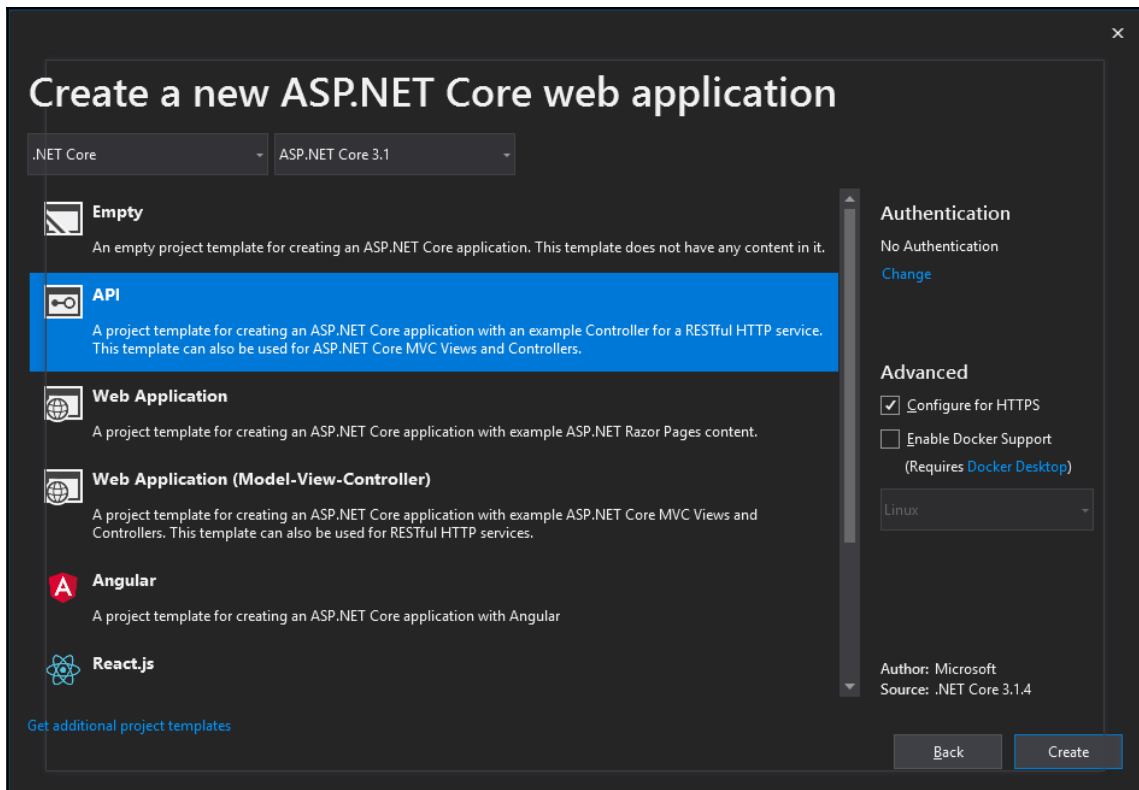
Creating the dividend calendar ASP.NET Core web application in Azure

To complete this stage of the project, you will need Visual Studio 2019 with the ASP.NET and web development workload installed:

1. Create a new ASP.NET Core web application:



2. Make sure **API** is selected with **No Authentication** set:



3. Click on **Create** to scaffold your new project. Then, run your project. By default, an example weather forecast API is defined, and it outputs the following JSON code in the browser window:

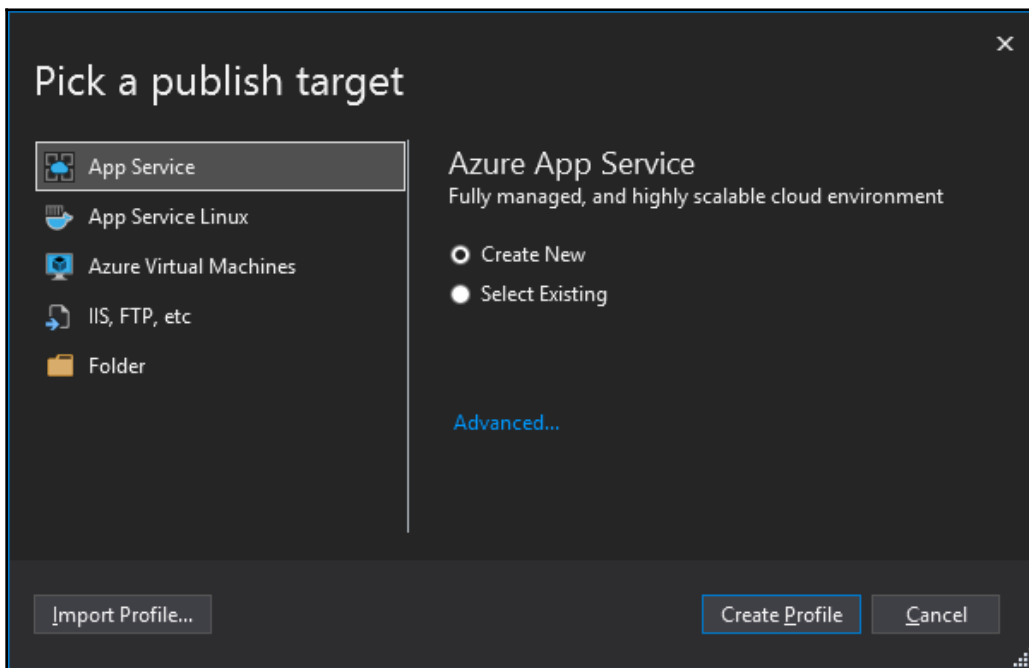
```
[{"date": "2020-04-13T20:02:22.8144942+01:00", "temperatureC": 0, "temperatureF": 32, "summary": "Balmy"}, {"date": "2020-04-14T20:02:22.8234349+01:00", "temperatureC": 13, "temperatureF": 55, "summary": "Warm"}, {"date": "2020-04-15T20:02:22.8234571+01:00", "temperatureC": 3, "temperatureF": 37, "summary": "Scorching"}, {"date": "2020-04-16T20:02:22.8234587+01:00", "temperatureC": -2, "temperatureF": 29, "summary": "Sweltering"}, {"date": "2020-04-17T20:02:22.8234602+01:00", "temperatureC": -13, "temperatureF": 9, "summary": "Cool"}]
```

Next, we will publish our application to Azure.

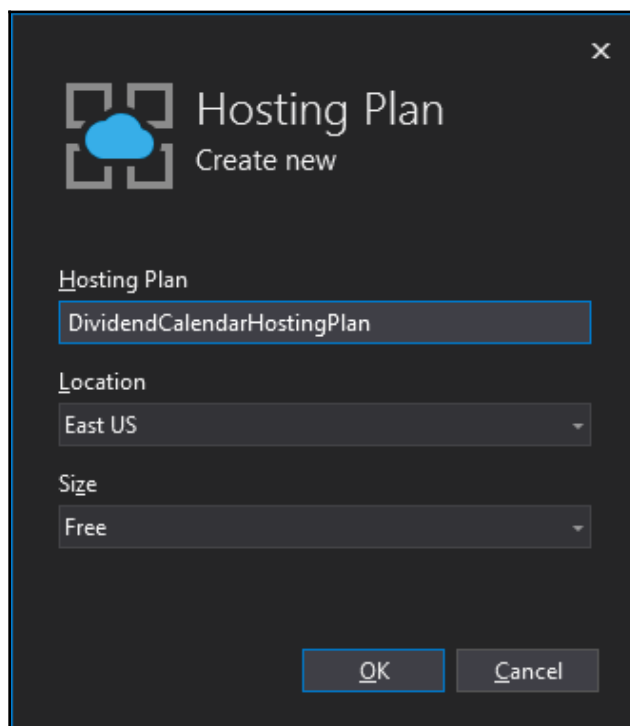
Publishing our web application

Before we can publish our web applications, we will first create a new Azure app service to publish our application to. We will need a resource group to contain our Azure app service, as well as a new hosting plan that specifies the location, size, and features of the web server farm that hosts our application. So, let's take care of these requirements, as follows:

1. Make sure you are signed in to your Azure account from Visual Studio. To create the app service, right-click on the project that you just created and select **Publish** from the menu. This will display the **Pick a publish target** dialog, as shown:



2. Select **App Service | Create New** and click on **Create Profile**. Create a new hosting plan, as in the following example:



Hosting Plan
Create new

Hosting Plan
DividendCalendarHostingPlan

Location
East US

Size
Free

OK Cancel

3. Then, make sure you provide a name, select a subscription, and select your resource group. It is recommended that you also set the **Application Insights** setting:

App Service
Create new

Microsoft account

Name
dividend-calendar

Subscription
Pay-As-You-Go

Resource group
(East US) [New...](#)

Hosting Plan
DividendCalendarHostingPlan* (East US, F1) [New...](#)

Application Insights
East US

[Export...](#)

Explore additional Azure services

- [Create a storage account](#)
- [Create a SQL Database](#)

Clicking the Create button will create the following Azure resources

- Hosting Plan - DividendCalendarHostingPlan
- App Service - dividend-calendar

[Create](#) [Cancel](#)

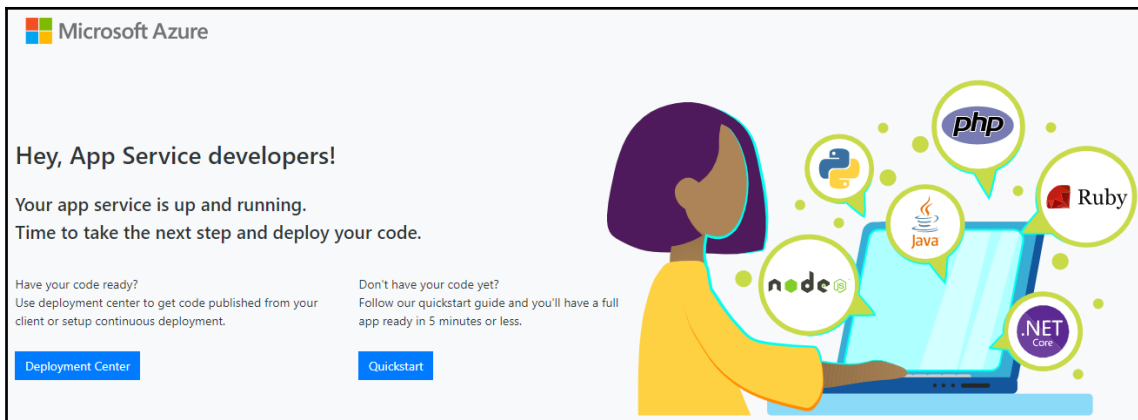
4. Click on **Create** to create your app service. Once it's created, your **Publish** screen should look like this:

The screenshot shows the 'Publish' page in the Azure Portal. At the top, it says 'Publish' and 'Deploy your app to a folder, IIS, Azure, or another destination. [More info](#)'. Below this is a dropdown menu showing 'dividend-calendar - Web Deploy' and a 'Publish' button. Under the dropdown are links for 'New', 'Edit', 'Rename', and 'Delete'. The page is divided into two main sections: 'Summary' and 'Actions'. The 'Summary' section contains a table with the following information:

Site URL	https://dividend-calendar.azurewebsites.net
Resource Group	Keys-APIs
Configuration	Release
Target Framework	netcoreapp3.1
Deployment Mode	Framework-Dependent

The 'Actions' section contains links: 'Preview changes', 'Manage in Cloud Explorer', 'Edit Azure App Service settings', and 'Open troubleshooting guide'. Below these sections is a 'Dependencies' section with a '+ Add' button. It contains the text: 'To see the list of existing dependencies you need to be signed in with the appropriate Azure subscription.' and a 'Sign in' link. At the bottom is a 'Continuous Delivery' section with the text: 'Automatically publish your application to Azure with continuous delivery.' and a 'Configure' link.

5. At this stage, you can click on the site URL. This will load your site URL in the browser. If your service is successfully configured and running, your browser should display the following page:



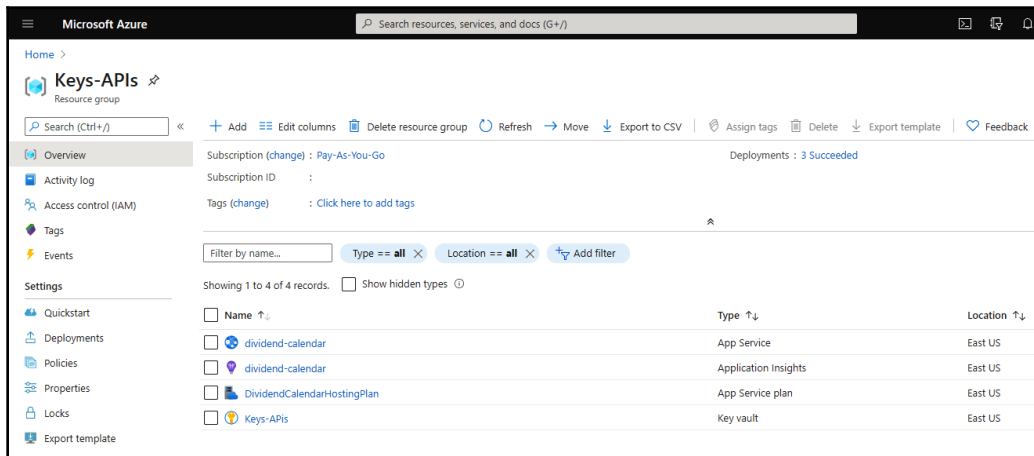
6. Let's publish our API. Click on the **Publish** button. When the web page runs, it will display an error page. Modify the URL to `https://dividend-calendar.azurewebsites.net/weatherforecast`. The web page should now display the weather forecast API JSON code:

```
[{"date":"2020-04-13T19:36:26.9794202+00:00","temperatureC":40,"temperatureF":103,"summary":"Hot"}, {"date":"2020-04-14T19:36:26.9797346+00:00","temperatureC":7,"temperatureF":44,"summary":"Bracing"}, {"date":"2020-04-15T19:36:26.9797374+00:00","temperatureC":8,"temperatureF":46,"summary":"Scorching"}, {"date":"2020-04-16T19:36:26.9797389+00:00","temperatureC":11,"temperatureF":51,"summary":"Freezing"}, {"date":"2020-04-17T19:36:26.9797403+00:00","temperatureC":3,"temperatureF":37,"summary":"Hot"}]
```

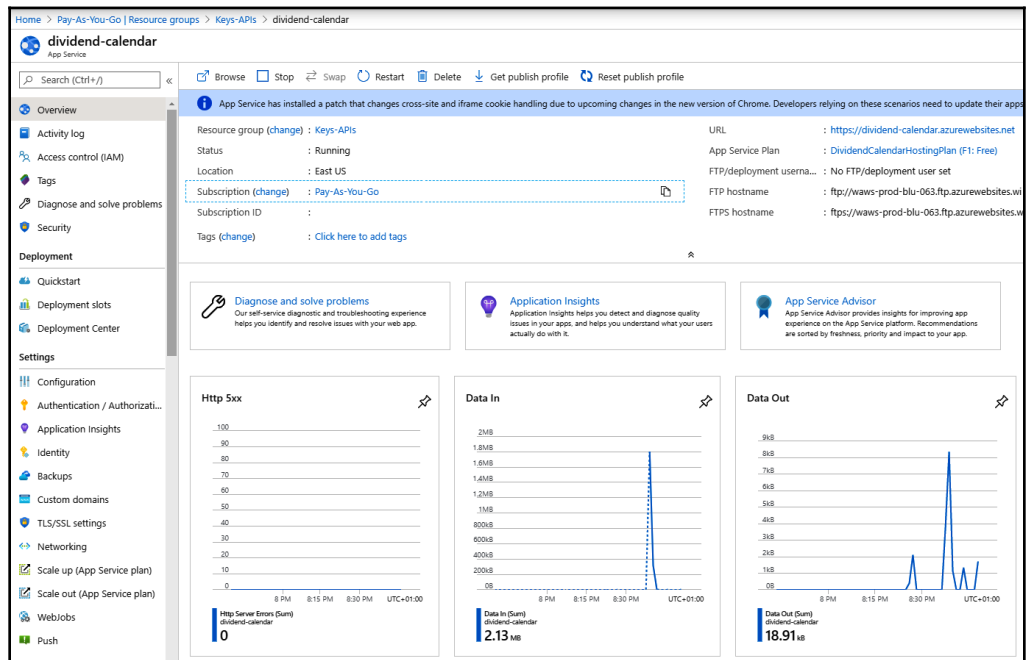
Our service is now live. If you log in to your Azure portal and visit the resource group for your hosting plan, you will see four resources. These resources are as follows:

- **App Service:** dividend-calendar
- **Application Insights:** dividend-calendar
- **App Service plan:** DividendCalendarHostingPlan

- **Key vault:** Whatever your key vault is called. In my case, it's `Keys-APIs`, as shown here:



If you click on your app service from the Azure portal home page (<https://portal.azure.com/#home>), you will see that you can browse to your service, as well as stop, restart, and delete your app service:



Now that we have our project in place with Application Insights and our Morningstar API key is stored securely, we can start building our dividend calendar.

Using an API key to secure our dividend calendar API

To secure access to our dividend calendar API, we are going to use a client API key. There are many ways to share client keys with your clients, but we will not be discussing them here. You can come up with your own strategies. What we will be focusing on is how to enable authenticated and authorized client access to our API.

To keep things simple, we will be using the **repository pattern**. The repository pattern helps to decouple our program from the underlying data store. This pattern improves maintainability and allows you to change the underlying data store without affecting the program. For our repository, our keys will be defined in a class, but in a commercial project, you would store the keys in a data store, such as Cosmos DB, SQL Server, or Azure Key Vault. You decide the strategy that best suits your needs, which is the main reason why we use the repository pattern as you are in control of the underlying data source for your own needs.

Setting up the repository

We are going to start by setting up our repository:

1. Add a new folder to your project called `Repository`. Then, add a new interface called `IRepository` and a class that will implement `IRepository`, called `InMemoryRepository`. Modify your interface, as follows:

```
using CH09_DividendCalendar.Security.Authentication;
using System.Threading.Tasks;

namespace CH09_DividendCalendar.Repository
{
    public interface IRepository
    {
        Task<ApiKey> GetApiKey(string providedApiKey);
    }
}
```


2. This interface defines one method for retrieving the API key. We have not yet defined the `ApiKey` class and we will do that later. Now, let's implement `InMemoryRepository`. Add the following using statements:

```
using CH09_DividendCalendar.Security.Authentication;
using CH09_DividendCalendar.Security.Authorisation;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

3. The security namespaces will be created when we start adding the authentication and authorization classes. Modify the `Repository` class to implement the `IRepository` interface. Add the member variable that will hold our API keys, and then add the `GetApiKey()` method:

```
public class InMemoryRepository : IRepository
{
    private readonly IDictionary<string, ApiKey> _apiKeys;

    public Task<ApiKey> GetApiKey(string providedApiKey)
    {
        _apiKeys.TryGetValue(providedApiKey, out var key);
        return Task.FromResult(key);
    }
}
```

4. The `InMemoryRepository` class implements the `GetApiKey()` method of `IRepository`. This returns a dictionary of API keys. These keys will be stored in our `_apiKeys` dictionary member variable. Now, we'll add our constructor:

```
public InMemoryRepository()
{
    var existingApiKeys = new List<ApiKey>
    {
        new ApiKey(1, "Internal", "C5BFF7F0-B4DF-475E-A331-F737424F013C", new DateTime(2019, 01, 01),
            new List<string>
            {
                Roles.Internal
            }
        ),
        new ApiKey(2, "External", "9218FACE-3EAC-6574-C3F0-08357FEDABE9", new DateTime(2020, 4, 15),
            new List<string>
            {
                Roles.External
            }
        )
    };
    _apiKeys = new Dictionary<string, ApiKey>();
    foreach (var apiKey in existingApiKeys)
    {
        _apiKeys[apiKey.Id] = apiKey;
    }
}
```

```

        })
    };

    _apiKeys = existingApiKeys.ToDictionary(x => x.Key, x => x);
}

```

5. Our constructor creates a new list of API keys. It creates an internal API key for internal use only and an external API key for external use only. It then converts the list into a dictionary and stores the dictionary in `_apiKeys`. So, we now have our repository in place.
6. We will be using an HTTP header called `X-Api-Key`. This will store the client's API key, which will be passed into our API for authentication and authorization. Add a new folder to the project called `Shared`, and then add a new file called `ApiKeyConstants`. Update the file with the following code:

```

namespace CH09_DividendCalendar.Shared
{
    public struct ApiKeyConstants
    {
        public const string HeaderName = "X-Api-Key";
        public const string MorningstarApiKeyUrl
            =
            "https://<YOUR_KEY_VAULT_NAME>.vault.azure.net/secrets/MorningstarA
            piKey";
    }
}

```

This file contains two constants—the header name, which will be used when establishing the user's identity, and the URL for the Morningstar API key, which is stored in the Azure key vault that we created earlier.

7. Since we will be handling JSON data, we need to set our JSON naming policy. Add a folder to your project called `Json`. Then, add a class called `DefaultJsonSerializerOptions`:

```

using System.Text.Json;

namespace CH09_DividendCalendar.Json
{
    public static class DefaultJsonSerializerOptions
    {
        public static JsonSerializerOptions Options => new
        JsonSerializerOptions
        {
            PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
            IgnoreNullValues = true
        }
    }
}

```

```
        };  
    }  
}
```

The `DefaultJsonSerializerOptions` class sets our JSON naming policy to ignore null values and to use camel case names.

We will now start adding authentication and authorization to our API.

Setting up authentication and authorization

We will now start work on the security classes for authentication and authorization. It is good to clarify what we mean by authentication and authorization first. Authentication is establishing whether the user is authorized to access our API. Authorization is establishing what permissions the user has once they gain access to our API.

Adding authentication

Before we continue, add a `Security` folder to your project and then under that folder, add the `Authentication` and `Authorisation` folders. We will start by adding our `Authentication` classes; the first class that we will add to our `Authentication` folder is `ApiKey`. Add the following properties to `ApiKey`:

```
public int Id { get; }  
public string Owner { get; }  
public string Key { get; }  
public DateTime Created { get; }  
public IReadOnlyCollection<string> Roles { get; }
```

These properties store information that pertains to the specified API key and its owner. The properties are set via the constructor:

```
public ApiKey(int id, string owner, string key, DateTime created,  
IReadOnlyCollection<string> roles)  
{  
    Id = id;  
    Owner = owner ?? throw new ArgumentNullException(nameof(owner));  
    Key = key ?? throw new ArgumentNullException(nameof(key));  
    Created = created;  
    Roles = roles ?? throw new ArgumentNullException(nameof(roles));  
}
```

The constructor sets the API key properties. If a person fails authentication, then they will be notified with an `Error 403 Unauthorized` message. So, let's now define our `UnauthorizedProblemDetails` class:

```
public class UnauthorizedProblemDetails : ProblemDetails
{
    public UnauthorizedProblemDetails(string details = null)
    {
        Title = "Forbidden";
        Detail = details;
        Status = 403;
        Type = "https://httpstatuses.com/403";
    }
}
```

This class inherits the `Microsoft.AspNetCore.Mvc.ProblemDetails` class. The constructor takes a single parameter of the string type, which defaults to `null`. You can pass details into this constructor to provide more information if required. Next, we add `AuthenticationBuilderExtensions`:

```
public static class AuthenticationBuilderExtensions
{
    public static AuthenticationBuilder AddApiKeySupport(
        this AuthenticationBuilder authenticationBuilder,
        Action<ApiKeyAuthenticationOptions> options
    )
    {
        return authenticationBuilder
            .AddScheme<ApiKeyAuthenticationOptions,
                ApiKeyAuthenticationHandler>
                (ApiKeyAuthenticationOptions.DefaultScheme, options);
    }
}
```

This extension method adds API key support to the authentication service, which will be set in the `ConfigureServices` method of the `Startup` class. Now, add the `ApiKeyAuthenticationOptions` class:

```
public class ApiKeyAuthenticationOptions : AuthenticationSchemeOptions
{
    public const string DefaultScheme = "API Key";
    public string Scheme => DefaultScheme;
    public string AuthenticationType = DefaultScheme;
}
```

The `ApiKeyAuthenticationOptions` class inherits the `AuthenticationSchemeOptions` class. We set the default scheme to use API key authentication. The final part of our authorization is to build up our `ApiKeyAuthenticationHandler` class. As the name suggests, this is the main class for validating the API key, ensuring the client is authorized to access and use our API:

```
public class ApiKeyAuthenticationHandler :
    AuthenticationHandler<ApiKeyAuthenticationOptions>
{
    private const string ProblemDetailsContentType =
        "application/problem+json";
    private readonly IRepository _repository;
}
```

Our `ApiKeyAuthenticationHandler` class inherits from `AuthenticationHandler` and uses `ApiKeyAuthenticationOptions`. We define the content type for the problem details (exception information) as `application/problem+json`. We also provide a placeholder for our API key repository using the `_repository` member variable. The next step is to declare our constructor:

```
public ApiKeyAuthenticationHandler(
    IOptionMonitor<ApiKeyAuthenticationOptions> options,
    ILoggerFactory logger,
    UrlEncoder encoder,
    ISystemClock clock,
    IRepository repository
) : base(options, logger, encoder, clock)
{
    _repository = repository ?? throw new
        ArgumentNullException(nameof(repository));
}
```

Our constructor passes the `ApiKeyAuthenticationOptions`, `ILoggerFactory`, `UrlEncoder`, and `ISystemClock` parameters to the base class. Explicitly, we set the repository. If the repository is null, we throw a null argument exception with the name of the repository. Let's add our `HandleChallengeAsync()` method:

```
protected override async Task HandleChallengeAsync(AuthenticationProperties
properties)
{
    Response.StatusCode = 401;
    Response.ContentType = ProblemDetailsContentType;
    var problemDetails = new UnauthorizedProblemDetails();
    await Response.WriteAsync(JsonSerializer.Serialize(problemDetails,
        DefaultJsonSerializerOptions.Options));
}
```

The `HandleChallengeAsync()` method returns an `Error 401 Unauthorized` response when the user challenge fails. Now, let's add our `HandleForbiddenAsync()` method:

```
protected override async Task HandleForbiddenAsync(AuthenticationProperties
properties)
{
    Response.StatusCode = 403;
    Response.ContentType = ProblemDetailsContentType;
    var problemDetails = new ForbiddenProblemDetails();
    await Response.WriteAsync(JsonSerializer.Serialize(problemDetails,
        DefaultJsonSerializerOptions.Options));
}
```

The `HandleForbiddenAsync()` method returns an `Error 403 Forbidden` response when the user permission check fails. Now, we need to add a final method that returns `AuthenticationResult`:

```
protected override async Task<AuthenticateResult> HandleAuthenticateAsync()
{
    if (!Request.Headers.TryGetValue(ApiKeyConstants.HeaderName, out var
apiKeyHeaderValues))
        return AuthenticateResult.NoResult();
    var providedApiKey = apiKeyHeaderValues.FirstOrDefault();
    if (apiKeyHeaderValues.Count == 0 ||
string.IsNullOrWhiteSpace(providedApiKey))
        return AuthenticateResult.NoResult();
    var existingApiKey = await _repository.GetApiKey(providedApiKey);
    if (existingApiKey != null) {
        var claims = new List<Claim> {new Claim(ClaimTypes.Name,
existingApiKey.Owner)};
        claims.AddRange(existingApiKey.Roles.Select(role => new
Claim(ClaimTypes.Role, role)));
        var identity = new ClaimsIdentity(claims,
Options.AuthenticationType);
        var identities = new List<ClaimsIdentity> { identity };
        var principal = new ClaimsPrincipal(identities);
        var ticket = new AuthenticationTicket(principal, Options.Scheme);
        return AuthenticateResult.Success(ticket);
    }
    return AuthenticateResult.Fail("Invalid API Key provided.");
}
```

The code we've just written checks whether our header exists. If the header is not present, then `AuthenticateResult()` returns a Boolean value of `true` for the `None` property, indicating that there was no information provided for this request. We then check whether the header has a value. If no value is provided, the `return` value indicates that no information was provided for this request. We then obtain our server-side key from our repository using the client-side key.

If the server-side key is null, then a failed `AuthenticationResult()` instance is returned, indicating that the provided API key is invalid, as identified in the `Failure` property of the `Exception` type. Otherwise, the user is deemed authentic and is allowed to access our API. For valid users, we set the claims for their identities and then pass back a successful `AuthenticateResult()` instance.

So, we have our authentication sorted. Now, we need to work on our authorization.

Adding authorization

Our authorization classes will be added to the `Authorisation` folder. Add the `Roles` struct with the following code:

```
public struct Roles
{
    public const string Internal = "Internal";
    public const string External = "External";
}
```

We expect our API to be used internally and externally. However, for our minimum viable product, only the code for internal users will be implemented. Now, add the `Policies` struct:

```
public struct Policies
{
    public const string Internal = nameof(Internal);
    public const string External = nameof(External);
}
```

In our `Policies` structure, we have added two policies that will be used for internal and external clients. Now, we'll add the `ForbiddenProblemDetails` class:

```
public class ForbiddenProblemDetails : ProblemDetails
{
    public ForbiddenProblemDetails(string details = null)
    {
        Title = "Forbidden";
        Detail = details;
        Status = 403;
        Type = "https://httpstatuses.com/403";
    }
}
```

This class provides the forbidden problem details if one or more permissions are not available to the authenticated user. You can pass a string into this class's constructor to provide relevant information if required.

For our authorization, we will need to add authorization requirements and handlers for both internal and external clients. We'll add the `ExternalAuthorisationHandler` class first:

```
public class ExternalAuthorisationHandler :
    AuthorizationHandler<ExternalRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        ExternalRequirement requirement
    )
    {
        if (context.User.IsInRole(Roles.External))
            context.Succeed(requirement);
        return Task.CompletedTask;
    }

    public class ExternalRequirement : IAuthorizationRequirement
    {
    }
}
```


The `ExternalRequirement` class is an empty class that implements the `IAuthorizationRequirement` interface. Now, add the `InternalAuthorisationHandler` class:

```
public class InternalAuthorisationHandler :
    AuthorizationHandler<InternalRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        InternalRequirement requirement
    )
    {
        if (context.User.IsInRole(Roles.Internal))
            context.Succeed(requirement);
        return Task.CompletedTask;
    }
}
```

The `InternalAuthorisationHandler` class handles the authorization of the internal requirement. If the context user is assigned to the internal role, then permission is granted. Otherwise, permission is denied. Let's add the required `InternalRequirement` class:

```
public class InternalRequirement : IAuthorizationRequirement
{
}
```

Here, the `InternalRequirement` class is an empty class that implements the `IAuthorizationRequirement` interface.

We now have our authentication and authorization classes in place. So, it is now time to update our `Startup` class to wire up the security classes. Start by modifying the `Configure()` method:

```
public void Configure(IApplicationBuilder app, IHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

The `Configure()` method sets the exception page to the developer page if we are in development. It then requests the app to use *routing* to match URIs with the actions in our controllers. The app is then informed that it should use our authentication and authorization methods. Finally, the application endpoints are mapped from the controllers.

The final method we need to update to complete our API key authentication and authorization is the `ConfigureServices()` method. The first thing we need to do is add our authentication service with API key support:

```
services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme =
    ApiKeyAuthenticationOptions.DefaultScheme;
    options.DefaultChallengeScheme =
    ApiKeyAuthenticationOptions.DefaultScheme;
}).AddApiKeySupport(options => { });
```

Here, we are setting the default authentication scheme. We add `AddApiKeySupport()` using our extension key, as defined in our `AuthenticationBuilderExtensions` class, which returns; `Microsoft.AspNetCore.Authentication.AuthenticationBuilder`. Our default scheme is set to the API key, as configured in our `ApiKeyAuthenticationOptions` class. The API key is a constant value that informs the authentication service that we will be using API key authentication. Now, we need to add our authorization service:

```
services.AddAuthorization(options =>
{
    options.AddPolicy(Policies.Internal, policy =>
    policy.Requirements.Add(new InternalRequirement()));
    options.AddPolicy(Policies.External, policy =>
    policy.Requirements.Add(new ExternalRequirement()));
});
```

Here, we are setting our internal and external policies and requirements. These are defined in our `Policies`, `InternalRequirement`, and `ExternalRequirement` classes.

Well, we've added all of our API key security classes. So, we can now test whether our API key authentication and authorization is working using Postman.

Testing our API key security

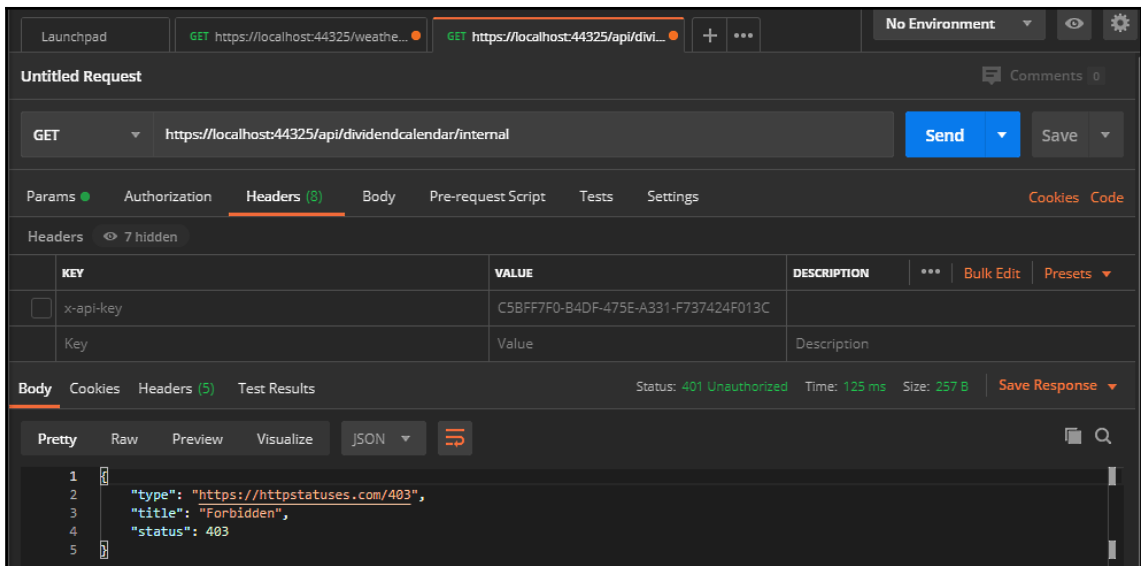
In this section, we are going to test our API key authentication and authorization using Postman. Add a class to your `Controllers` folder called `DividendCalendar`. Update the class as follows:

```
[ApiController]
[Route("api/[controller]")]
public class DividendCalendar : ControllerBase
{
    [Authorize(Policy = Policies.Internal)]
    [HttpGet("internal")]
    public IActionResult GetDividendCalendar()
    {
        var message = $"Hello from {nameof(GetDividendCalendar)}.";
        return new ObjectResult(message);
    }

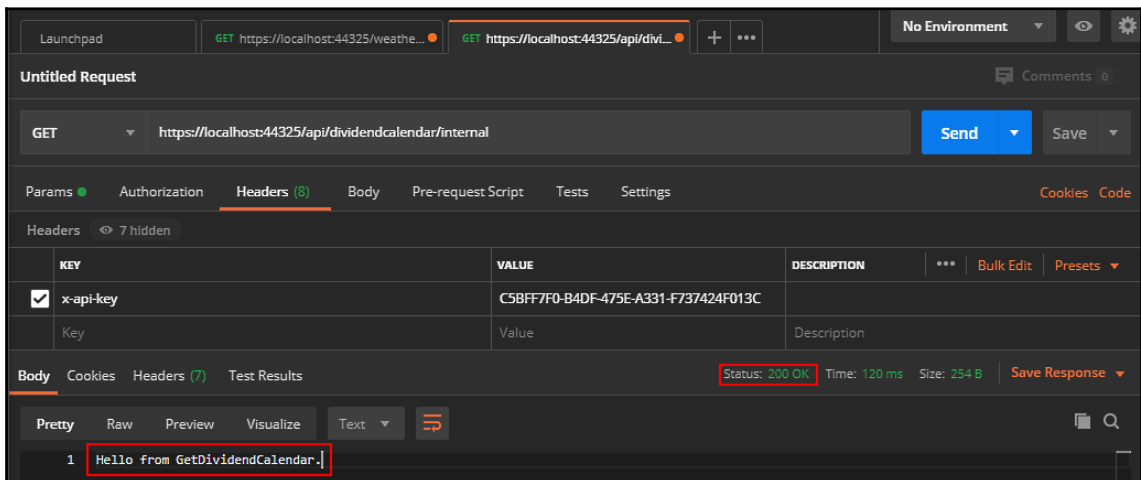
    [Authorize(Policy = Policies.External)]
    [HttpGet("external")]
    public IActionResult External()
    {
        var message = "External access is currently unavailable.";
        return new ObjectResult(message);
    }
}
```

This class will contain all of our dividend calendar API code functionality. Even though external code will not be used in this initial release of our minimum viable product, we will be able to test our internal and external authentication and authorization.

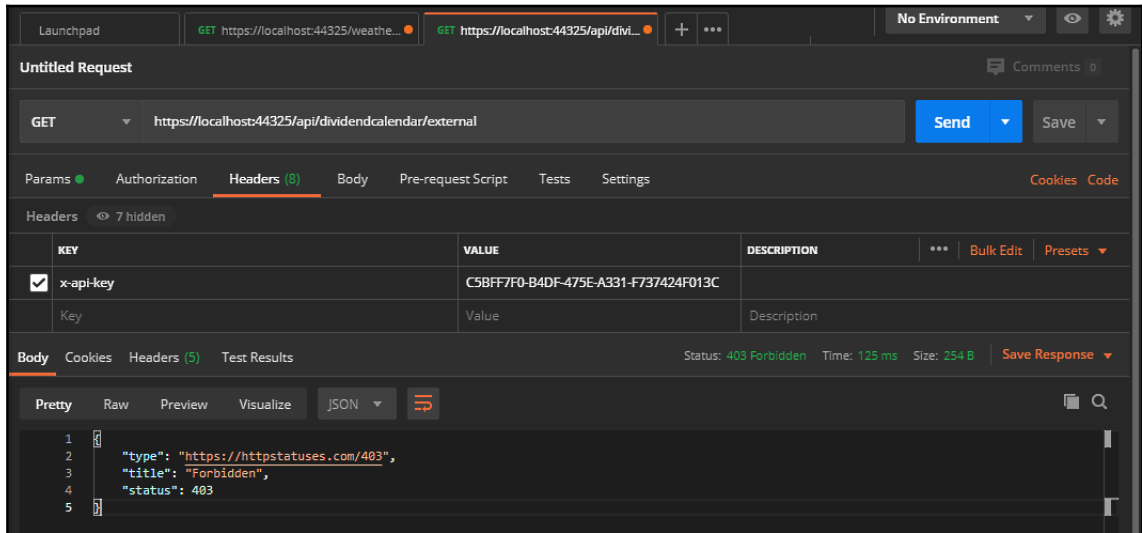
1. Open Postman and create a new GET request. For the URL, use `https://localhost:44325/api/dividendcalendar/internal`. Click **Send**:



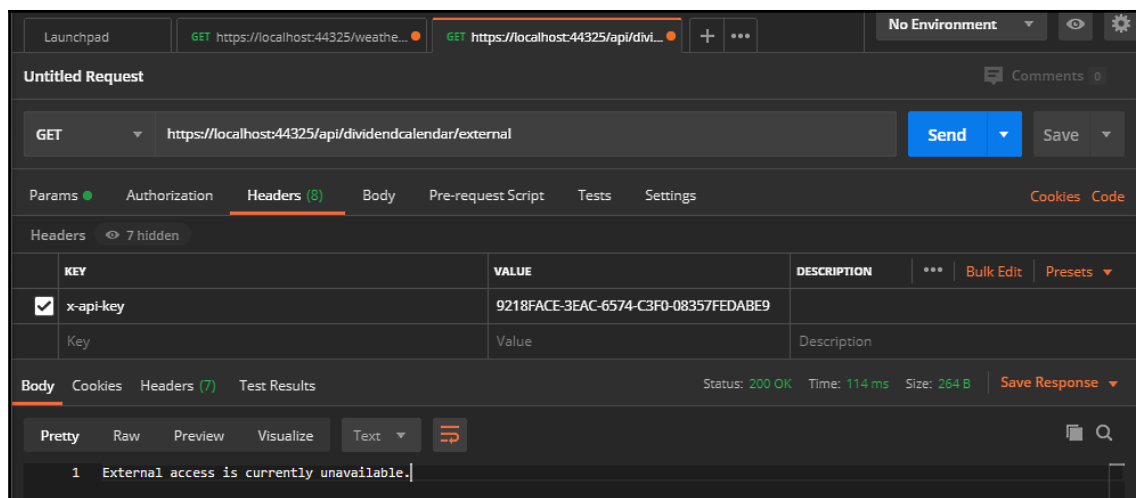
2. As you can see, without the API key present in the API request, we get the expected 401 Unauthorized status with our forbidden JSON, as defined in the `ForbiddenProblemDetails` class. Now, add the `x-api-key` header with the `C5BFF7F0-B4DF-475E-A331-F737424F013C` value. Then, click **Send**:



3. You will now have a status of 200 OK. This means that the API request has been successful. You can see the result of the request in the body. Internal users will see Hello from GetDividendCalendar. Run the request again, but change the URL so that the route is external instead of internal. So, the URL should be `https://localhost:44325/api/dividendcalendar/external`:



4. You should receive a status of 403 Forbidden with the forbidden JSON. This is because the API key is a valid API key, but the route is for an external client and the external client does not have access to the internal API. Change the `x-api-key` header value to `9218FACE-3EAC-6574-C3F0-08357FEDABE9`. Then, click **Send**:



You will see that you have a status of 200 OK and that the body has the External access is currently unavailable text.

Good news! Our role-based security system using API key authentication and authorization is tested and working. So, before we have even added our actual FinTech API, we have implemented and tested our API key, which is used to secure our FinTech API. So, we have put the security of our API first before writing a single line of our actual API. Now, we can start in earnest to build our dividend calendar API functionality, knowing that it is secure.

Adding the dividend calendar code

Our internal API only has one purpose, which is to build up an array of dividends that are to be paid out this year. You, however, can build on this project to save the JSON to a file or database of some type. So, you would only make an internal call once a month to save money on API calls. However, the external role could access the data from your file or database as often as needed.

We already have our controller in place for our dividend calendar API. This security is in place to prevent unauthenticated and unauthorized users from accessing our internal `GetDividendCalendar()` API endpoint. So, all we have to do now is generate the dividend calendar JSON, which our method will return.

So that you can see what we will be working toward, have a look at the following truncated JSON response:

```
[{"Mic": "XLON", "Ticker": "ABDP", "CompanyName": "AB Dynamics  
PLC", "DividendYield": 0.0, "Amount": 0.0279, "ExDividendDate": "2020-01-02T00:00:00", "DeclarationDate": "2019-11-27T00:00:00", "RecordDate": "2020-01-03T00:00:00", "PaymentDate": "2020-02-13T00:00:00", "DividendType": null, "CurrencyCode": null},  
  
...  
  
{"Mic": "XLON", "Ticker": "ZYT", "CompanyName": "Zytronic  
PLC", "DividendYield": 0.0, "Amount": 0.152, "ExDividendDate": "2020-01-09T00:00:00", "DeclarationDate": "2019-12-10T00:00:00", "RecordDate": "2020-01-10T00:00:00", "PaymentDate": "2020-02-07T00:00:00", "DividendType": null, "CurrencyCode": null}]
```

This JSON response is an array of dividends. A dividend consists of the `Mic`, `Ticker`, `CompanyName`, `DividendYield`, `Amount`, `ExDividendDate`, `DeclarationDate`, `RecordDate`, `PaymentDate`, `DividendType`, and `CurrencyCode` fields. Add a new folder to your project called `Models`, and then add the `Dividend` class with the following code:

```
public class Dividend  
{  
    public string Mic { get; set; }  
    public string Ticker { get; set; }  
    public string CompanyName { get; set; }  
    public float DividendYield { get; set; }  
    public float Amount { get; set; }  
    public DateTime? ExDividendDate { get; set; }  
    public DateTime? DeclarationDate { get; set; }  
    public DateTime? RecordDate { get; set; }  
    public DateTime? PaymentDate { get; set; }  
    public string DividendType { get; set; }  
    public string CurrencyCode { get; set; }  
}
```

Let's see what each of these fields represents:

- **Mic: ISO 10383 Market Identification Code (MIC)**, which is where the stock is listed. See <https://www.iso20022.org/10383/iso-10383-market-identifier-codes> for more information.
- **Ticker:** The stock market ticker for the common stock.
- **CompanyName:** The name of the company that owns the stock.
- **DividendYield:** The ratio of the company's annual dividend compared to its share price. The dividend yield is calculated in terms of percentage and is calculated with the *Dividend Yield = Annual Dividend / Share Price* formula.
- **Amount:** The amount that will be paid out to the shareholder per share.
- **ExDividendDate:** The date before which you must purchase the share in order to receive the next dividend payment.
- **DeclarationDate:** The date that the company declares they are paying a dividend.
- **RecordDate:** The date that the company looks at its records to determine who will receive the dividend.
- **PaymentDate:** The date that the shareholders receives the dividend payment.
- **DividendType:** This can be, for example, Cash Dividend, Property Dividend, Stock Dividend, Scrip Dividend, or Liquidating Dividend.
- **CurrencyCode:** The currency that the amount will be paid in.

The next class we need in our `Models` folder is the `Company` class:

```
public class Company
{
    public string MIC { get; set; }
    public string Currency { get; set; }
    public string Ticker { get; set; }
    public string SecurityId { get; set; }
    public string CompanyName { get; set; }
}
```



The `Mic` and `Ticker` fields are the same as for our `Dividend` class. Between the different API calls, the APIs use different names for the currency identifier. That is why we have `CurrencyCode` in `Dividend` and `Currency` in `Company`. This helps the JSON with the object-mapping process so that we don't experience formatting exceptions.

Each of these fields represents the following:

- **Currency:** The currency used to price the stock
- **SecurityId:** The stock market security identifier for the common stock
- **CompanyName:** The name of the company that owns the stock

Our next `Models` class is called `Companies`. This class is required to store the companies that are returned in the initial Morningstar API call. We will be looping through the list of companies to make further API calls to get each company's record so that we can then make our API call to get the company's dividend:

```
public class Companies
{
    public int Total { get; set; }
    public int Offset { get; set; }
    public List<Company> Results { get; set; }
    public string ResponseStatus { get; set; }
}
```

Each of these properties defines the following:

- **Total:** The total number of records returned from the API query
- **Offset:** The record offset
- **Results:** The list of companies returned
- **ResponseStatus:** Provides detailed response information, especially if errors are returned

Now, we will add the `Dividends` class. This class holds the list of dividends returned by the dividends' Morningstar API response:

```
public class Dividends
{
    public int Total { get; set; }
    public int Offset { get; set; }
    public List<Dictionary<string, string>> Results { get; set; }
    public ResponseStatus ResponseStatus { get; set; }
}
```

Each of these properties is the same as those defined previously, except for the `Results` property, which defines a list of dividend payments returned for the specified company.

The final class we need to add to our `Models` folder is the `ResponseStatus` class. This is mainly used to store error information:

```
public class ResponseStatus
{
    public string ErrorCode { get; set; }
    public string Message { get; set; }
    public string StackTrace { get; set; }
    public List<Dictionary<string, string>> Errors { get; set; }
    public List<Dictionary<string, string>> Meta { get; set; }
}
```

The properties of this class are as follows:

- `ErrorCode`: The number of the error
- `Message`: The error message
- `StackTrace`: The error diagnostics
- `Errors`: A list of errors
- `Meta`: A list of the error metadata

We now have all the models we need in place. So now, we can start to make our API calls to build up our dividend payment calendar. In the controller, add a new method called `FormatStringDate()`, as follows:

```
private DateTime? FormatStringDate(string date)
{
    return string.IsNullOrEmpty(date) ? (DateTime?)null :
    DateTime.Parse(date);
}
```

This method takes a string date. If the string is null or empty, then `null` is returned. Otherwise, the string is parsed and a nullable `DateTime` value is passed back. We'll also need a method that extracts our Morningstar API key from our Azure key vault:

```
private async Task<string> GetMorningstarApiKey()
{
    try
    {
        AzureServiceTokenProvider azureServiceTokenProvider = new
        AzureServiceTokenProvider();
        KeyVaultClient keyVaultClient = new KeyVaultClient(
            new KeyVaultClient.AuthenticationCallback(
                azureServiceTokenProvider.KeyVaultTokenCallback
            )
        );
    }
}
```

```
        var secret = await
keyVaultClient.GetSecretAsync(ApiKeyConstants.MorningstarApiKeyUrl)
                                                    .ConfigureAwait(false);
        return secret.Value;
    }
    catch (KeyVaultErrorException keyVaultException)
    {
        return keyVaultException.Message;
    }
}
```

The `GetMorningstarApiKey()` method instantiates `AzureServiceTokenProvider`. It then creates a new `KeyVaultClient` object type, which performs cryptographic key operations. Then, the method awaits the response to get the Morningstar API key from the Azure key vault. It then passes back the response value. If an error occurs processing a request, `KeyVaultErrorException.Message` is returned.

When processing the dividends, we first obtain a list of companies from a stock exchange. We then loop through these companies and make another call to get the dividends for each company in that stock exchange. So, we'll start with our method to obtain a list of companies by MIC. Remember, we are using the `RestSharp` library. So, if you have not already installed it, then now is a good time to do so:

```
private Companies GetCompanies(string mic)
{
    var client = new RestClient(
    $"https://morningstar1.p.rapidapi.com/companies/list-by-exchange?Mic={mic}"
    );
    var request = new RestRequest(Method.GET);
    request.AddHeader("x-rapidapi-host", "morningstar1.p.rapidapi.com");
    request.AddHeader("x-rapidapi-key", GetMorningstarApiKey().Result);
    request.AddHeader("accept", "string");
    IRestResponse response = client.Execute(request);
    return JsonConvert.DeserializeObject<Companies>(response.Content);
}
```

Our `GetCompanies()` method creates a new REST client that points to the API URL that retrieves a list of companies that are listed on the specified stock exchange. The type of the request is a GET request. We add three headers to our GET request for `x-rapidapi-host`, `x-rapidapi-key`, and `accept`. Then, we execute the request and return the deserialized JSON data via the `Companies` model.

Now, we will write the methods that return the dividends for the specified exchange and company. Let's start by adding the `GetDividends()` method:

```
private Dividends GetDividends(string mic, string ticker)
{
    var client = new RestClient(
        $"https://morningstar1.p.rapidapi.com/dividends?Ticker={ticker}&Mic={mic}"
    );
    var request = new RestRequest(Method.GET);
    request.AddHeader("x-rapidapi-host", "morningstar1.p.rapidapi.com");
    request.AddHeader("x-rapidapi-key", GetMorningstarApiKey().Result);
    request.AddHeader("accept", "string");
    IRestResponse response = client.Execute(request);
    return JsonConvert.DeserializeObject<Dividends>(response.Content);
}
```

The `GetDividends()` method is the same as the `GetCompanies()` method, except the request returns the dividends for the specified stock exchange and company. The JSON is deserialized into an instance of the `Dividends` object and is returned.

For our final method, we need our minimum viable product to be built into the `BuildDividendCalendar()` method. This method is the method that will build up the dividend calendar JSON that will be returned to the client:

```
private List<Dividend> BuildDividendCalendar()
{
    const string MIC = "XLON";
    var thisYearsDividends = new List<Dividend>();
    var companies = GetCompanies(MIC);
    foreach (var company in companies.Results) {
        var dividends = GetDividends(MIC, company.Ticker);
        if (dividends.Results == null)
            continue;
        var currentDividend = dividends.Results.FirstOrDefault();
        if (currentDividend == null || currentDividend["payableDt"] ==
null)
            continue;
        var dateDiff = DateTime.Compare(
            DateTime.Parse(currentDividend["payableDt"]),
            new DateTime(DateTime.Now.Year - 1, 12, 31)
        );
        if (dateDiff > 0) {
            var payableDate = DateTime.Parse(currentDividend["payableDt"]);
            var dividend = new Dividend() {
                Mic = MIC,
                Ticker = company.Ticker,
                CompanyName = company.CompanyName,
```

```

        ExDividendDate =
        FormatStringDate(currentDividend["exDividendDt"]),
        DeclarationDate =
        FormatStringDate(currentDividend["declarationDt"]),
        RecordDate = FormatStringDate(currentDividend["recordDt"]),
        PaymentDate =
        FormatStringDate(currentDividend["payableDt"]),
        Amount = float.Parse(currentDividend["amount"])
    };
    thisYearsDividends.Add(dividend);
}
}
return thisYearsDividends;
}

```

In this version of the API, we hardcode the MIC to "XLON"—the **London Stock Exchange**. However, in future releases, this method and the public endpoint could be updated to accept a MIC as a request parameter. We then add a list variable to hold this year's dividend payments. Then, we perform our Morningstar API call to extract the list of companies that are currently listed on the specified MIC. Once the list is returned, we loop through results. For each company, we then make a further API call to get the complete dividend record for the MIC and the ticker. If the company has no dividends listed, then we continue with the next iteration and select the next company.

If the company has dividend records, we get the first record, which will be the latest dividend payment. We check whether the payable date is `null`. If the payable date is `null`, then we continue on to the next iteration with the next customer. If the payable date is not `null`, we check whether the payable date is greater than December 31st from the previous year. If the date difference is greater than 1, then we add a new dividend object to this year's dividends list. Once we have iterated through all the companies and built up a list of this year's dividends, we then pass the list back to the calling method.

The final step before we run our project is to update the `GetDividendCalendar()` method to call the `BuildDividendCalendar()` method:

```

[Authorize(Policy = Policies.Internal)]
[HttpGet("internal")]
public IActionResult GetDividendCalendar()
{
    return new
    ObjectResult(JsonConvert.SerializeObject(BuildDividendCalendar()));
}

```

In the `GetDividendCalendar()` method, we return a JSON string from the serialized list of this year's dividends. So, if you run the project in Postman using the internal `x-api-key` variable, then you should find that after around 20 minutes, the following JSON is returned:

```
[{"Mic": "XLON", "Ticker": "ABDP", "CompanyName": "AB Dynamics  
PLC", "DividendYield": 0.0, "Amount": 0.0279, "ExDividendDate": "2020-01-02T00:00:  
00", "DeclarationDate": "2019-11-27T00:00:00", "RecordDate": "2020-01-03T00:00:  
00", "PaymentDate": "2020-02-13T00:00:00", "DividendType": null, "CurrencyCode":  
null},  
  
...  
  
{"Mic": "XLON", "Ticker": "ZYT", "CompanyName": "Zytronic  
PLC", "DividendYield": 0.0, "Amount": 0.152, "ExDividendDate": "2020-01-09T00:00:  
00", "DeclarationDate": "2019-12-10T00:00:00", "RecordDate": "2020-01-10T00:00:  
00", "PaymentDate": "2020-02-07T00:00:00", "DividendType": null, "CurrencyCode":  
null}]
```

This query does take a lot of time to run, roughly around 20 minutes, and the results will change over the course of a year. So, a strategy we could use is to throttle the API to run once a month and then store the JSON either in a file or a database. Then, this file or database record is what you would update the external method to call and pass back to external clients. Let's throttle our API to only run once a month.

Throttling our API

When exposing APIs, you need to throttle them. There are many methods available to do this, such as limiting the number of simultaneous users or limiting the number of calls within a given period of time, for example.

In this section, we are going to throttle our API. The method we will use to throttle our API will be to limit our API to run only once a month on the 25th of the month. Add the following line to your `appsettings.json` file:

```
"MorningstarNextRunDate": null,
```

This value will contain the date that the next API can be executed. Now, add the `AppSettings` class at the root of your project, and then add the following property:

```
public DateTime? MorningstarNextRunDate { get; set; }
```

This property will hold the value of the `MorningstarNextRunDate` key. The next thing to do is to add our static method, which will be called to add or update an application setting in the `appsetting.json` file:

```
public static void AddOrUpdateAppSetting<T>(string sectionPathKey, T value)
{
    try
    {
        var filePath = Path.Combine(AppContext.BaseDirectory,
            "appsettings.json");
        string json = File.ReadAllText(filePath);
        dynamic jsonObj =
            Newtonsoft.Json.JsonConvert.DeserializeObject(json);
        SetValueRecursively(sectionPathKey, jsonObj, value);
        string output = Newtonsoft.Json.JsonConvert.SerializeObject(
            jsonObj,
            Newtonsoft.Json.Formatting.Indented
        );
        File.WriteAllText(filePath, output);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error writing app settings | {0}", ex.Message);
    }
}
```

`AddOrUpdateAppSetting()` tries to get the file path for the `appsettings.json` file. It then reads the JSON from the file. The JSON is then deserialized into a dynamic object. We then call our method to recursively set the required value. Then, we write the JSON back to the same file. If an error is encountered, then we output an error message to the console. Let's write our `SetValueRecursively()` method:

```
private static void SetValueRecursively<T>(string sectionPathKey, dynamic
jsonObj, T value)
{
    var remainingSections = sectionPathKey.Split(":", 2);
    var currentSection = remainingSections[0];
    if (remainingSections.Length > 1)
    {
        var nextSection = remainingSections[1];
        SetValueRecursively(nextSection, jsonObj[currentSection], value);
    }
    else
    {
        jsonObj[currentSection] = value;
    }
}
```

The `SetValueRecursively()` method splits the string at the first apostrophe character. It then proceeds to recursively process the JSON, moving down the tree. When it gets to where it needs to be—that is, it finds the required value—the value is then set and the method returns. Add the `ThrottleMonthDay` constant to the `ApiKeyConstants` struct:

```
public const int ThrottleMonthDay = 25;
```

This constant is used for our day-of-the-month check when an API request is issued. In `DividendCalendarController`, add the `ThrottleMessage()` method:

```
private string ThrottleMessage()
{
    return "This API call can only be made once on the 25th of each month.";
}
```

The `ThrottleMessage()` method simply returns the message, "This API call can only be made once on the 25th of each month.". Now, add the following constructor:

```
public DividendCalendarController(IOptions<AppSettings> appSettings)
{
    _appSettings = appSettings.Value;
}
```

This constructor provides us with access to the values in the `appsettings.json` file. Add these two lines to the end of your `Startup.ConfigureServices()` method:

```
var appSettingsSection = Configuration.GetSection("AppSettings");
services.Configure<AppSettings>(appSettingsSection);
```

These two lines enable the `AppSettings` class to be dynamically injected into our controller when we need it. Add the `SetMorningstarNextRunDate()` method to the `DividendCalendarController` class:

```
private DateTime? SetMorningstarNextRunDate()
{
    int month;
    if (DateTime.Now.Day < 25)
        month = DateTime.Now.Month;
    else
        month = DateTime.Now.AddMonths(1).Month;
    var date = new DateTime(DateTime.Now.Year, month,
        ApiKeyConstants.ThrottleMonthDay);
    AppSettings.AddOrUpdateAppSetting<DateTime?>(
        "MorningstarNextRunDate",
        date
    );
}
```



```

    );
    return date;
}

```

The `SetMorningstarNextRunDate()` method checks whether the current month's day is less than 25. If the current month's day is less than 25, then the month is set to the current month so that the API can be run on the 25th of the current month. Otherwise, for days that are 25 and upward, the month is set to the following month. The new date is then assembled and the `MorningstarNextRunDate` key of `appsettings.json` is then updated and the nullable `DateTime` value is returned:

```

private bool CanExecuteApiRequest()
{
    DateTime? nextRunDate = _appSettings.MorningstarNextRunDate;
    if (!nextRunDate.HasValue)
        nextRunDate = SetMorningstarNextRunDate();
    if (DateTime.Now.Day == ApiKeyConstants.ThrottleMonthDay) {
        if (nextRunDate.Value.Month == DateTime.Now.Month) {
            SetMorningstarNextRunDate();
            return true;
        }
        else {
            return false;
        }
    }
    else {
        return false;
    }
}

```

`CanExecuteApiRequest()` gets the current value of the `MorningstarNextRunDate` value from the `AppSettings` class. If `DateTime?` does not have a value, then the value is set and assigned to the `nextRunDate` local variable. If the current month's day does not equal `ThrottleMonthDay`, then we return `false`. If the current month does not equal the next run date month, then we return `false`. Otherwise, we set the next API run date to the 25th of the following month and return `true`.

Finally, we update our `GetDividendCalendar()` method, as follows:

```

[Authorize(Policy = Policies.Internal)]
[HttpGet("internal")]
public IActionResult GetDividendCalendar()
{
    if (CanExecuteApiRequest())
        return new
            ObjectResult(JsonConvert.SerializeObject(BuildDividendCalendar()));
}

```

```
        else
            return new ObjectResult(ThrottleMessage());
    }
```

When an internal user calls the API now, their request will be validated to see whether it can run. If it runs, then the serialized JSON for the dividend calendar is returned. Otherwise, we return the `throttle` message.

That concludes our project.

Well, we've completed our project. It is not perfect, and there are improvements and extensions that we can make. The next step would be to document our API and deploy the API and documentation. We should also add logging and monitoring.

Logging is useful for storing exception details and for tracking how our API is used. Monitoring is a way to keep an eye on the health of our API so that we can be alerted if anything goes wrong. This way, we can be proactive in keeping our API up and running. I will leave you to extend the API as you desire. It will be a good learning exercise for you.



The next chapter addresses cross-cutting concerns. It will give you an idea about how to address logging and monitoring using aspects and attributes.

Let's summarise what we have learned.

Summary

In this chapter, you signed up to a third-party API and received your own key. The API key is stored in your Azure key vault and kept secure from access by unauthorized clients. You then moved on to create an ASP.NET Core web application and published it to Azure. Then, you set about securing the web application by using authentication and role-based authorization.

The authorization we set up is performed using an API key. You used two API keys in this project—one for internal use and one for external use. The testing of our API and API key security was performed using the Postman application. Postman is a very good and useful tool for testing HTTP requests and responses for the various HTTP verbs.

You then added the dividend calendar API code and enabled internal and external access based on API keys. The project itself performed a number of different API calls to build up a list of companies that are expecting to pay dividends to investors. The project then serializes the objects into JSON format, which is returned to the client. Finally, the project is throttled to run once a month.

So, by working through this chapter, you have created a FinTech API that you can run once a month. This API will provide dividend payment information for the current year. Your clients can deserialize this data and then perform LINQ queries on it to extract data that meets their specific requirements.

In the next chapter, we will be using PostSharp to implement **Aspect-Oriented Programming (AOP)**. With our AOP framework, we will learn how to manage common functionalities such as exception handling, logging, security, and transactions within our applications. But before that, let's put your brain to work to see what you have learned.

Questions

1. What URL is a good source for hosting your own APIs and accessing third-party APIs?
2. What are the two required parts for securing an API?
3. What are claims and why should you use them?
4. What do you use Postman for?
5. Why should you use the repository pattern for your data store?

Further reading

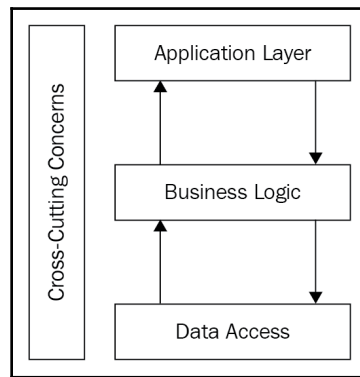
- <https://docs.microsoft.com/en-us/aspnet/web-api/overview/security/individual-accounts-in-web-api> is Microsoft's in-depth guide to web API security.
- <https://docs.microsoft.com/en-us/aspnet/web-forms/overview/older-versions-security/membership/creating-the-membership-schema-in-sql-server-vb> covers creating the ASP.NET membership database.
- <https://www.iso20022.org/10383/iso-10383-market-identifier-codes> is about ISO 10383 MIC.

- <https://docs.microsoft.com/en-gb/azure/key-vault/vs-key-vault-add-connected-service> covers adding key vault to your web application by using Visual Studio Connected Services.
- <https://aka.ms/installazurecliwindows> is about the Azure CLI MSI installer.
- <https://docs.microsoft.com/en-us/azure/key-vault/service-to-service-authentication> is the Azure service-to-service documentation.
- https://azure.microsoft.com/en-gb/free/?WT.mc_id=A261C142F is where you can sign up for your free 12-month subscription to Azure if you are a new customer.
- <https://docs.microsoft.com/en-us/azure/key-vault/basic-concepts> looks at the Azure Key Vault basic concepts.
- <https://docs.microsoft.com/en-us/azure/app-service/app-service-web-get-started-dotnet> covers creating a .NET Core app in Azure.
- <https://docs.microsoft.com/en-gb/azure/app-service/overview-hosting-plans> provides an Azure App Service plan overview.
- <https://docs.microsoft.com/en-us/azure/key-vault/tutorial-net-create-vault-azure-web-app> is a tutorial on using Azure Key Vault with an Azure web app in .NET.

11

Addressing Cross-Cutting Concerns

There are two types of concerns that you need to have when writing clean code—core concerns and cross-cutting concerns. **Core concerns** are the reasons for the software and why it is being developed. **Cross-cutting concerns** are the concerns that are not part of the business requirements and that form the core concerns, but must be addressed in all areas of the code, as illustrated in the following diagram:



It is the cross-cutting concerns that we will be covering in this chapter by building a reusable class library that you can modify or extend to your liking. Cross-cutting concerns include configuration management, logging, auditing, security, validation, exception-handling, instrumentation, transactions, resource pooling, caching, and threading and concurrency. We will use the decorator pattern and the PostSharp Aspect Framework to help us build our reusable library, which is injected at compile time.

As you read through this chapter, you will see how **attribute programming** can result in using a lot less boilerplate code, as well as code that is smaller, more readable, and easier to maintain and extend. This leaves only the required business code in your methods with the boilerplate code



We have discussed many of these ideas already. However, they are mentioned here again as they are cross-cutting concerns.

In this chapter, we'll be covering the following topics:

- The decorator pattern
- The proxy pattern
- **Aspect-Oriented Programming (AOP)** with PostSharp
- Project – cross-cutting concerns reusable library

By the end of this chapter, you will have the skills to do the following:

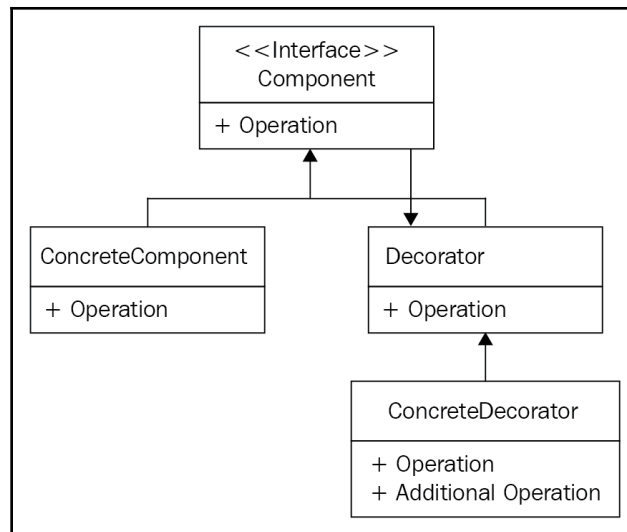
- Implement the decorator pattern.
- Implement the proxy pattern.
- Apply AOP using PostSharp.
- Build your own reusable AOP library that addresses your cross-cutting concerns.

Technical requirements

To get the most out of this chapter, you will need Visual Studio 2019 and PostSharp installed. For the code files for this chapter, refer to <https://github.com/PacktPublishing/Clean-Code-in-C-/tree/master/CH11>. Let's start by looking at the decorator pattern.

The decorator pattern

The decorator design pattern is a structural pattern that is used to add new functionality to an existing object without changing its structure. The original class is wrapped in decorator class wraps and new behaviors and operations are added to an object at runtime:



The **Component** interface and the members it contains are implemented by the **ConcreteComponent** class and the **Decorator** class. **ConcreteComponent** implements the **Component** interface. The **Decorator** class is an abstract class that implements the **Component** interface and contains the reference to a **Component** instance. The **Decorator** class is the base class for components. The **ConcreteDecorator** class inherits from the **Decorator** class and provides a decorator for components.

We are going to write an example that wraps an operation in a `try/catch` block. Both `try` and `catch` will output a string to the console. Create a new .NET 4.8 console application named `CH10_AddressCrossCuttingConcerns`. Then, add a folder called `DecoratorPattern`. Add a new interface called `IComponent`:

```
public interface IComponent {
    void Operation();
}
```

To keep things simple, our interface only has a single operation of the `void` type. Now that we have our interface in place, we need to add an abstract class that implements the interface. Add a new abstract class called **Decorator** that implements the **IComponent** interface. Add a member variable to store our **IComponent** object:

```
private IComponent _component;
```

The `_component` member variable, which stores the `IComponent` object, is set via the constructor, as follows:

```
public Decorator(IComponent component) {  
    _component = component;  
}
```

In the preceding code, the constructor sets the component we will be decorating. Next, we add our interface method:

```
public virtual void Operation() {  
    _component.Operation();  
}
```

We've declared the `Operation()` method as `virtual` so that it can be overridden in the derived classes. We'll now create our `ConcreteComponent` class, which implements `IComponent`:

```
public class ConcreteComponent : IComponent {  
    public void Operation() {  
        throw new NotImplementedException();  
    }  
}
```

As you can see, our class consists of one operation, which throws `NotImplementedException`. Now, we can write about the `ConcreteDecorator` class:

```
public class ConcreteDecorator : Decorator {  
    public ConcreteDecorator(IComponent component) : base(component) { }  
}
```

The `ConcreteDecorator` class inherits the `Decorator` class. The constructor takes an `IComponent` parameter and passes it to the base constructor, where the member variable is then set. Next, we'll override the `Operation()` method:

```
public override void Operation() {  
    try {  
        Console.WriteLine("Operation: try block.");  
        base.Operation();  
    } catch (Exception ex) {  
        Console.WriteLine("Operation: catch block.");  
        Console.WriteLine(ex.Message);  
    }  
}
```

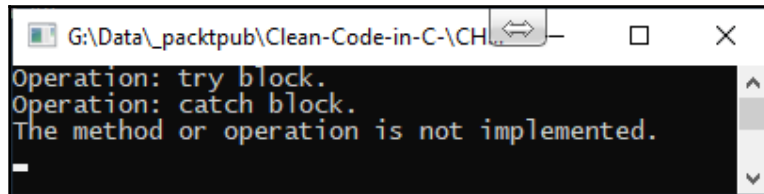

In our overridden method, we have a `try/catch` block. In the `try` block, we write a message to the console and execute the base class' `Operation()` method. In the `catch` block, when an exception is encountered, a message is written, followed by the error message. Before we can use our code, we need to update the `Program` class. Add the `DecoratorPatternExample()` method to the `Program` class:

```
private static void DecoratorPatternExample() {  
    var concreteComponent = new ConcreteComponent();  
    var concreteDecorator = new ConcreteDecorator(concreteComponent);  
    concreteDecorator.Operation();  
}
```

In our `DecoratorPatternExample()` method, we create a new concrete component. We then pass it into the constructor of a new concrete decorator. Then, we call the `Operation()` method on the concrete decorator. Add the following two lines to the `Main()` method:

```
DecoratorPatternExample();  
Console.ReadKey();
```

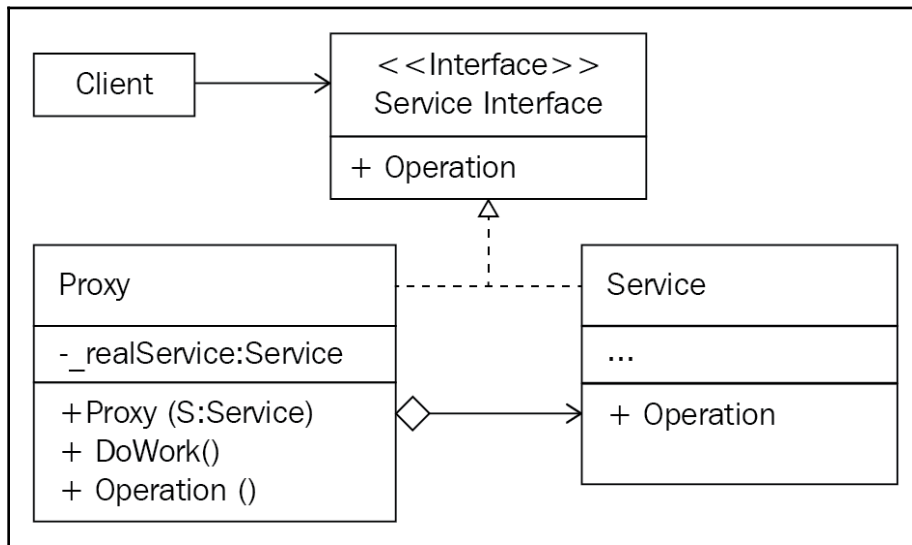
Those two lines execute our example and then wait for the user to press a key before exiting. Run the code and you should see the same output as in the following screenshot:



That concludes our look at the decorator pattern. Now, it's time to look at the proxy pattern.

The proxy pattern

The proxy pattern is a structural design pattern providing objects that act as substitutes for real service objects used by clients. Proxies receive client requests, perform the required work, and then pass the request to service objects. Proxy objects are interchangeable with services as they share the same interfaces:



An example of when you would want to use the proxy pattern is when you have a class that you do not want to change, but where you do need additional behaviors to be added. Proxies delegate work to other objects. Unless a proxy is a derivative of a service, proxy methods should finally refer to a `Service` object.

We will look at a very simple implementation of the proxy pattern. Add a folder to the root of your `Chapter 11` project called `ProxyPattern`. Add an interface called `IService` with a single method to handle a request:

```
public interface IService {
    void Request();
}
```

The `Request()` method performs the work that carries out the request. Both the proxy and the service will implement this interface to use the `Request()` method. Now, add the `Service` class and implement the `IService` interface:

```
public class Service : IService {
    public void Request() {
        Console.WriteLine("Service: Request()");
    }
}
```

Our `Service` class implements the `IService` interface and handles the actual service `Request()` method. This `Request()` method will be called by the `Proxy` class. The final step to implementing the proxy pattern is to write the `Proxy` class:

```
public class Proxy : IService {
    private IService _service;

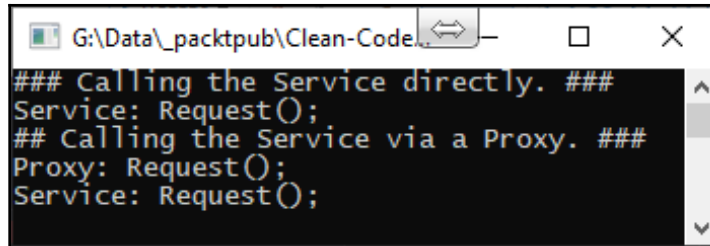
    public Proxy(IService service) {
        _service = service;
    }

    public void Request() {
        Console.WriteLine("Proxy: Request()");
        _service.Request();
    }
}
```

Our `Proxy` class implements `IService` and has a constructor that accepts a single `IService` parameter. The `Request()` method of the `Proxy` class is called by the client. The `Proxy.Request()` method will do what it needs to do and will be responsible for calling `_service.Request()`. So that we can see this in action, let's update our `Program` class. Add the `ProxyPatternExample()` call to the `Main()` method. Then, add the `ProxyPatternExample()` method:

```
private static void ProxyPatternExample() {
    Console.WriteLine("### Calling the Service directly. ###");
    var service = new Service();
    service.Request();
    Console.WriteLine("## Calling the Service via a Proxy. ###");
    new Proxy(service).Request();
}
```

Our test method runs the `Request()` method of the `Service` class direction. Then, it runs the same method via the `Request()` method of the `Proxy` class. Run the project and you should see the following:



```
### Calling the Service directly. ###
Service: Request();
## Calling the Service via a Proxy. ##
Proxy: Request();
Service: Request();
```

Now that you have a working understanding of the decorator and proxy patterns, let's take a look at AOP with PostSharp.

AOP with PostSharp

AOP can be used with OOP. An **aspect** is an attribute applied to classes, methods, parameters, and properties that, at compile-time, weaves code into the class, method, parameter, or property to which it is applied. This approach allows the cross-cutting concerns of a program to be moved from the business source code to a class library. The concerns are added where needed as attributes. The compiler then weaves the required code in at runtime. This keeps your business code small and readable. In this chapter, we will be using PostSharp. You can download it from <https://www.postsharp.net/download>.

So, how does AOP work with PostSharp?

You add the PostSharp package to your project. Then, you annotate your code with attributes. The C# compiler builds your code into binary, and then PostSharp analyzes the binary and injects the implementation of the aspects. Although the binaries are modified with injected code at compile-time, your project's source code remains unaltered. This means you can keep your code nice, clean, and simple, which in turn makes maintenance, reuse, and extending existing code bases much easier in the long term.

PostSharp has some really good ready-made patterns for you to utilize. These cover **Model-View-ViewModel (MVVM)**, caching, multi-threading, logging and architecture validation, and more. But the good news is that if there is nothing that meets your requirements, then you can automate your own patterns by extending the aspect framework and/or the architecture framework.

With the aspect framework, you develop your simple or composite aspect, apply it to the code, and validate its usage. As for the architectural framework, you develop your custom architectural constraints. Before we delve into the cross-cutting concerns, let's briefly take a look at extending the aspect and architectural frameworks.



You need to add the `PostSharp.Redist` NuGet package when writing aspects and attributes. Once done, if you find that your attributes and aspects are not working, then right-click on the project and select **Add PostSharp to Project**. After you've done this, your aspects should work.

Extending the aspect framework

In this section, we are going to develop a simple aspect and apply it to some code. Then, we will validate the usage of our aspect.

Developing our aspect

Our aspect will be a simple one that is composed of a single transformation. We will derive our aspect from a primitive aspect class. Then, we will override some methods known as **advice**. If you would like to know how to create a composite aspect, you can read how to do so at <https://doc.postsharp.net/complex-aspects>.

Injecting behaviors before and after the method execution

The `OnMethodBoundaryAspect` aspect implements the decorator pattern. You have already seen how to implement the decorator pattern earlier in this chapter. With this aspect, you can execute logic before and after the execution of a target method. The following table provides a list of the advice methods that are available in the `OnMethodBoundaryAspect` class:

Advice	Description
<code>OnEntry (MethodExecutionArgs)</code>	Used when the method's execution starts, before any user code.
<code>OnSuccess (MethodExecutionArgs)</code>	Used when the method's execution succeeds (that is, returns without an exception), after any user code.
<code>OnException (MethodExecutionArgs)</code>	Used when the method execution fails with an exception, after any user code. It is equivalent to a <code>catch</code> block.

OnExit (MethodExecutionArgs)	Used when the method execution exits, whether successfully or with an exception. This advice runs after any user code and after the OnSuccess (MethodExecutionArgs) or OnException (MethodExecutionArgs) method of the current aspect. It is equivalent to a finally block.
------------------------------	---

For our simple aspect, we are going to look at all the methods in use. Before we begin, add PostSharp to your project. If you have already downloaded PostSharp, you can right-click on your project and then select **Add PostSharp to Project**. After that, add a new folder to your project called Aspects, and then add a new class called LoggingAspect:

```
[PSerializable]
public class LoggingAspect : OnMethodBoundaryAspect { }
```

The [PSerializable] attribute is a custom attribute that, when applied to a type, causes PostSharp to generate a serializer for use by PortableFormatter. Now, override the OnEntry() method:

```
public override void OnEntry(MethodExecutionArgs args) {
    Console.WriteLine("The {0} method has been entered.",
        args.Method.Name);
}
```

The OnEntry() method is executed before any user code. Now, override the OnSuccess() method:

```
public override void OnSuccess(MethodExecutionArgs args) {
    Console.WriteLine("The {0} method executed successfully.",
        args.Method.Name);
}
```

The OnSuccess() method runs after the user code has completed without exception. Override the OnExit() method:

```
public override void OnExit(MethodExecutionArgs args) {
    Console.WriteLine("The {0} method has exited.", args.Method.Name);
}
```

The `OnExit()` method executes when the user method completes successfully or unsuccessfully and exits. It is equivalent to a `finally` block. Finally, override the `OnException()` method:

```
public override void OnException(MethodExecutionArgs args) {  
    Console.WriteLine("An exception was thrown in {0}.", args.Method.Name);  
}
```

The `OnException()` method executes when method execution fails with an exception, after any user code. It is equivalent to a `catch` block.

The next step is to write two methods that we can apply `LoggingAspect` to. We'll add `SuccessfulMethod()`:

```
[LoggingAspect]  
private static void SuccessfulMethod() {  
    Console.WriteLine("Hello World, I am a success!");  
}
```

`SuccessfulMethod()` uses `LoggingAspect` and prints a message to the console. Now, let's add `FailedMethod()`:

```
[LoggingAspect]  
private static void FailedMethod() {  
    Console.WriteLine("Hello World, I am a failure!");  
    var x = 1;  
    var y = 0;  
    var z = x / y;  
}
```

`FailedMethod()` uses `LoggingAspect` and prints a message to the console. Then, it performs a division by zero operations, which results in `DivideByZeroException`. Call both of these methods from your `Main()` method, and then run through your project. You should see the following output:

At this point, the debugger will cause the program to exit. That's it. As you can see, creating your own PostSharp aspects to meet your needs is a simple process. Now, we will look at adding our own architectural constraint.

Extending the architectural framework

An architectural constraint is the adoption of custom design patterns that must be respected across all modules. We will implement a scalar constraint that validates an element of code.

Our scalar constraint, called `BusinessRulePatternValidation`, will validate that any class deriving from the `BusinessRule` class must have a nested class named `Factory`. Start by adding the `BusinessRulePatternValidation` class:

```
[MulticastAttributeUsage(MulticastTargets.Class, Inheritance =
MulticastInheritance.Strict)]
public class BusinessRulePatternValidation : ScalarConstraint { }
```

`MulticastAttributeUsage` designates that this validation aspect will only work with classes and inheritance allowed. Let's override the `ValidateCode()` method:

```
public override void CodeValidation(object target) {
    var targetType = (Type)target;
    if (targetType.GetNestedType("Factory") == null) {
        Message.Write(
            targetType, SeverityType.Warning,
            "10",
            "You must include a 'Factory' as a nested type for {0}.",
            targetType.DeclaringType,
            targetType.Name);
    }
}
```

Our `ValidateCode()` method checks whether the target object has a nested `Factory` type. If the `Factory` type is not present, then an exception message is written to the output window. Add the `BusinessRule` class:

```
[BusinessRulePatternValidation]
public class BusinessRule { }
```


The `BusinessRule` class is empty and devoid of `Factory`. It has our `BusinessRulePatternValidation` attribute assigned to it, which is an architectural constraint. Build your project and you will see the message in the output window. We will now start to build a reusable class library that you can extend and use in your own projects to address cross-cutting concerns using AOP and the decorator pattern.

Project – cross-cutting concerns reusable library

In this section, we will be working through writing a reusable library for addressing various cross-cutting concerns. It will have limited functionality, but it will give you the knowledge you need to further expand the project for your own needs. The class library you will be creating will be a .NET standard library so that it can be used for apps that target both .NET Framework and .NET Core. You will also create a .NET Framework console application to see the library in action.

Start by creating a new .NET standard class library called `CrossCuttingConcerns`. Then, add a .NET Framework console application to the solution called `TestHarness`. We will be adding reusable functionality to address various concerns, starting with caching.

Adding the caching concern

Caching is a storage technique for improving performance when accessing various kinds of resources. The cache used can be memory, a filesystem, or a database. The type of cache you use will be dependent on the needs of the project. For our demonstration, we will be using memory caching to keep things simple.

Add a folder called `Caching` to the `CrossCuttingConcerns` project. Then, add a class called `MemoryCache`. Add the following NuGet packages to the project:

- `PostSharp`
- `PostSharp.Patterns.Common`
- `PostSharp.Patterns.Diagnostics`
- `System.Runtime.Caching`

Update the `MemoryCache` class with the following code:

```
public static class MemoryCache {
    public static T GetItem<T>(string itemName, TimeSpan timeInCache,
Func<T> itemCacheFunction) {
        var cache = System.Runtime.Caching.MemoryCache.Default;
        var cachedItem = (T) cache[itemName];
        if (cachedItem != null) return cachedItem;
        var policy = new CacheItemPolicy {AbsoluteExpiration =
DateTimeOffset.Now.Add(timeInCache)};
        cachedItem = itemCacheFunction();
        cache.Set(itemName, cachedItem, policy);
        return cachedItem;
    }
}
```

The `GetItem()` method takes the name of the cached item, `itemName`, the length of time the item is to remain in the cache, `timeInCache`, and the function to call to place the item in the cache if it is not already there, `itemCacheFunction`. Add a new class to the `TestHarness` project and call it `TestClass`. Then, add the `GetCachedItem()` and `GetMessage()` methods, as shown:

```
public string GetCachedItem() {
    return MemoryCache.GetItem<string>("Message", TimeSpan.FromSeconds(30),
GetMessage);
}

private string GetMessage() {
    return "Hello, world of cache!";
}
```

The `GetCachedItem()` method gets a string called "Message" from the cache. If it is not in the cache, then it will be stored in the cache by the `GetMessage()` method for 30 seconds.

Update your `Main()` method in the `Program` class to call the `GetCachedItem()` method, as shown:

```
var harness = new TestClass();
Console.WriteLine(harness.GetCachedItem());
Console.WriteLine(harness.GetCachedItem());
Thread.Sleep(TimeSpan.FromSeconds(1));
Console.WriteLine(harness.GetCachedItem());
```

The first call to `GetCachedItem()` stores the item in the cache and then returns it. The second call obtains the item from the cache and returns it. The sleeping thread invalidates the cache, and so the last call stores the item in the cache before returning it.

Adding file logging capabilities

In our project, the logging, auditing, and instrumentation processes will send their output to a text file. So, we will need a class to manage adding the files if they don't exist, and then adding the output to those files and saving them. Add a folder to the class library called `FileSystem`. Then, add a class called `LogFile`. Set the class as `public static` and add the following member variables:

```
private static string _location = string.Empty;
private static string _filename = string.Empty;
private static string _file = string.Empty;
```

The `_location` variable is assigned the folder for the entry assembly. The `_filename` variable is assigned the name of the file with the file extension. We need to add the `Logs` folder at runtime (if it does not exist). So, we will add the `AddDirectory()` method to the `FileSystem` class:

```
private static void AddDirectory() {
    if (!Directory.Exists(_location))
        Directory.CreateDirectory("Logs");
}
```

The `AddDirectory()` method checks whether the location exists. If it does not exist, then the directory is created. Next, we need to deal with adding the file if it does not exist. So, add the `AddFile()` method:

```
private static void AddFile() {
    _file = Path.Combine(_location, _filename);
    if (File.Exists(_file)) return;
    using (File.Create($"Logs\\{_filename}")) {
    }
}
```

In the `AddFile()` method, we combine the location and filename. If the filename already exists, then we exit the method; otherwise, we create the file. If we don't use the `using` statement, we will encounter `IOException` when we create our first record, but subsequent saves will be fine. So, by using the `using` statement, we avoid the exception and log the data. We can now write a method that actually saves the data to a file. Add the `AppendTextToFile()` method:

```
public static void AppendTextToFile(string filename, string text) {
    _location =
        $"{Path.GetDirectoryName(Assembly.GetEntryAssembly()?.Location)}\\Logs";
    _filename = filename;
```

```
        AddDirectory();  
        AddFile();  
        File.AppendAllText(_file, text);  
    }
```

The `AppendTextToFile()` method takes a filename and text and sets the location to that of the entry assembly. It then ensures that the file and directory exist. Then, it saves the text to the specified file. Our file logging capabilities are now taken care of, so now, we can move on to look at our logging concern.

Adding the logging concern

Most applications need some form of logging. The usual methods of logging are to the console, filesystem, event logs, and database. In our project, we will only focus on console and text file logging. Add a folder called `Logging` to the class library. Then, add a file called `ConsoleLoggingAspect` and update it as follows:

```
[Serializable]  
public class ConsoleLoggingAspect : OnMethodBoundaryAspect { }
```

The `[Serializable]` attribute informs PostSharp to generate a serializer for use by `PortableFormatter`. `ConsoleLoggingAspect` inherits from `OnMethodBoundaryAspect`. The `OnMethodBoundaryAspect` class has methods that we can override to add code before a method body executes, after a method body executes, when a method body executes successfully, and when an exception is encountered. We will override these methods to output a message to the console. This can be a very useful tool when it comes to debugging to see whether code actually gets called and whether it successfully completes or encounters an exception. We will start by overriding the `OnEntry()` method:

```
public override void OnEntry(MethodExecutionArgs args) {  
    Console.WriteLine($"Method: {args.Method.Name}, OnEntry().");  
}
```

The `OnEntry()` method executes before the body of our methods do, and our override prints out the name of the method been executed and its own name. Next, we'll override the `OnExit()` method:

```
public override void OnExit(MethodExecutionArgs args) {  
    Console.WriteLine($"Method: {args.Method.Name}, OnExit().");  
}
```

The `OnExit()` method executes after the body of our methods have finished executing, and our override prints out the name of the method that has been executed and its own name. Now, we'll add the `OnSuccess()` method:

```
public override void OnSuccess(MethodExecutionArgs args) {  
    Console.WriteLine($"Method: {args.Method.Name}, OnSuccess().");  
}
```

The `OnSuccess()` method executes after the body of the method it is applied to has finished and returns without exception. When our override executes, it prints out the name of the executed method and its own name. The last method we will override is the `OnException()` method:

```
public override void OnException(MethodExecutionArgs args) {  
    Console.WriteLine($"An exception was thrown in {args.Method.Name}.  
{args}");  
}
```

The `OnException()` method executes when an exception is encountered, and in our override, we print out the name of the method and the argument's object. To apply the attribute, use `[ConsoleLoggingAspect]`. To add a text file logging aspect, add a class called `TextFileLoggingAspect`. `TextFileLoggingAspect` is identical to `ConsoleLoggingAspect`, apart from the contents of the overridden methods. The `OnEntry()`, `OnExit()`, and `OnSuccess()` methods call the `LogFile.AppendTextToFile()` method and append the contents to the `Log.txt` file. The `OnException()` method does the same, except it appends the contents to the `Exception.log` file. Here is the `OnEntry()` example:

```
public override void OnEntry(MethodExecutionArgs args) {  
    LogFile.AppendTextToFile("Log.txt", $"\\nMethod: {args.Method.Name},  
OnEntry().");  
}
```

That is our logging taken care of. Now, we'll move on to adding our exceptions concern.

Adding the exception-handling concern

It is inevitable with software that exceptions will be experienced by users of the software. So, there needs to be some way to log them. The normal way of logging exceptions is to store the error in a file on the user's system, such as with `Exception.log`. That's what we'll do in this section. We will inherit from the `OnExceptionAspect` class and write our exception data to the `Exception.log` file, which will be located in the `Logs` folder of our application. `OnExceptionAspect` wraps the tagged method in a `try/catch` block. Add a new folder to the class library called `Exceptions`, and then add a file called `ExceptionAspect` with the following code:

```
[Serializable]
public class ExceptionAspect : OnExceptionAspect {
    public string Message { get; set; }
    public Type ExceptionType { get; set; }
    public FlowBehavior Behavior { get; set; }

    public override void OnException(MethodExecutionArgs args) {
        var message = args.Exception != null ? args.Exception.Message :
"Unknown error occurred.";
        LogFile.AppendTextToFile(
            "Exceptions.log", $"{DateTime.Now}: Method: {args.Method},
Exception: {message}"
        );
        args.FlowBehavior = FlowBehavior.Continue;
    }

    public override Type GetExceptionType(System.Reflection.MethodBase
targetMethod) {
        return ExceptionType;
    }
}
```

The `ExceptionAspect` class is assigned the `[Serializable]` aspect and inherits from `OnExceptionAspect`. We have three properties: `message`, `ExceptionType`, and `FlowBehavior`. `message` contains the exception message, `ExceptionType` contains the type of exception encountered, and `FlowBehavior` determines whether execution continues once the exception is handled or whether the process terminates. The `GetExceptionType()` method returns the type of exception that was thrown. The `OnException()` method starts by constructing the error message. It then logs the exception to file by calling `LogFile.AppendTextToFile()`. Finally, the flow of the exception's behavior is set to continue.

All you have to do to use the `[ExceptionAspect]` aspect is add it as an attribute to your method. We have now covered exception-handling. So, we'll move on to adding our security concern.

Adding the security concern

The security needs will be specific to the project being worked on. The most common concerns are that users are authenticated and authorized to access and use various parts of the system. In this section, we will use the decorator pattern to implement a secure component with role-based methods.



Security is a very large subject in itself and beyond the scope of this book. There are many good APIs out there, such as the various Microsoft APIs. Refer to <https://docs.microsoft.com/en-us/dotnet/standard/security/> for more information, and for OAuth 2.0, refer to <https://oauth.net/code/dotnet/>. We will leave you to select and implement your own method of security. In this chapter, we simply add our own custom-defined security using the decorator pattern. You can use this as a base for implementing any of the aforementioned security methods.

Add a new folder called `Security` and add an interface to it called `ISecureComponent`:

```
public interface ISecureComponent {
    void AddData(dynamic data);
    int EditData(dynamic data);
    int DeleteData(dynamic data);
    dynamic GetData(dynamic data);
}
```

Our secure component interface contains the preceding four methods, which are self-explanatory. The `dynamic` keyword means that any type of data can be passed in as a parameter and that any type of data can be returned from the `GetData()` method. Next, we need an abstract class that implements the interface. Add a class called `DecoratorBase`, as shown:

```
public abstract class DecoratorBase : ISecureComponent {
    private readonly ISecureComponent _secureComponent;

    public DecoratorBase(ISecureComponent secureComponent) {
        _secureComponent = secureComponent;
    }
}
```

The `DecoratorBase` class implements `ISecureComponent`. We declare a member variable of the `ISecureComponent` type and set it in the default constructor. We need to add the missing methods of `ISecureComponent`. Add the `AddData()` method:

```
public virtual void AddData(dynamic data) {
    _secureComponent.AddData(data);
}
```

This method will take any type of data and then pass it into the call to the `AddData()` method of `_secureComponent`. Add the missing methods for `EditData()`, `DeleteData()`, and `GetData()`. Now, add a class called `ConcreteSecureComponent`, which implements `ISecureComponent`. For each method, write a message to the console. For the `DeleteData()` and `EditData()` methods, also return a value of 1. Return "Hi!" for `GetData()`. The `ConcreteSecureComponent` class is the class that executes the secure work that we are interested in.

We need a way to validate the user and obtain their role. The role will be checked before executing any methods. So, add the following struct:

```
public readonly struct Credentials {
    public static string Role { get; private set; }

    public Credentials(string username, string password) {
        switch (username)
        {
            case "System" when password == "Administrator":
                Role = "Administrator";
                break;
            case "End" when password == "User":
                Role = "Restricted";
                break;
            default:
                Role = "Imposter";
                break;
        }
    }
}
```

To keep things simple, the struct takes a username and password and sets the appropriate role. Restricted users have fewer privileges than administrators. The final class for our security concern is the `ConcreteDecorator` class. Add the class, as follows:

```
public class ConcreteDecorator : DecoratorBase {
    public ConcreteDecorator(ISecureComponent secureComponent) :
        base(secureComponent) { }
}
```


The `ConcreteDecorator` class inherits the `DecoratorBase` class. Our constructor takes a type of `ISecureComponent` and passes it to the base class. Add the `AddData()` method:

```
public override void AddData(dynamic data) {
    if (Credentials.Role.Contains("Administrator") ||
        Credentials.Role.Contains("Restricted")) {
        base.AddData((object) data);
    } else {
        throw new UnauthorizedAccessException("Unauthorized");
    }
}
```

`AddMethod()` checks the user's role against the allowed `Administrator` and `Restricted` roles. If the user is in one of these roles, then the `AddData()` method is executed in the base class; otherwise, `UnauthorizedAccessException` is thrown. The rest of the methods follow this same pattern. Override the rest of the methods, but make sure the `DeleteData()` method can only be executed by administrators.

We will now put our security concerns to work. Add the following line to the top of the `Program` class:

```
private static readonly ConcreteDecorator ConcreteDecorator = new
    ConcreteDecorator(
        new ConcreteSecureComponent()
    );
```

We are declaring and instantiating a concrete decorator object and passing in the concrete secure object. This object will be referenced in our data methods. Update the `Main()` method, as follows:

```
private static void Main(string[] _) {
    // ReSharper disable once ObjectCreationAsStatement
    new Credentials("End", "User");
    DoSecureWork();
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
```

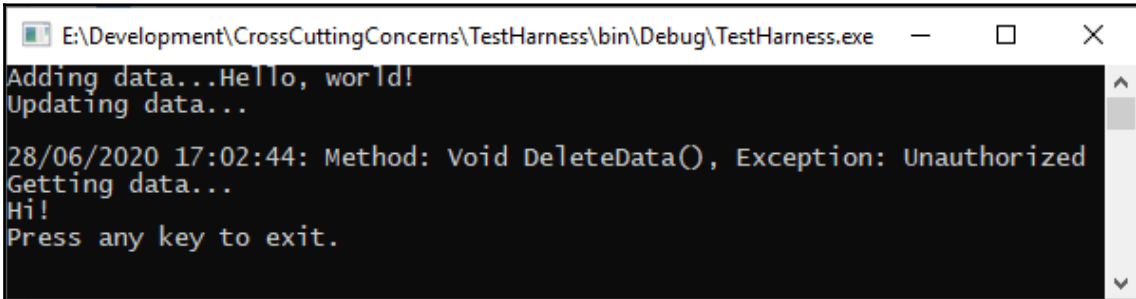
We assign the username and password to the `Credentials` struct. This causes `Role` to be set. We then call the `DoWork()` method. The `DoWork()` method will be responsible for calling the data methods. We then pause for the user to press any key and exit. Add the `DoWork()` method:

```
private static void DoSecureWork() {  
    AddData();  
    EditData();  
    DeleteData();  
    GetData();  
}
```

The `DoSecureWork()` method calls each of the data methods that call the data methods on the concrete decorator. Add the `AddData()` method:

```
[ExceptionHandler(consoleOutput: true)]  
private static void AddData() {  
    ConcreteDecorator.AddData("Hello, world!");  
}
```

`[ExceptionHandler]` is applied to the `AddData()` method. This will ensure any errors are logged to the `Exceptions.log` file. The parameter is set to `true`, and so the error message will also be printed in the console window. The method itself calls the `AddData()` method on the `ConcreteDecorator` class. Add the rest of the methods by following the same procedure. Then, run your code. You should see the following output:



```
E:\Development\CrossCuttingConcerns\TestHarness\bin\Debug\TestHarness.exe  
Adding data...Hello, world!  
Updating data...  
28/06/2020 17:02:44: Method: Void DeleteData(), Exception: Unauthorized  
Getting data...  
Hi!  
Press any key to exit.
```

We now have a working role-based object, complete with exception handling. Our next step is to implement our validation concern.

Adding the validation concern

All user-entered data should be validated as it could be malicious, incomplete, or in the wrong format. You need to ensure that your data is clean and cannot cause harm. For our demonstration concern, we will implement null validation. Start by adding a folder called `Validation` to the class library. Then, add a new class called

`AllowNullAttribute`:

```
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.ReturnValue |
AttributeTargets.Property)]
public class AllowNullAttribute : Attribute { }
```

This attribute allows nulls on parameters, return values, and properties. Now, add the `ValidationFlags` enum to a new file of the same name:

```
[Flags]
public enum ValidationFlags {
    Properties = 1,
    Methods = 2,
    Arguments = 4,
    OutValues = 8,
    ReturnValues = 16,
    NonPublic = 32,
    AllPublicArguments = Properties | Methods | Arguments,
    AllPublic = AllPublicArguments | OutValues | ReturnValues,
    All = AllPublic | NonPublic
}
```

These flags are used to determine what items an aspect can be applied to. Next, we'll add a class called `ReflectionExtensions`:

```
public static class ReflectionExtensions {
    private static bool IsCustomAttributeDefined<T>(this
ICustomAttributeProvider value) where T
        : Attribute {
        return value.IsDefined(typeof(T), false);
    }

    public static bool AllowsNull(this ICustomAttributeProvider value) {
        return value.IsCustomAttributeDefined<AllowNullAttribute>();
    }

    public static bool MayNotBeNull(this ParameterInfo arg) {
        return !arg.AllowsNull() && !arg.IsOptional &&
!arg.ParameterType.IsValueType;
    }
}
```

The `IsCustomAttributeDefined()` method returns `true` if the attribute type is defined on this member, and `false` otherwise. The `AllowsNull()` method returns `true` if the `[AllowNull]` attribute is already applied, and `false` if not. The `MayNotBeNull()` method checks to see whether nulls are allowed, whether the parameter is optional, and what type of value the parameter is. A Boolean value is then returned by performing logical AND operations on these values. It's time to add `DisallowNonNullAspect`:

```
[Serializable]
public class DisallowNonNullAspect : OnMethodBoundaryAspect {
    private int[] _inputArgumentsToValidate;
    private int[] _outputArgumentsToValidate;
    private string[] _parameterNames;
    private bool _validateReturnValue;
    private string _memberName;
    private bool _isProperty;

    public DisallowNonNullAspect() : this(ValidationFlags.AllPublic) { }

    public DisallowNonNullAspect(ValidationFlags validationFlags) {
        ValidationFlags = validationFlags;
    }

    public ValidationFlags ValidationFlags { get; set; }
}
```

This class has the `[Serializable]` attribute applied to inform PostSharp to generate a serializer for `PortableFormatter`. It also inherits the `OnMethodBoundaryAspect` class. We then declare variables to hold the input and output arguments as validated parameter names, return value validation and the member name, and check whether the item being validated is a property. The default constructor is configured to allow the validator to be applied to all public members. We also have a constructor that takes a `ValidationFlags` value and a `ValidationFlags` property. Now, we'll override the `CompileTimeValidate()` method:

```
public override bool CompileTimeValidate(MethodBase method) {
    var methodInformation = MethodInformation.GetMethodInformation(method);
    var parameters = method.GetParameters();

    if (!ValidationFlags.HasFlag(ValidationFlags.NonPublic) &&
        !methodInformation.IsPublic) return false;
    if (!ValidationFlags.HasFlag(ValidationFlags.Properties) &&
        methodInformation.IsProperty)
        return false;
    if (!ValidationFlags.HasFlag(ValidationFlags.Methods) &&
        !methodInformation.IsProperty) return false;
```

```

        _parameterNames = parameters.Select(p => p.Name).ToArray();
        _memberName = methodInformation.Name;
        _isProperty = methodInformation.IsProperty;

        var argumentsToValidate = parameters.Where(p =>
p.MayNotBeNull()).ToArray();

        _inputArgumentsToValidate =
ValidationFlags.HasFlag(ValidationFlags.Arguments) ?
argumentsToValidate.Where(p => !p.IsOut).Select(p => p.Position).ToArray()
: new int[0];

        _outputArgumentsToValidate =
ValidationFlags.HasFlag(ValidationFlags.OutValues) ?
argumentsToValidate.Where(p => p.ParameterType.IsByRef).Select(p =>
p.Position).ToArray() : new int[0];

        if (!methodInformation.IsConstructor) {
            _validateReturnValue =
ValidationFlags.HasFlag(ValidationFlags.ReturnValues) &&
methodInformation.ReturnParameter.MayNotBeNull();
        }

        var validationRequired = _validateReturnValue ||
_inputArgumentsToValidate.Length > 0 || _outputArgumentsToValidate.Length >
0;

        return validationRequired;
    }

```

This method ensures that the aspect is correctly applied at compile-time. If the aspect is applied to a wrong type of member, then `false` is returned. Otherwise, it returns `true`. We now override the `OnEntry()` method:

```

public override void OnEntry(MethodExecutionArgs args) {
    foreach (var argumentPosition in _inputArgumentsToValidate) {
        if (args.Arguments[argumentPosition] != null) continue;
        var parameterName = _parameterNames[argumentPosition];

        if (_isProperty) {
            throw new ArgumentNullException(parameterName,
                $"Cannot set the value of property '{_memberName}' to
null.");
        } else {
            throw new ArgumentNullException(parameterName);
        }
    }
}

```

This method checks the *input arguments* to validate. If any arguments are null, then `ArgumentNullException` is thrown; otherwise, the method exits without throwing an exception. Let's override the `OnSuccess()` method now:

```
public override void OnSuccess(MethodExecutionArgs args) {
    foreach (var argumentPosition in _outputArgumentsToValidate) {
        if (args.Arguments[argumentPosition] != null) continue;
        var parameterName = _parameterNames[argumentPosition];
        throw new InvalidOperationException($"Out parameter
'{parameterName}' is null.");
    }

    if (!_validateReturnValue || args.ReturnValue != null) return;

    if (_isProperty) {
        throw new InvalidOperationException($"Return value of property
'_{memberName}' is null.");
    }
    throw new InvalidOperationException($"Return value of method
'_{memberName}' is null.");
}
```

The `OnSuccess()` method validates the *output parameters* to validate. If any arguments are null, then `InvalidOperationException` will be thrown. The next thing we need to do is add private class for extracting method information. Add the following class to the bottom of the `DisallowNonNullAspect` class before the closing brace:

```
private class MethodInformation { }
```

Add the following three constructors to the `MethodInformation` class:

```
private MethodInformation(ConstructorInfo constructor) :
this((MethodBase)constructor) {
    IsConstructor = true;
    Name = constructor.Name;
}

private MethodInformation(MethodInfo method) : this((MethodBase)method) {
    IsConstructor = false;
    Name = method.Name;
    if (method.IsSpecialName &&
        (Name.StartsWith("set_", StringComparison.Ordinal) ||
         Name.StartsWith("get_", StringComparison.Ordinal))) {
        Name = Name.Substring(4);
        IsProperty = true;
    }
    ReturnParameter = method.ReturnParameter;
}
```

```
}

private MethodInformation(MethodBase method)
{
    IsPublic = method.IsPublic;
}
```

These constructors differentiate between constructors and methods and perform the necessary initialization of the method. Add the following method:

```
private static MethodInformation CreateInstance(MethodInfo method) {
    return new MethodInformation(method);
}
```

The `CreateInstance()` method creates a new instance of the `MethodInformation` class based on the `MethodInfo` data of the method passed in and returns that instance. Add the `GetMethodInformation()` method:

```
public static MethodInformation GetMethodInformation(MethodBase methodBase)
{
    var ctor = methodBase as ConstructorInfo;
    if (ctor != null) return new MethodInformation(ctor);
    var method = methodBase as MethodInfo;
    return method == null ? null : CreateInstance(method);
}
```

This method casts `methodBase` to `ConstructorInfo` and checks for `null`. If `ctor` is not `null`, then a new `MethodInformation` class is generated based on the constructor. However, if `ctor` is `null`, then `methodBase` is cast to `MethodInfo`. If the method is not `null`, then the `CreateInstance()` method is called, passing in the method. Otherwise, `null` is returned. Finally, add the following properties to the class:

```
public string Name { get; private set; }
public bool IsProperty { get; private set; }
public bool IsPublic { get; private set; }
public bool IsConstructor { get; private set; }
public ParameterInfo ReturnParameter { get; private set; }
```

These properties are properties of the method that has the aspect applied. We have now finished writing our validation aspect. You can now use the validator to allow nulls by attaching the `[AllowNull]` attribute. You can disallow nulls by attaching `[DisallowNonNullAspect]`. Now, we'll add our transaction concern.

Adding the transaction concern

Transactions are processes that must run to completion or rollback. Add a new folder to the class library called `Transactions`, and then add the `RequiresTransactionAspect` class:

```
[Serializable]
[AttributeUsage(AttributeTargets.Method)]
public sealed class RequiresTransactionAspect : OnMethodBoundaryAspect {
    public override void OnEntry(MethodExecutionArgs args) {
        var transactionScope = new
TransactionScope(TransactionScopeOption.Required);
        args.MethodExecutionTag = transactionScope;
    }

    public override void OnSuccess(MethodExecutionArgs args) {
        var transactionScope = (TransactionScope)args.MethodExecutionTag;
        transactionScope.Complete();
    }

    public override void OnExit(MethodExecutionArgs args) {
        var transactionScope = (TransactionScope)args.MethodExecutionTag;
        transactionScope.Dispose();
    }
}
```

The `OnEntry()` method starts the transaction, the `OnSuccess()` method completes the exception, and the `OnExit()` method disposes of the transaction. To use the aspect, add `[RequiresTransactionAspect]` to your method. To log any exceptions that prevent the completion of the transaction, you can also assign the `[ExceptionAspect(consoleOutput: false)]` aspect. Next, we'll add our resource pool concern.

Adding the resource pool concern

Resource pools are a good way to improve performance when multiple instances of an object are expensive to create and destroy. We will create a very simple resource pool for our needs. Add a folder called `ResourcePooling`, and then add the `ResourcePool` class:

```
public class ResourcePool<T> {
    private readonly ConcurrentBag<T> _resources;
    private readonly Func<T> _resourceGenerator;

    public ResourcePool(Func<T> resourceGenerator) {
        _resourceGenerator = resourceGenerator ??

```



```
                throw new
ArgumentNullException(nameof(resourceGenerator));
        _resources = new ConcurrentBag<T>();
    }

    public T Get() => _resources.TryTake(out T item) ? item :
_resourceGenerator();
    public void Return(T item) => _resources.Add(item);
}
```

This class creates a new resource generator and stores resources in `ConcurrentBag`. When an item is requested, it issues a resource from the pool. If one does not exist, then it is created, added to the pool, and issued to the caller:

```
var pool = new ResourcePool<Course>(() => new Course()); // Create a new
pool of Course objects.
var course = pool.Get(); // Get course from pool.
pool.Return(course); // Return the course to the pool.
```

The code you've just seen shows you how to use the `ResourcePool` class to create a pool, obtain a resource, and return it to the pool.

Adding the configuration settings concern

Configuration settings should always be centralized. Since desktop applications store their settings in the `app.config` file and web applications store their settings in `Web.config`, we can use `ConfigurationManager` to access the application settings. Add the `System.Configuration.Configuration` NuGet library to your class library and test the harness. Then, add a folder called `Configuration` and the following `Settings` class:

```
public static class Settings {
    public static string GetAppSetting(string key) {
        return System.Configuration.ConfigurationManager.AppSettings[key];
    }

    public static void SetAppSettings(this string key, string value) {
        System.Configuration.ConfigurationManager.AppSettings[key] = value;
    }
}
```

This class will get and set app settings in the `Web.config` file and the `App.config` file. To include the class in your files, add the following `using` statement:

```
using static CrossCuttingConcerns.Configuration.Settings;
```

The following code shows you how to use the methods:

```
Console.WriteLine(GetAppSetting("Greeting"));
"Greeting".SetAppSettings("Goodbye, my friends!");
Console.WriteLine(GetAppSetting("Greeting"));
```

Using the static import, you don't have to include the `class` prefix. You can extend the `Settings` class to get connection strings or to do whatever configuration you need in your apps.

Adding the instrumentation concern

Our final cross-cutting concern is that of instrumentation. We use instrumentation to profile our application and see how long it takes for methods to execute. Add a folder to the class library called `Instrumentation`, and then add the `InstrumentationAspect` class, as shown:

```
[Serializable]
[AttributeUsage(AttributeTargets.Method)]
public class InstrumentationAspect : OnMethodBoundaryAspect {
    public override void OnEntry(MethodExecutionArgs args) {
        LogFile.AppendTextToFile("Profile.log",
            $"\nMethod: {args.Method.Name}, Start Time: {DateTime.Now}");
        args.MethodExecutionTag = Stopwatch.StartNew();
    }

    public override void OnException(MethodExecutionArgs args) {
        LogFile.AppendTextToFile("Exception.log",
            $"\n{DateTime.Now}: {args.Exception.Source} -
{args.Exception.Message}");
    }

    public override void OnExit(MethodExecutionArgs args) {
        var stopwatch = (Stopwatch)args.MethodExecutionTag;
        stopwatch.Stop();
        LogFile.AppendTextToFile("Profile.log",
            $"\nMethod: {args.Method.Name}, Stop Time: {DateTime.Now},
Duration: {stopwatch.Elapsed}");
    }
}
```

As you can see, the instrumentation aspect only applies to methods, times the start and stop times of the method, and logs the profile information to the `Profile.log` file. If an exception is encountered, then the exception is logged to the `Exception.log` file.

We now have a functional and reusable cross-cutting concerns library. Let's summarize what we have learned in this chapter.

Summary

We've learned some valuable information. We started off by looking at the decorator pattern and then the proxy pattern. The proxy pattern provides objects that act as substitutes for real service objects used by clients. A proxy receives a client request, performs the necessary work, and then passes the request to the service object. Since proxies share the same interfaces as the services they substitute, they are interchangeable.

After covering the proxy pattern, we then moved onto AOP with PostSharp. We saw how we can use aspects and attributes together to decorate code so that at compile-time, it injects code to perform the required operations, such as exception handling, logging, auditing, and security. We extended the aspect framework by developing our own aspect and looked at how to use PostSharp and the decorator pattern to address the cross-cutting concerns of configuration management, logging, auditing, security, validation, exception handling, instrumentation, transactions, resource pooling, caching, threading, and concurrency.

In the next chapter, we will look at using tools to help you improve your code quality. But before then, test your knowledge and then further your reading.

Questions

1. What is a cross-cutting concern and what does AOP stand for?
2. What is an aspect and how do you apply one?
3. What is an attribute and how do you apply one?
4. How do the aspects and attributes work together?
5. How does the build process work with aspects?

Further reading

- The PostSharp home page: <https://www.postsharp.net/>

12

Using Tools to Improve Code Quality

As a programmer, enhancing code quality is one of your chief concerns. Improving the quality of your code demands the utilization of various tools. Tools designed to improve your code and also speed up development include **code metrics**, **quick actions**, the **JetBrains dotTrace** profiler, **JetBrains ReSharper**, and **Telerik JustDecompile**.

This is the main thing that we'll be doing in this chapter, with the following topics:

- Defining good-quality code
- Performing code cleanup and calculating code metrics
- Performing code analysis
- Using quick actions
- Using the JetBrains dotTrace profiler
- Using JetBrains ReSharper
- Using Telerik JustDecompile

By the end of this chapter, you will have gained the following skills:

- Using code metrics to measure software complexity and maintainability
- Using quick actions to make changes using a single command
- Profiling your code and analyzing bottlenecks with JetBrains dotTrace
- Refactoring code using JetBrains ReSharper
- Decompiling code and generating a solution using Telerik JustDecompile

Technical requirements

- The source code for this book: <https://github.com/PacktPublishing/Clean-Code-in-C->
- Visual Studio 2019 Community Edition or higher: <https://visualstudio.microsoft.com/downloads/>
- Telerik JustDecompile: <https://www.telerik.com/products/decompiler.aspx>
- JetBrains ReSharper Ultimate: <https://www.jetbrains.com/resharper/download/#section=resharper-installer>

Defining good-quality code

Good code quality is an essential software property. Financial loss, wasted time and effort, and even death can result from poor-quality code. High-standard code will have the qualities of **Performance, Availability, Security, Scalability, Maintainability, Accessibility, Deployability, and Extensibility (PASSMADE)**.

Performant code is small, only does what it needs to do, and is very fast. Performant code will not grind a system to a halt. Things that grind a system to a halt are file **input/output (I/O)** operations, memory usage, and **central processing unit (CPU)** usage. Low-performing code is a candidate for refactoring.

Availability refers to the software being continually available at the required level of performance. Availability is the ratio between the **time the software is functional (tsf)** to the **total time it is expected to function (ttef)**—for example, $tsf=700$; $ttef=744$. $700 / 744 = 0.9409 = 94.09\%$ availability.

Secure code is the code that properly validates input to protect against invalid data formats, an invalid range data, and malicious attacks and that fully authenticates and authorizes its users. Secure code is also code that is fault-tolerant. For example, if you are halfway through transferring money from one account to another and the system crashes, the operation should ensure the data remains intact, with no money taken from the account in question.

Scalable code is code that can safely handle exponential growth in the number of users using the system without the system grinding to a halt. So, whether the software handles one request per hour or a million requests per hour, there is no degradation in the performance of the code and no downtime due to excessive load.

Maintainability refers to how easy it is to fix bugs and add new functionality. Maintainable code should be well organized and easy to read. There should be low coupling and high cohesion so that the code can be easily maintained and extended.

Accessible code is code that people with limited abilities find easy to modify and use according to their needs. Examples include user interfaces with high contrast, a narrator for dyslexic and blind people, and so on.

Deployability focuses on the users of the software—will the users be standalone, remote access, or local network users? Whatever type the user is, the software should be very easy to deploy without any issues.

Extensibility refers to how easy it is to extend an application by adding new features to it. Spaghetti code and highly coupled code with low cohesion make this very difficult and error-prone. Such code can be very hard to read and maintain and is not easy to extend. Therefore, extensible code is code that is easy to read, easy to maintain, and—thus—easy to add new features to.

From the PASSMADE requirements of good-quality code, you can easily infer the kinds of problems that could arise from failing to meet these requirements. Failure to meet these requirements would lead to poor-performing code that becomes frustrating and unusable. Clients would be annoyed by increased downtime. Hackers would be able to exploit vulnerabilities in code that is not secure. The software would degrade exponentially as more users are added to the system. Code would be hard to fix or extend, and in some cases impossible to fix or extend. Users with limited abilities would not be able to modify the software around their limitations, and deployment would be a configuration nightmare.

Code metrics to the rescue. Code metrics enable developers to measure code complexity and maintainability and thus help us to identify code that is a candidate for refactoring.

With Quick Actions, you can use a single command to refactor C# code, such as extracting code out into its own method. JetBrains dotTrace allows you to profile your code and find performance bottlenecks. Further, JetBrains ReSharper is a Visual Studio productivity extension that enables you to analyze code quality and detect code smells, enforce coding standards, and refactor code. And Telerik JustDecompile helps you to decompile existing code for troubleshooting, and to create **Intermediate Language (IL)**, C#, and VB.NET projects from. This is particularly useful if you no longer have the source code and need to maintain or extend the compiled code. You can even generate debug symbols for the compiled code.

Let's take a deeper look at the tools mentioned, starting with code metrics.

Performing code cleanup and calculating code metrics

Before we look at how to gather code metrics, we first need to know what they are and why they are useful to us. **Code metrics** are mainly concerned with software complexity and maintainability. They help us to see how we can improve the maintainability of our source code and reduce source code complexity.

The code metrics that Visual Studio 2019 calculates for you consist of the following:

- **Maintainability index:** Code maintainability is an essential component of **Application Lifecycle Management (ALM)**. Until software reaches its end of life, it must be maintained. The harder the code base is to maintain, the shorter the lifespan of the source code before a complete replacement is required. Writing new software to replace an ailing system is far more work and is more expensive when compared to maintaining an existing system. The measurement for code maintainability is known as the maintainability index. This value is an integer value between 0 and 100. Here are the maintainability index ratings, their colors, and their meanings:
 - Any value from *20 and above* has a *green* rating for good maintainability.
 - Moderately maintainable code is *between 10 and 19*, with a *yellow* rating.
 - Anything *below 10* has a rating of *red*, meaning that it is hard to maintain.
- **Cyclomatic complexity:** Code complexity, also known as cyclomatic complexity, refers to the various code paths through the software. The more paths there are, the more complex the software is. And the more complex the software is, the harder it is to test and maintain. Complex code can lead to more error-prone software releases and can make it hard to maintain and extend the software. Hence, it is advisable that code complexity should be kept to a minimum.

- **Depth of Inheritance:** The Depth of Inheritance and class coupling metrics are affected by the popular programming paradigm called **Object-Oriented Programming (OOP)**. With OOP, classes are able to inherit from other classes. A class that is inherited from is known as a base class. Classes that inherit from a base class are known as subclasses. The metric for the number of classes that inherit from each other is known as the Depth of Inheritance.

The deeper the level of inheritance, the more chance you have of errors in derived classes if something is changed in one of the base classes. The ideal Depth of Inheritance is 1.

- **Class coupling:** OOP allows class coupling. Class coupling arises when a class is directly referenced by a parameter, a local variable, a return type, a method call, a generic or template instantiation, base classes, interface implementations, fields defined on extra types, and an attribute decoration.

The class coupling code metric determines the level of coupling between classes. To make code easier to maintain and extend, class coupling should be kept to an absolute minimum. In OOP, one way to achieve this is to use interface-based programming. This way, you avoid directly accessing a class. The benefit of this method of programming is that you can swap classes in and out, as long as they implement the same interface. Poor-quality code has high coupling and low cohesion, but good-quality code has low coupling and high cohesion.

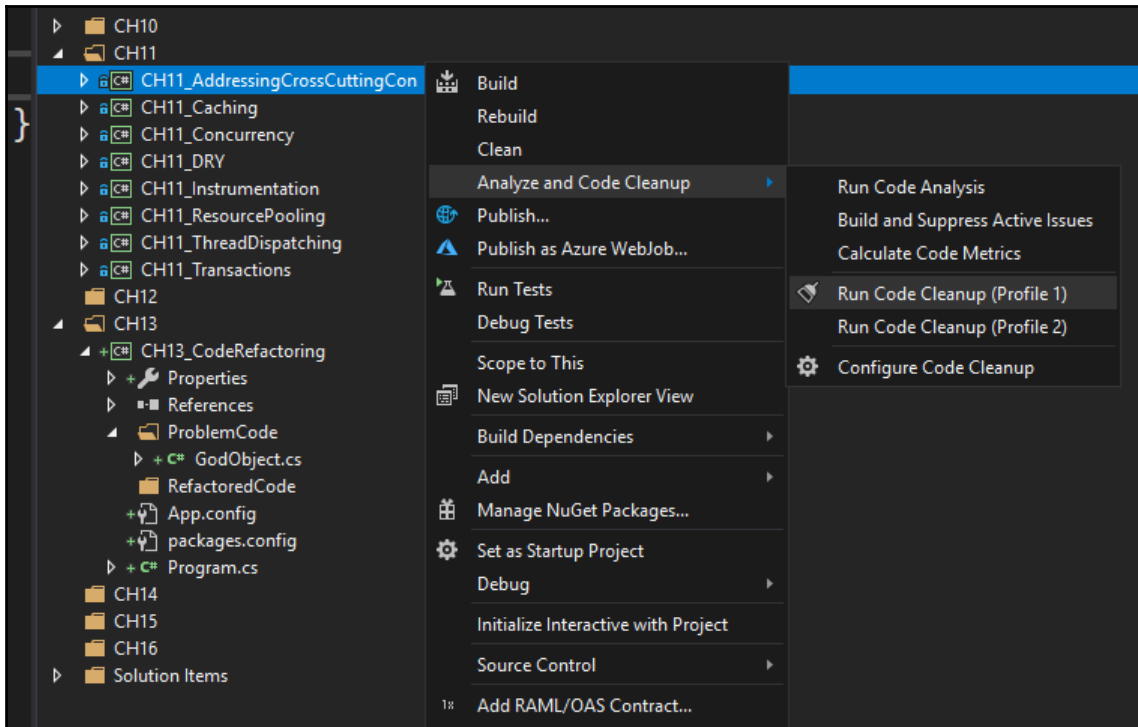


Ideally, software should be highly cohesive with low coupling, because it makes programs easier to test, maintain, and extend.

- **Lines of source code:** The complete count of the lines of your source code, including blank lines, is measured by the lines of source code metric.
- **Lines of executable code:** The measure of operations in executable code is measured by the lines of executable code metric.

Now that you have a heads-up on what code metrics are and which measurements are available in Visual Studio 2019 version 16.4 onward, it's time to see them in action, as follows:

1. Open any project you like within Visual Studio.
2. Right-click on the project.
3. Select **Analyze and Code Cleanup | Run Code Cleanup (Profile 1)**, as illustrated in the following screenshot:



4. Now, select **Calculate Code Metrics**.
5. You should see the **Code Metrics Results** window appear, as shown in the following screenshot:

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
CH10/CH10_AddressCrossCuttingConcerns (Debug)	87	62	6	89	517	127
CH10_AddressCrossCuttingConcerns	93	13	1	25	109	28
CH10_AddressCrossCuttingConcerns.ArchitecturalConstraints	70	2	5	7	22	5
CH10_AddressCrossCuttingConcerns.Aspects	85	16	6	34	119	28
CH10_AddressCrossCuttingConcerns.Attributes	74	20	6	41	160	53
CH10_AddressCrossCuttingConcerns.DecoratorPattern	95	6	2	6	61	8
ConcreteComponent	100	1	1	3	8	1
Operation() : void	100	1	1	1	4	1
ConcreteDecorator	86	2	2	4	19	5
ConcreteDecorator(IComponent)	100	1	2	2	2	0
Operation() : void	76	1	3	14	14	5
Decorator	96	2	1	1	14	2
component : IComponent	100	0	1	1	1	0
Decorator(IComponent)	96	1	1	1	5	1
Operation() : void	100	1	1	1	5	1
IComponent	100	1	0	0	4	0
Operation() : void	100	1	0	0	1	0
CH10_AddressCrossCuttingConcerns.Interfaces	100	1	0	0	7	0
IFilterable	100	1	0	0	4	0
ApplyFilter() : void	100	1	0	0	1	0
CH10_AddressCrossCuttingConcerns.ProxyPattern	98	4	1	3	39	5
IService	100	1	0	0	4	0
Request() : void	100	1	0	0	1	0
Proxy	94	2	1	3	16	4
logSource : LogSource	93	0	1	1	1	1
_service : IService	100	0	1	1	1	0
Proxy(IService)	96	1	1	1	5	1
Request() : void	88	1	2	2	6	2
Service	100	1	1	2	7	1
Request() : void	100	1	1	1	4	1

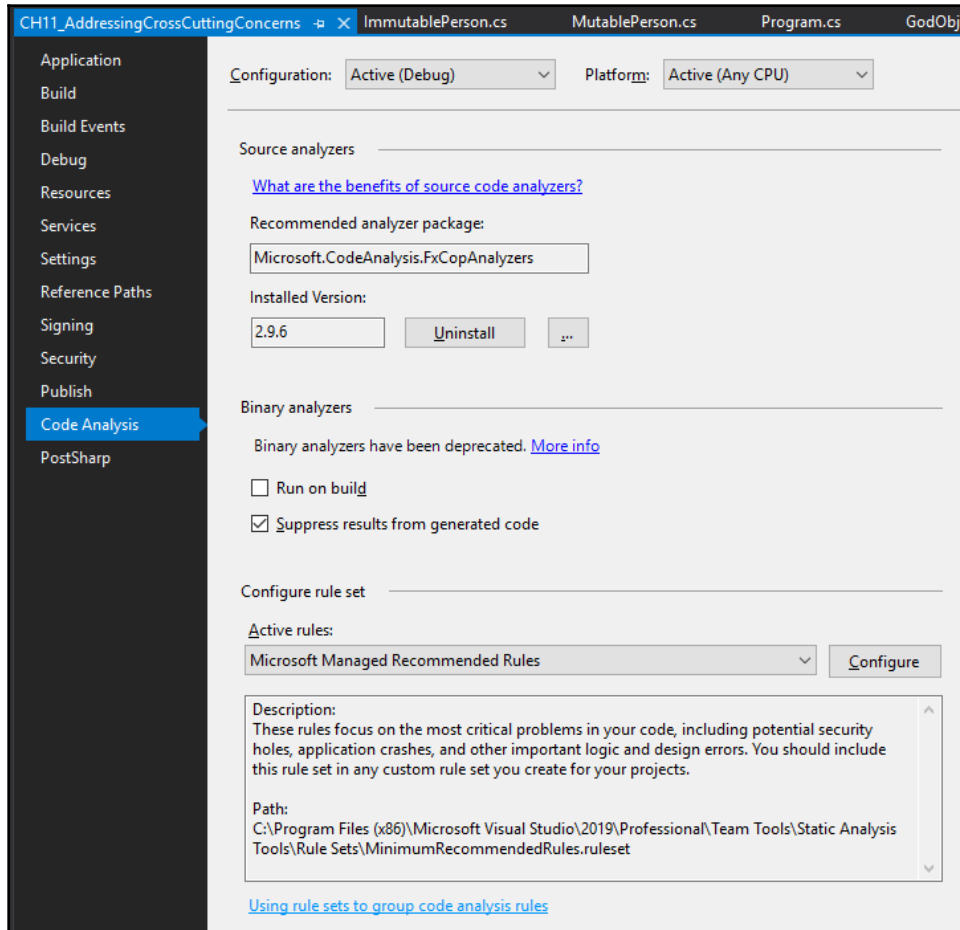
As you can see from the screenshot, all our classes, interfaces, and methods are marked with a *green* indicator. This means that the selected project is one that is maintainable. If any of these lines were marked yellow or red, then you would need to address them and refactor them to make them green. Well, we've covered code metrics, and so, naturally, we move on to cover code analysis.

Performing code analysis

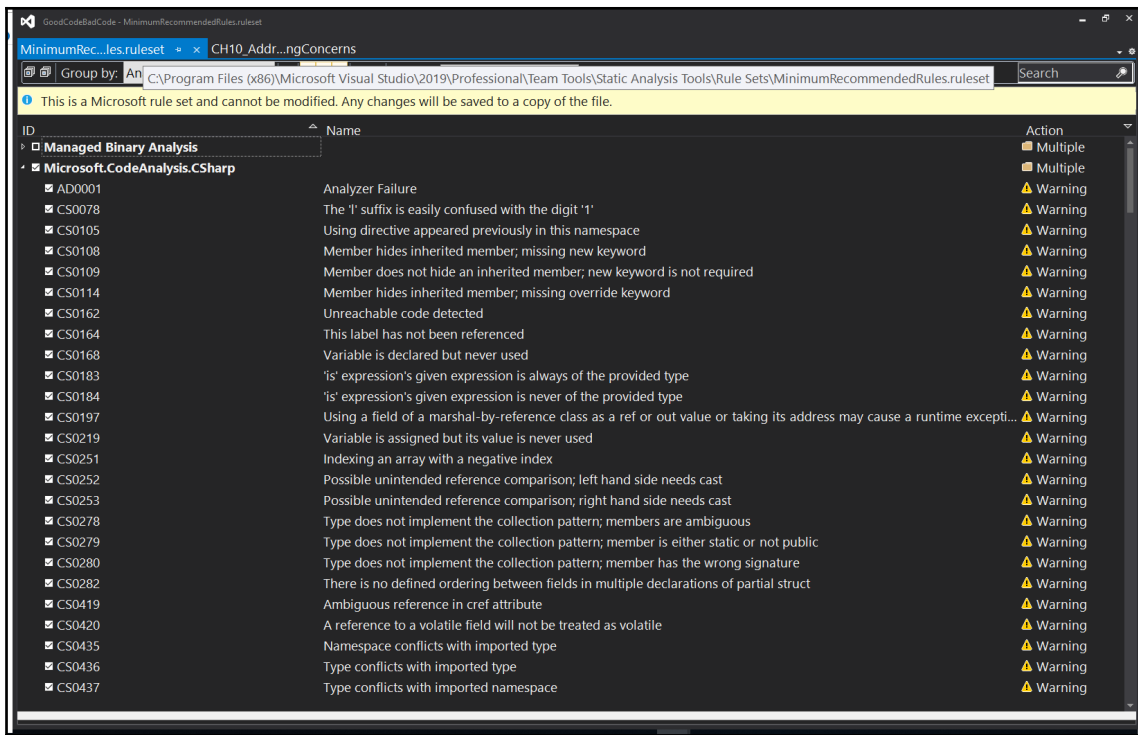
To help developers identify potential problems with their source code, Microsoft provides the **Code Analysis** tool as part of Visual Studio. **Code Analysis** performs a static source code analysis. The tool will identify design flaws, issues with globalization, security problems, issues with performance, and interoperability problems.

Open the book solution, and select the **CH11_AddressingCrossCuttingConcerns** project. Then, from the **Project** menu, select **Project |**

CH11_AddressingCrossCuttingConcerns | Properties from the menu. On the properties page for the project, select **Code Analysis**, as illustrated in the following screenshot:

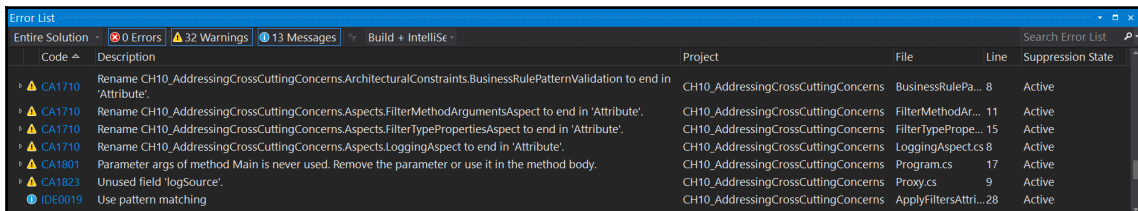


As shown in the preceding screenshot, if you see that the recommended analyzer package is not installed, click on **Install** to install it. Once installed, the version number will be displayed in the installed version box. For me, it is version **2.9.6**. By default, the active rules are **Microsoft Managed Recommended Rules**. The location of this ruleset, as shown in the description, is **C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\Team Tools\Static Analysis Tools\Rule Sets\MinimumRecommendedRules.ruleset**. Open the file. It will open as a Visual Studio tool window, as shown here:






As you can see in the preceding screenshot, you can select and deselect rules. When you close the window, you will be prompted to save any changes. To run a code analysis, go to **Analyze and Code Cleanup | Code Analysis**. In order to view the results, you will need the **Error List** window to be open. You can open it from the **View** menu.

Once you have run the code analysis, you will see a list of errors, warnings, and messages. You can address each and every one of them to improve the overall quality of your software. A sample of these can be seen in the following screenshot:

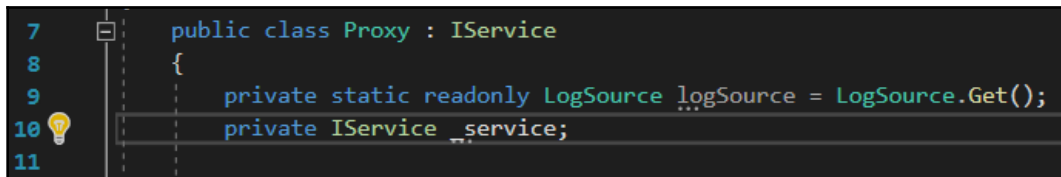


From the preceding screenshot, you can see that the CH10_AddressingCrossCuttingConcerns project has 32 warnings and 13 messages. If we were to work on the warnings and messages, we would get them down to 0 messages and 0 warnings. So, now that you have seen how to use code metrics to see how maintainable your software is and you've analyzed it to see what improvements you can make, it's now time to look at quick actions.

Using quick actions

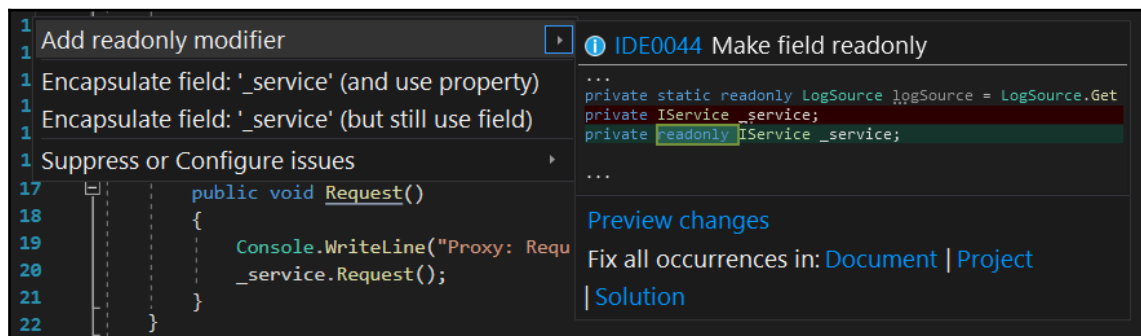
Another handy tool that I like to use is the **Quick Action** tool. Appearing as a screwdriver , a lightbulb , or an error light bulb  on a line of code, quick actions enable you to use a single command that will generate code, refactor code, suppress warnings, perform code fixes, and add using statements.

Since the CH10_AddressingCrossCuttingConcerns project had 32 warnings and 13 messages, we can use this project to see the quick actions in action. Have a look at the following screenshot:



```
7 public class Proxy : IService
8 {
9     private static readonly LogSource logSource = LogSource.Get();
10    private IService _service;
11 }
```

Looking at the preceding screenshot, we see the lightbulb on line 10. If we click on the lightbulb, the following menu pops up:



If we click on **Add readonly modifier**, the `readonly` access modifier is placed after the `private` access modifier. Have a go yourself at using quick actions to modify the code. It is fairly straightforward once you get the hang of it. Once you have had a play around with quick actions, move on to look at the JetBrains dotTrace profiling tool.

Using the JetBrains dotTrace profiler

The JetBrains dotTrace profiler is a part of JetBrains ReSharper Ultimate license. Since we will be looking at both tools, I recommend that you download and install JetBrains ReSharper Ultimate before we continue.



JetBrains does have a trial version available if you don't already own a copy. There are versions available for Windows, macOS, and Linux.

The JetBrains dotTrace profiling tool works with Mono, .NET Framework, and .NET Core. All application types are supported by the profiler, and you can use the profiler to analyze and track down performance issues with your code base. The profiler will help you to get to the bottom of such problems that cause 100% CPU usage, 100% of the disk I/O, maxing out the memory or running into overflow exception, and many other issues.

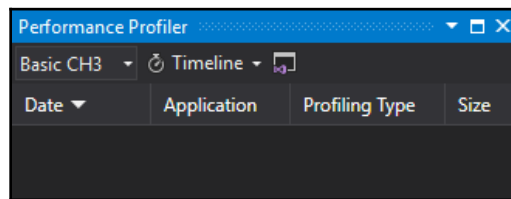
Many applications perform **HyperText Transfer Protocol (HTTP)** requests. The profiler will analyze how the application is processing these requests, and it will also do the same with **Structured Query Language (SQL)** queries on a database. Static methods and unit tests can be profiled, and you can view the results from within Visual Studio. There is also a standalone version that you can use.

There are four basic profiling options—**Sampling**, **Tracing**, **Line-by-Line**, and **Timeline**. The first time you start looking at the performance of an application, you may decide to use **Sampling**, which provides an accurate measurement of call time. **Tracing** and **Line-by-Line** offer more detailed profiling, but they do add more overhead (memory and CPU usage) to the program being profiled. **Timeline** is similar to sampling and collects application events over time. Between them, there is no problem that can't be tracked down and resolved.

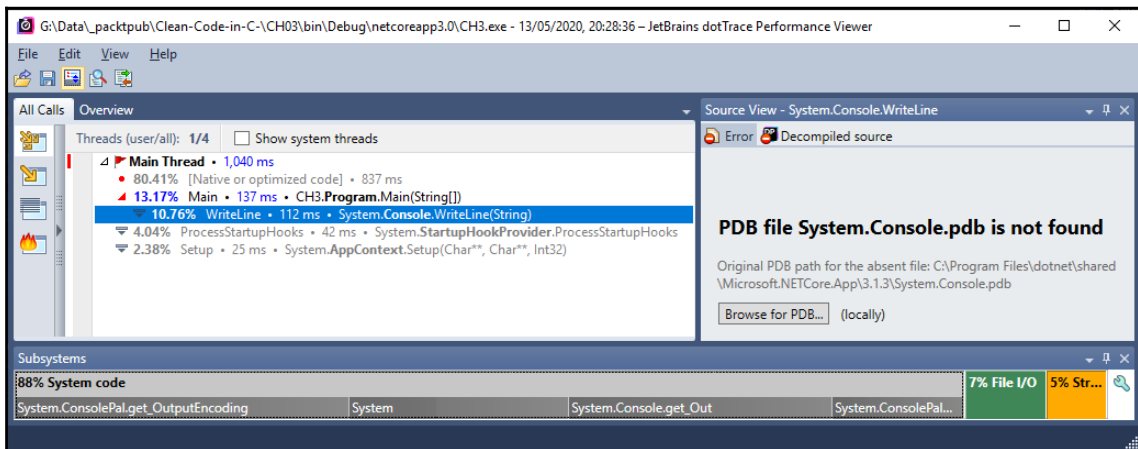
Advanced profiling options include real-time performance counters, thread time, real-time CPU instructions, and thread cycle time. The real-time performance counters measure the time between method entry and exit. Thread time measures the thread running time. Based on the CPU register, the real-time CPU instructions provide an accurate time of method entry and exit.

The profiler can attach to running .NET Framework 4.0 (or later) or .NET Core 3.0 (or later) applications and processes, profile local applications, and profile remote applications. These include standalone applications; .NET Core applications; **Internet Information Services (IIS)**-hosted web applications; IIS Express-hosted applications; .NET Windows Services, **Windows Communication Foundation (WCF)** services; Windows Store and **Universal Windows Platform (UWP)** applications; any .NET processes (started after you run the profiling session); desktop or console applications based on Mono; and Unity editor or standalone Unity applications.

To access the profiler in Visual Studio 2019 from the menu, select **Extensions | ReSharper | Profile | Show Performance Profiler**. In the following screenshot, you can see that nothing has been profiled yet. Also, the currently selected project to be profiled is set to **Basic CH3**, and the profiling type is set to **Timeline**. We will profile **CH3** using **Sampling** to profile our project by expanding the **Timeline** drop-down functionality and selecting **Sampling**, as illustrated in the following screenshot:



If you want to sample a different project, just expand the **Project** drop-down list and select the project that you want to profile. The project will be built, and the profiler started. Your project will then run and be shut down. The results will then be displayed in the dotTrace profiling application, as shown in the following screenshot:

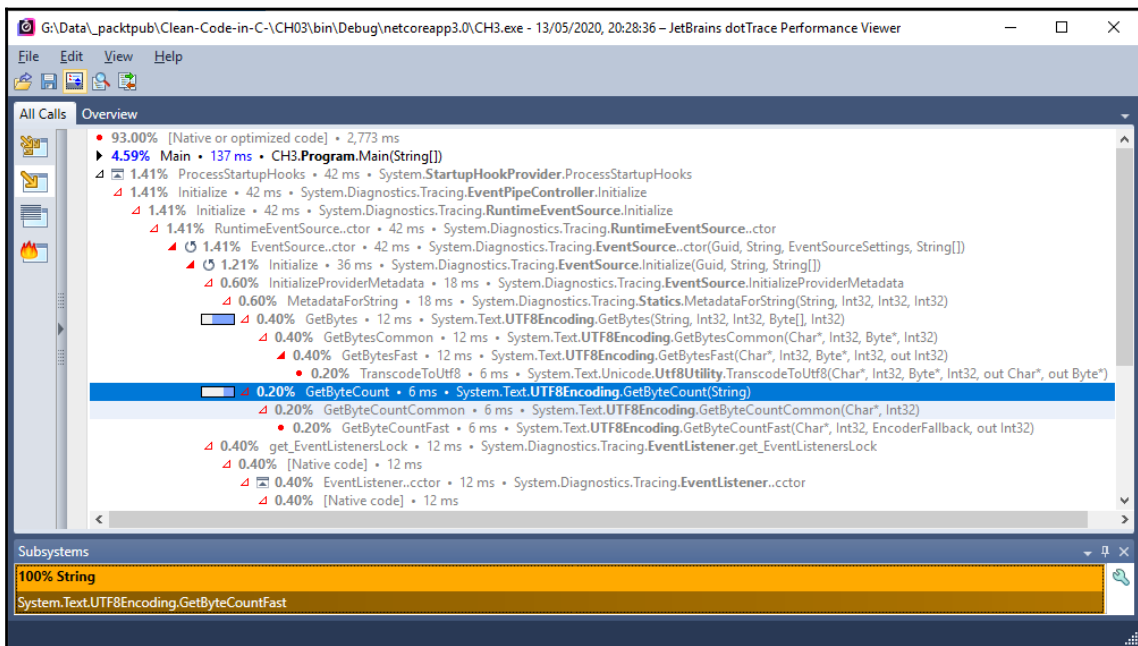


From the preceding screenshot, you can see that the first of four threads are being shown. This is the thread for our program. The other threads are for the supporting processes that enable our program to run along with the finalizer thread that is responsible for exiting the program and cleaning up system resources.

The **All Calls** menu items down the left-hand side comprise the following:

- **Thread Tree**
- **Call Tree**
- **Plain List**
- **Hot Spots**

The current option selected is the **Thread Tree**. Let's have a look at the expanded **Call Tree** in the following screenshot:

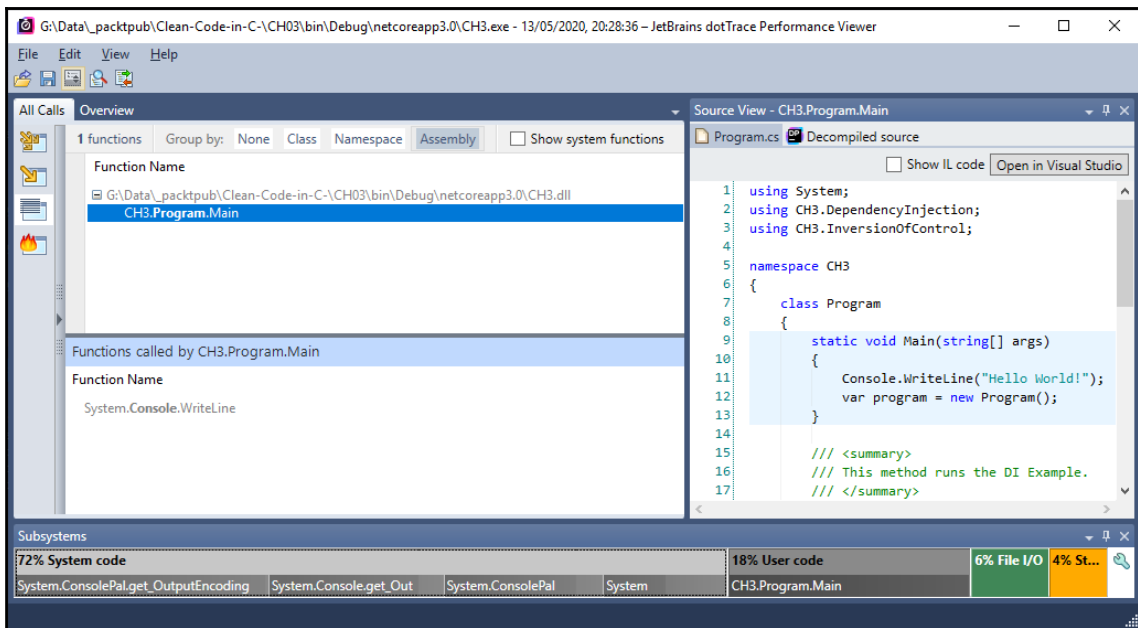


The profiler shows you the complete **Call Tree** for your code, and that includes system code as well as your own code. You can see the percentage of time spent on making the call. This allows you to identify any long-running methods and address them.

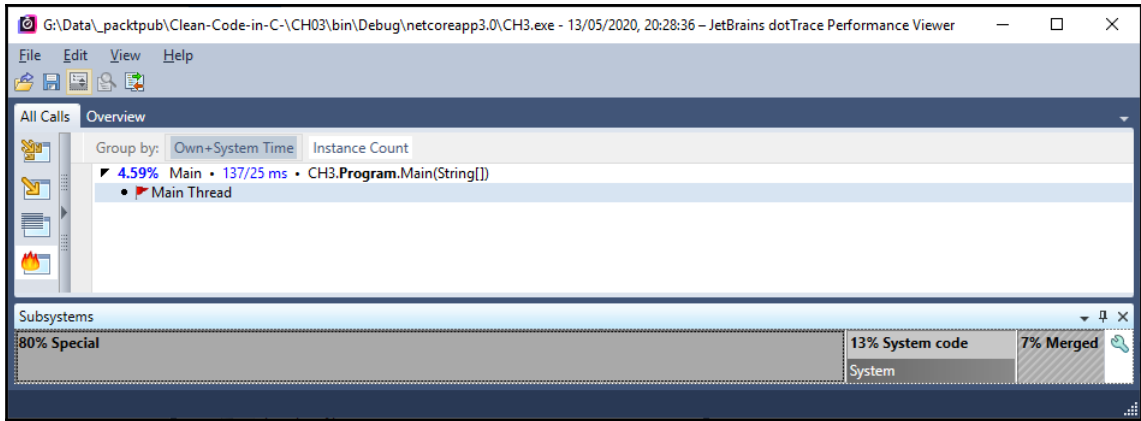
Now, we'll look at the **Plain List**. As you can see with the **Plain List** view in the screenshot that follows, we can group it according to the following criteria:

- **None**
- **Class**
- **Namespace**
- **Assembly**

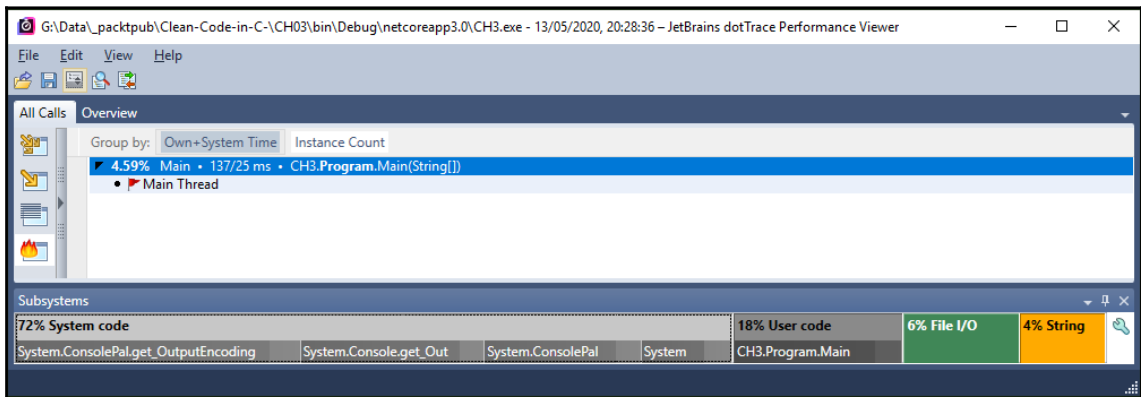
You can see the preceding criteria in the following screenshot:



When you click on an item in the list, you get to view the source code for the class where the method resides. This is useful, as you can see the code where the problem lies and what needs to be done. The last sampling profile screen we'll look at is the **Hot Spots** view, illustrated in the following screenshot:



The profiler is showing that the **Main Thread**, which is our code's starting point, only takes 4.59% of the processing time. If you click on the root, 18% of the code is our user code, and 72% of the code is system code, as shown in the following screenshot:



We have only touched the surface with this profiling tool. There is more to it, and I encourage you to try it out for yourself. The main purpose of this chapter is to introduce you to the tools that are available to you.



For further information on how to use JetBrains dotTrace, I refer you to their online learning materials, at <https://www.jetbrains.com/profiler/documentation/documentation.html>.

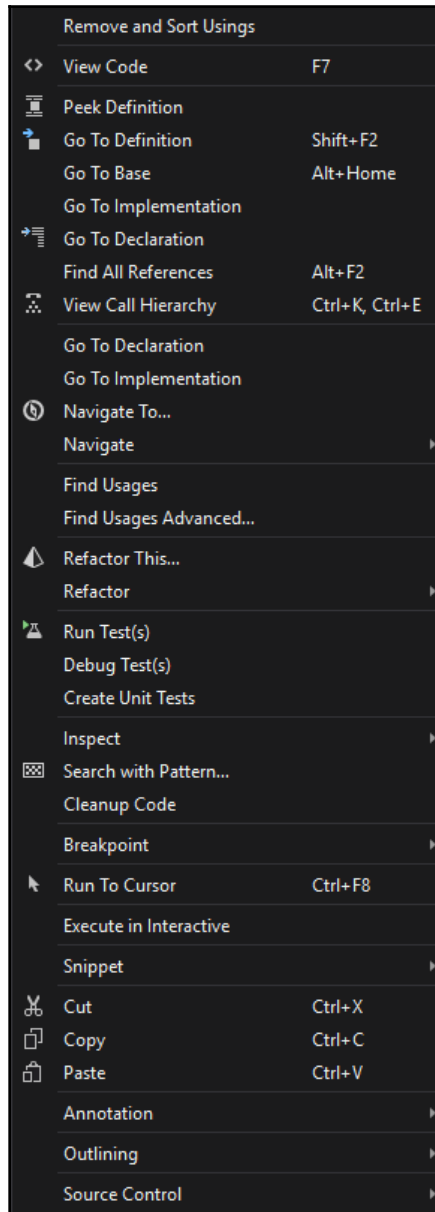
Next up, we look at JetBrains ReSharper.

Using JetBrains ReSharper

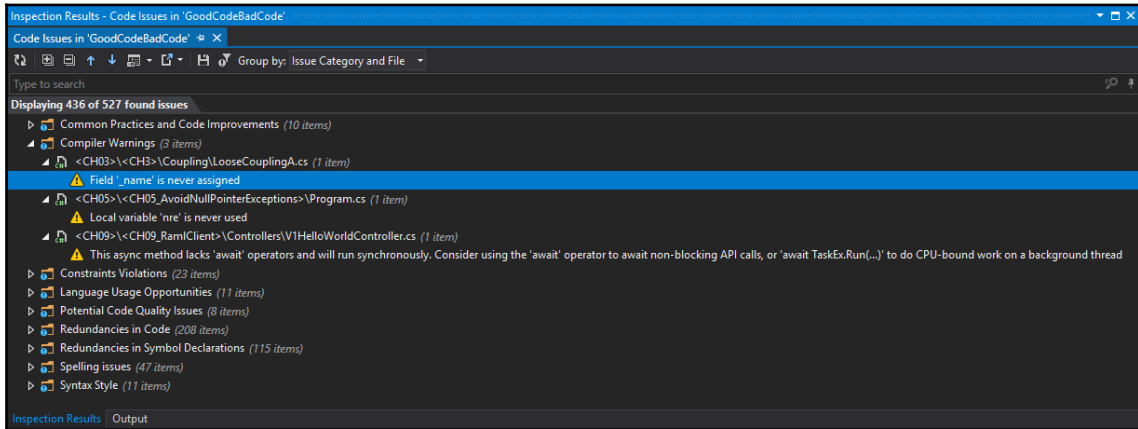
In this section, we look at how JetBrains ReSharper can help you improve your code. ReSharper is quite an extensive tool, and just as with the profiler, which is a part of the Ultimate edition of ReSharper, we will only be touching the surface, but you will hopefully come to an appreciation of what the tool is and what it can do for you to improve your Visual Studio coding experience. Here are a few benefits of using ReSharper:

- With ReSharper, you can perform an analysis of your code quality.
- It will provide options to improve your code, remove code smells, and fix coding problems.
- With the navigation system, you are able to completely traverse your solution and jump to any item of interest. You have many different helpers that include extended IntelliSense, code reorganization, and more.
- Refactoring benefits from ReSharper's offerings that can be localized or solution-wide.
- You can also generate source code using ReSharper, such as base class and superclasses, and inline methods.
- Here, code can be cleaned up in keeping with your company's coding policies to get rid of unused imports and other unused code.

You can visit the **ReSharper** menu from the Visual Studio 2019 **Extensions** menu. When in the code editor, right-clicking the mouse on a piece of code will bring up a context menu with the appropriate menu items. The **ReSharper** menu item in the context menu is **Refactor This...**, as shown in the following screenshot:



Now, from the Visual Studio 2019 menu, run **Extensions | ReSharper | Inspect | Code Issues in Solution**. ReSharper will process the solution and then display the **Inspection Results** window, as shown in the following screenshot:



As you can see in the preceding screenshot, ReSharper found 527 issues with our code—436 of which are being displayed. These problems include common practices and code improvements, compiler warnings, constraint violations, language usage opportunities, potential code quality issues, redundancies in code, redundancies in symbol declarations, spelling issues, and syntax style.

If we expand **Compiler Warnings**, we see that there are three problems, as follows:


- The `_name` field is never assigned.
- The `nre` local variable is never used.
- This `async` method lacks `await` operators and will run synchronously. Use the `await` operator to await non-blocking **Application Programming Interface (API)** calls, or `await TaskEx.Run(...)` to do CPU-bound work on a background thread.

These problems are variable declarations that don't get assigned or used, and an `async` method lacking an `await` operator that will run synchronously. If you click on the first warning, it will take you to the line of code that is never assigned. Looking at the class, you can see that the string is declared and used, but it is never assigned. Since we check if the string contains `string.Empty`, we can assign that value to the declaration. Hence, the updated line will be as follows:

```
private string _name = string.Empty;
```

Since the `_name` variable still highlights, we can hover over it and see what the problem is. The Quick Action informs us that the `_name` variable can be marked read-only. Let's add the `readonly` modifier. So, the line now becomes this:

```
private readonly string _name = string.Empty;
```

If we click on the refresh  button, we will find that the number of issues found is now 526. Yet, we fixed two problems. So, should the number be 525? Well, no. The second problem that we fixed was not a problem picked up by ReSharper, but an improvement picked up by Visual Studio Quick Actions. So, ReSharper is showing the correct number of issues it has detected.

Let's have a look at the potential code quality issue for the `LooseCouplingB` class. ReSharper reports a possible `System.NullReferenceException` within this method. Let's look at the code first, as follows:

```
public LooseCouplingB()
{
    LooseCouplingA lca = new LooseCouplingA();
    lca = null;
    Debug.WriteLine($"Name is {lca.Name}");
}
```

And sure enough, we do have `System.NullReferenceException` staring us in the face. We'll look at the `LooseCouplingA` class, to confirm which members should be set to `null`. Also, the member to be set is `_name`, as illustrated in the following code snippet:

```
public string Name
{
    get => _name.Equals(string.Empty) ? StringIsEmpty : _name;

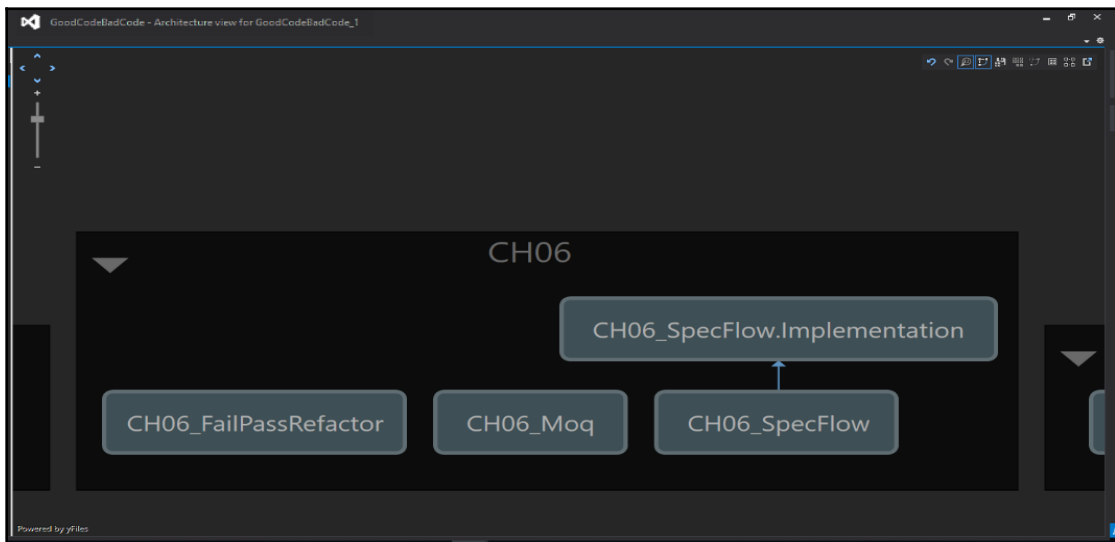
    set
    {
        if (value.Equals(string.Empty))
            Debug.WriteLine("Exception: String length must be greater than
zero.");
    }
}
```

However, `_name` is being checked for empty. And so, really, the code should be setting `_name` to `string.Empty`. So, our fixed constructor in `LooseCouplingB` becomes the following:

```
public LooseCouplingB()
{
    var lca = new LooseCouplingA
    {
        Name = string.Empty
    };
    Debug.WriteLine($"Name is {lca.Name}");
}
```

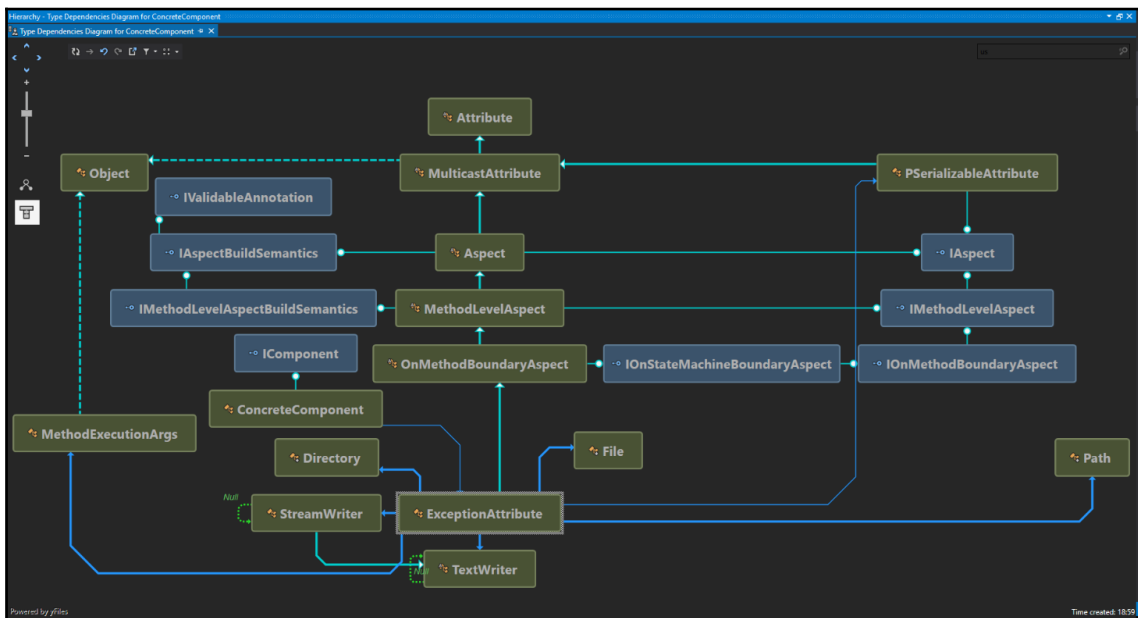
Now, if we refresh the **Inspection Results** window, our list of issues has gone down by five, because apart from correctly assigning the `Name` property, we made use of the language usage opportunity to simplify our instantiation and initialization, which was detected by ReSharper. Have a play around with the tool and eliminate the problems found in the **Inspection Results** window.

ReSharper can also generate *dependency diagrams*. To generate a dependency diagram for our solution, select **Extensions | ReSharper | Architecture | Show Project Dependency Diagram**. This will display the project dependency diagram for our solution. The black container box called `CH06` is the namespace, and the gray/blue boxes prefixed with `CH06_` are projects, as illustrated in the following screenshot:

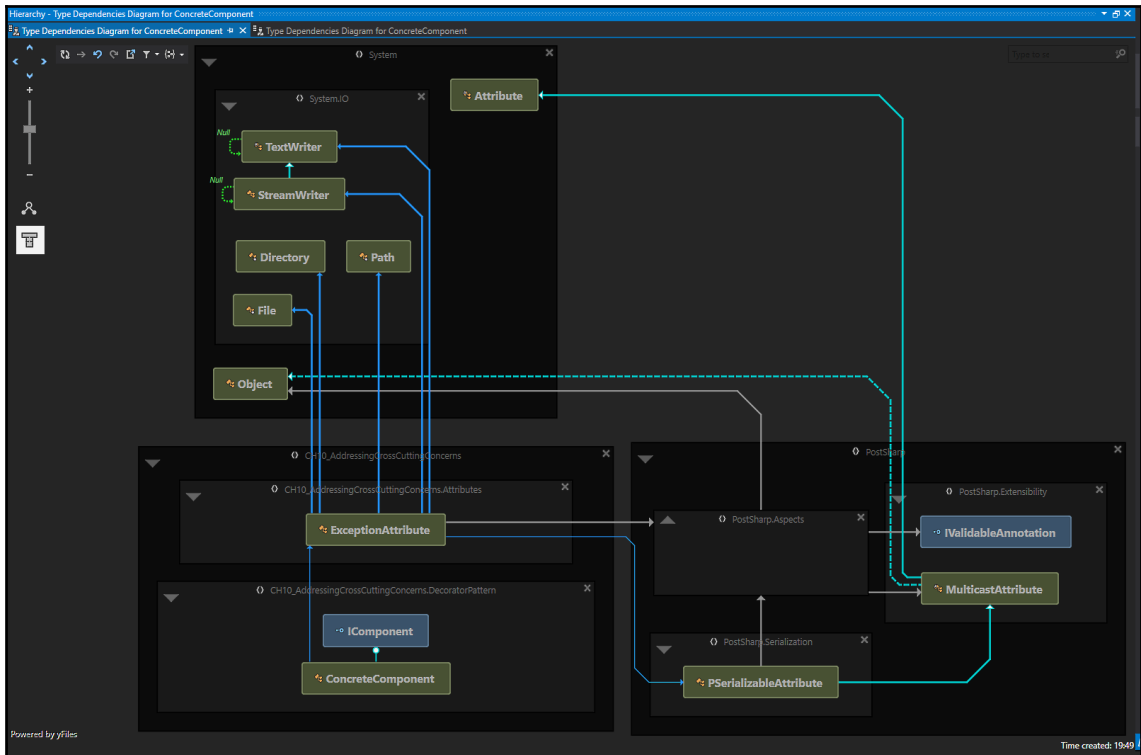


As you can see from the project dependency diagram in the CH06 namespace, there is a project dependency between CH06_SpecFlow and CH06_SpecFlow.Implementation. Similarly, you can also generate type dependency diagrams using ReSharper. Select **Extensions | ReSharper | Architecture | Type Dependencies Diagram**.

If we generate the diagram for `ConcreteClass` in the CH10_AddressingCrossCuttingConcerns project, then the diagram will be generated, but only the `ConcreteComponent` class will be initially displayed. Right-click the `ConcreteComponent` box on the diagram and select **Add All Referenced Types**. You will see the addition of the `ExceptionAttribute` class and the `IComponent` interface. Right-click on the `ExceptionAttribute` class and select **Add All Referenced Types**, and you end up with the following:



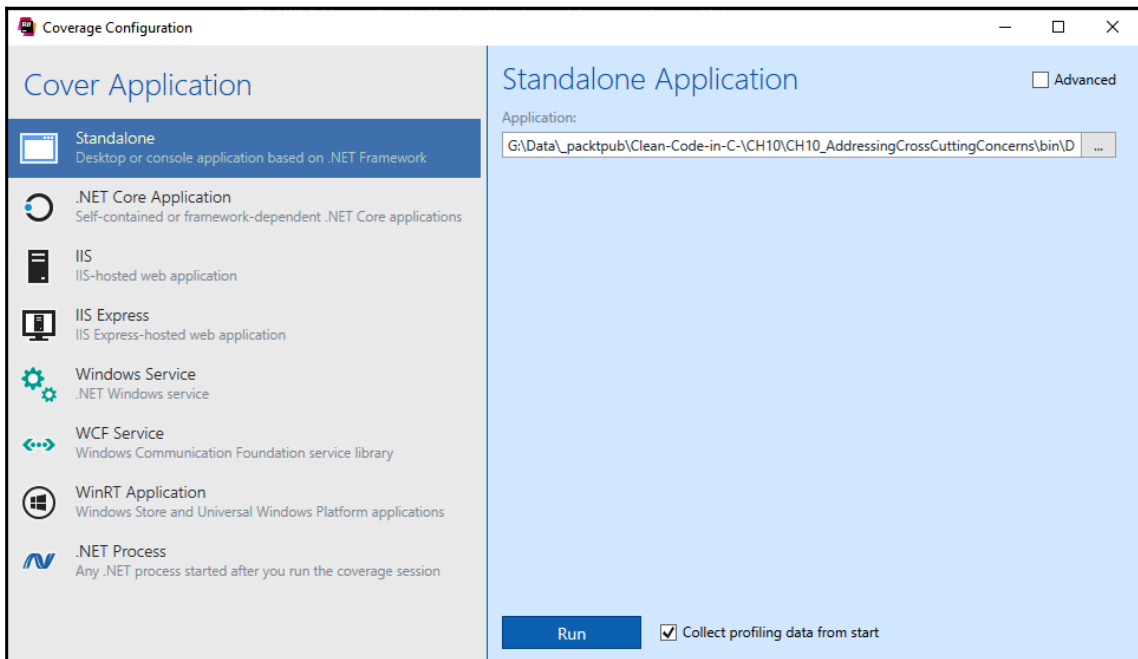
What's really wonderful about this tool is that you can order the diagram elements by namespace. This can be really useful for massive solutions with multiple large projects and deep-nested namespaces. Though it's good that we can right-click on code and go to the item declaration, you can't beat visually seeing the lay of the land in terms of the project that you are working on, and that is why this tool can be really useful. Here is an example of a typed dependencies diagram organized by namespaces:



Many a time, I could have really used a diagram such as this in my day-to-day work. This diagram is technical documentation that will help developers find their way around a complex solution. They will be able to see which namespaces are available and how everything is interlinked. This will empower developers with the correct knowledge as to where new classes, enums, and interfaces should be placed when performing new development, but also, they will know where to find objects if they are performing maintenance. This diagram is also good for finding duplicate namespaces, interfaces, and object names.

Let's now look at coverage. Proceed as follows:

1. Select **Extensions | ReSharper | Cover | Cover Application**.
2. The **Coverage Configuration** dialog will be displayed, and the default selected option will be **Standalone**.
3. Select your executable.
4. You can select a .NET app from the `bin` folder.
5. The following screenshot shows the **Coverage Configuration** dialog:



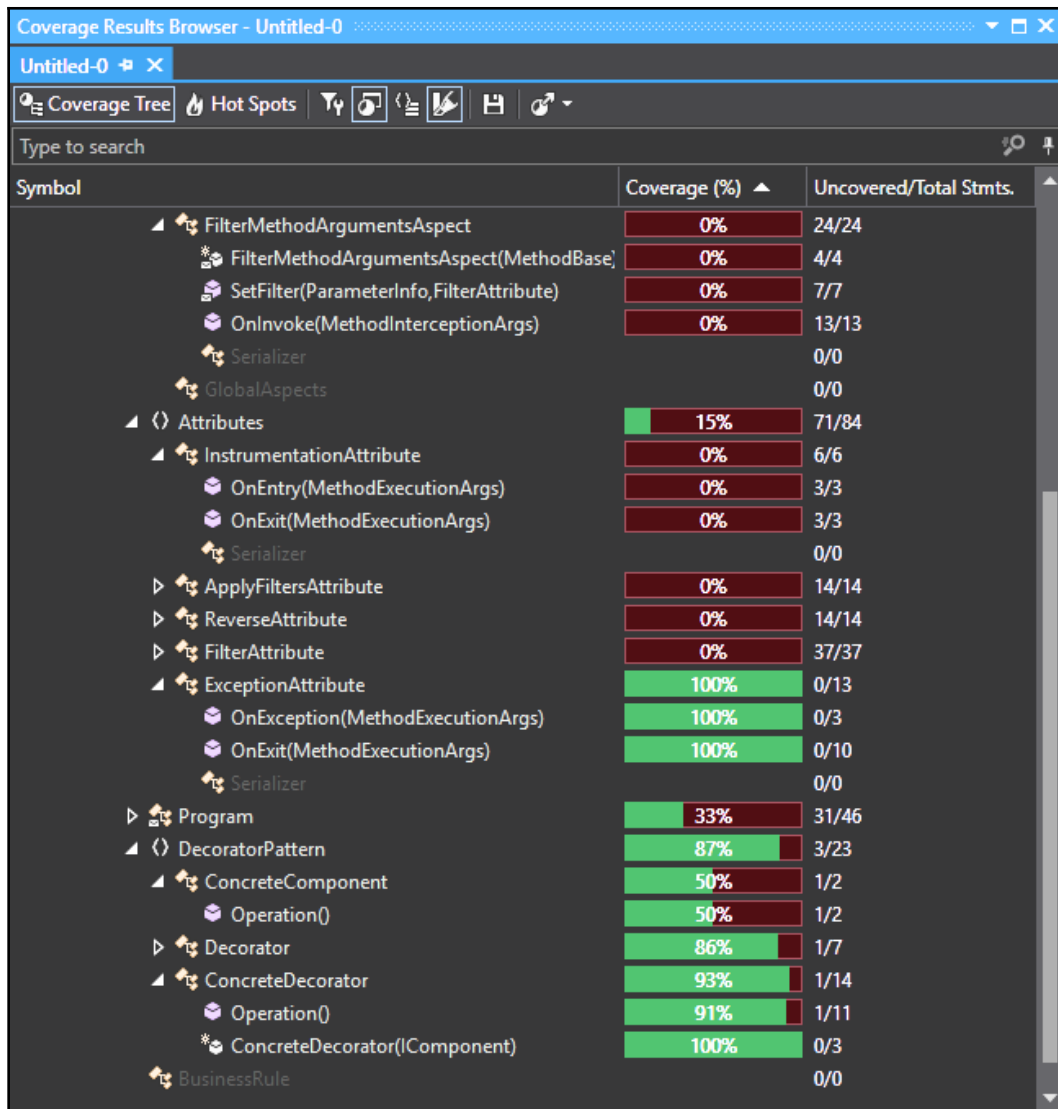
- Click the **Run** button to start the application and collect profiling data. ReSharper will display the following dialog:



The application will then run. As the application is running, the coverage profiler will be collecting data. Our selected executable is a console application that displays the following data:

```
G:\Data\_packtpub\Clean-Code-in-C-\CH10\CH10_AddressCrossCuttingConcerns\bin\Debug\CH10_AddressCrossCuttingConcerns.exe
Starting.
Debug | Trace | ConcreteDecorator..ctor({CH10_AddressCrossCuttingConcerns.DecoratorPattern.ConcreteComponent}) |
Succeeded.
Audit: [Member Name: .ctor, Operation: ConcreteDecorator..ctor({CH10_AddressCrossCuttingConcerns.DecoratorPattern.Con
creteComponent}), Time: 15/05/2020 20:12:51]
Debug | Trace | ConcreteDecorator.Operation() | Starting.
Operation: try block.
Debug | Trace | Decorator.Operation() | Starting.
Debug | Trace | ConcreteComponent.Operation() | Starting.
Oops! The method or operation is not implemented.
Audit: [Member Name: Operation, Operation: ConcreteComponent.Operation(), Time: 15/05/2020 20:12:51]
Warning | Trace | ConcreteComponent.Operation() | Failed: exception = {System.NotImplementedException}.
System.NotImplementedException: The method or operation is not implemented.
at CH10_AddressCrossCuttingConcerns.DecoratorPattern.ConcreteComponent.Operation() in G:\Data\_packtpub\Clea
n-Code-in-C-\CH10\CH10_AddressCrossCuttingConcerns\DecoratorPattern\ConcreteComponent.cs:line 14
Warning | Trace | Decorator.Operation() | Failed: exception = {System.NotImplementedException}.
System.NotImplementedException: The method or operation is not implemented.
at CH10_AddressCrossCuttingConcerns.DecoratorPattern.ConcreteComponent.Operation() in G:\Data\_packtpub\Clean-
Code-in-C-\CH10\CH10_AddressCrossCuttingConcerns\DecoratorPattern\ConcreteComponent.cs:line 14
at CH10_AddressCrossCuttingConcerns.DecoratorPattern.Decorator.Operation() in G:\Data\_packtpub\Clean-Code-in-
C-\CH10\CH10_AddressCrossCuttingConcerns\DecoratorPattern\Decorator.cs:line 20
Audit: [Member Name: Operation, Operation: Decorator.Operation(), Time: 15/05/2020 20:12:51]
Operation: catch block.
The method or operation is not implemented.
Debug | Trace | ConcreteDecorator.Operation() | Succeeded.
Audit: [Member Name: Operation, Operation: ConcreteDecorator.Operation(), Time: 15/05/2020 20:12:51]
Audit: [Member Name: DecoratorPatternExample, Operation: Program.DecoratorPatternExample(), Time: 15/05/2020 20:12:51]
```

- Click the console window, and then press any key to exit. The coverage dialog will disappear, and storage will then be initialized. Finally, the **Coverage Results Browser** window will be displayed, as shown here:



This window contains really useful information. It provides a visual indicator of code that was not called, marked in red. The code that was executed is marked in green. Using this information, you can see if the code is dead code that can be removed, or was not executed due to the path taken through the system but is still required, or was commented out for testing purposes, or was simply not called because the developer forgot to add the call in the correct place or a condition check was wrong.

To go to the item of interest, you just have to double-click on the item, and then you will be taken to the specific code you are interested in. Our `Program` class only covers 33% of the code. So, let's double-click `Program`, and see what's the matter. The resulting output is shown in the following code block:

```
static void Main(string[] args)
{
    LoggingServices.DefaultBackend = new ConsoleLoggingBackend();
    AuditServices.RecordPublished += AuditServices_RecordPublished;
    DecoratorPatternExample();
    //ProxyPatternExample();
    //SecurityExample();

    //ExceptionHandlingAttributeExample();

    //SuccessfulMethod();
    //FailedMethod();

    Console.ReadKey();
}
```

As you can see from the code, the reason why some of our code was not covered is because calls to the code were commented out for testing purposes. We can leave the code as it is (which we will do in this case). However, you can also remove the dead code or reinstate the code by removing the comments. Now, you know why the code is not being covered.

Well, now that you've been introduced to ReSharper and had a look at tools to assist you in writing good, clean C# code, it is time to look at our next tool, called Telerik JustDecompile.

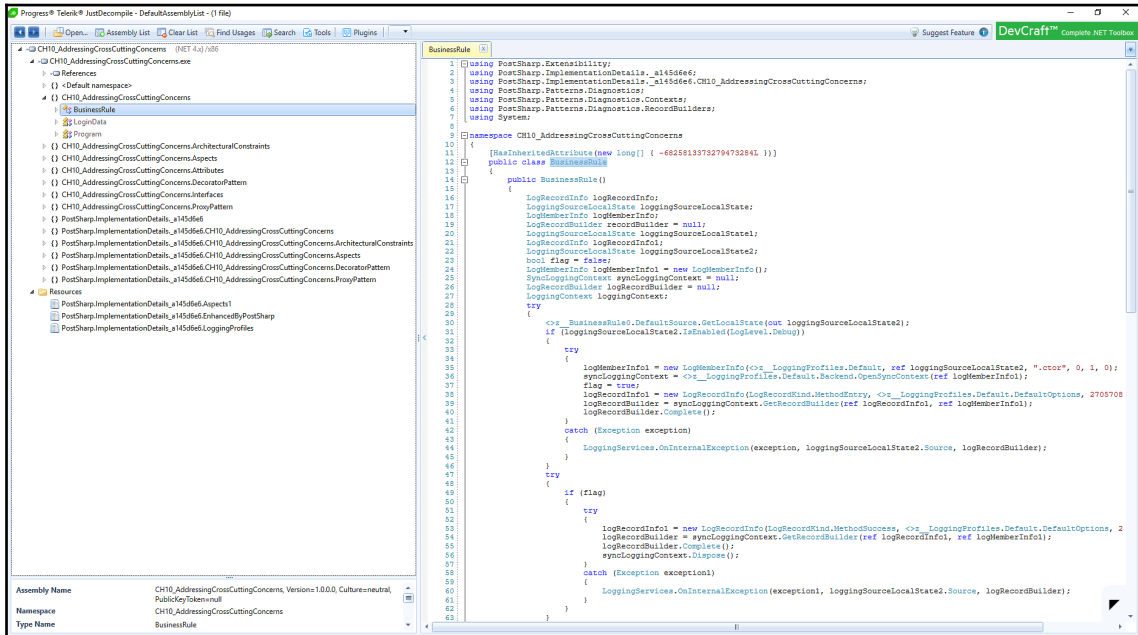
Using Telerik JustDecompile

I have used Telerik JustDecompile on a number of occasions, for things such as tracking down bugs in third-party libraries, recovering essential project source code that has been lost, checking the strength of assembly obfuscation, and for learning purposes. It is a tool that I highly recommend, as over the years it has proven its worth many times.

The decompilation engine is open source and you can obtain the source code from <https://github.com/telerik/justdecompileengine>, so you are free to contribute to the project and write your own extensions for it. You can download Windows Installer from the Telerik website, at <https://www.telerik.com/products/decompiler.aspx>. All source code is fully navigable. The decompiler is available as a standalone application or as a Visual Studio extension. You create VB.NET or C# projects from assemblies that you decompile, and you extract and save resources from the decompiled assemblies.

Download and install Telerik JustDecompile. We will then go through the decompilation process, and generate a C# project from an assembly. You may be prompted to install other tools during the installation process, but you can deselect the other offerings from Telerik.

Run the Telerik JustDecompile standalone application. Find a .NET assembly and then drag it into the left pane of Telerik JustDecompile. It will decompile the code and display the code tree on the left. If you select an item on the left, the code is shown on the right, as you can see in the screenshot:



As you can see, the decompilation process is fast and it does a pretty good job of decompiling our assembly. The decompilation is not perfect, but in most cases, it does the job. Proceed as follows:

1. In the dropdown to the right of the **Plugins** menu item, select **C#**.
2. Then, click on **Tools | Create Project**.
3. You will sometimes be prompted to select the .NET version to target; other times, not.
4. Then, you will be asked where to save the project.
5. The project will then be written to that location.

You can then open the project in Visual Studio and work on it. Should you encounter any problems, Telerik logs the issues in your code and provides an email. You can always email them with any issues you encounter. They are good at responding to and fixing problems.

Well, we have completed our look at the tools in this chapter, so now, let's look at what we have learned in summary.

Summary

In this chapter, you have seen how code metrics provide several measurements of code quality, and how easy it is to generate them. Code metrics include the number of lines—including blank lines—versus the number of executable lines of code, the cyclomatic complexity, the level of cohesion and coupling, and how maintainable your code is. The refactoring color codes are green for good, yellow for ideally needs refactoring, and red for definitely needs refactoring.

You then saw how easy it is to provide a static code analysis of projects and view the results. Viewing and modifying rulesets that govern what gets analyzed and what doesn't get analyzed was also covered. Then, you experienced quick actions and saw how we can perform bug fixes, add using statements, and refactor code with a single command.

We then used the JetBrains dotTrace profiler to measure our application's performance, track down bottlenecks, and identify hungry methods that take up the most processing time. The next tool we looked at was JetBrains ReSharper, which enables us to inspect code for various problems and potential improvements. We identified a couple of them and made the necessary changes, and saw how easy it was to improve the code with this tool. Then, we looked at creating architectural diagrams for dependencies and type dependencies.

Finally, we looked at Telerik JustDecompile, a very useful tool that can be used to decompile assemblies and generate projects in either C# or VB.NET from them. This can be very useful when bugs are encountered or the program needs to be expanded, but you no longer have access to the existing source code.

In the chapters that follow, we will mainly be looking at code, and how we can refactor it. But for now, test your knowledge with the following questions and further your reading with the links provided in the *Further reading* section.

Questions

1. What are code metrics, and why should we use them?
2. Name six code metric measurements.
3. What is code analysis, and why is it useful?
4. What are quick actions?
5. What is JetBrains dotTrace used for?
6. What is JetBrains ReSharper used for?
7. Why use Telerik JustDecompile to decompile assemblies?

Further reading

- Official Microsoft documentation on code metrics: <https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019>
- Official Microsoft documentation on Quick Actions: <https://docs.microsoft.com/en-us/visualstudio/ide/quick-actions?view=vs-2019>
- JetBrains dotTrace profiler: <https://www.jetbrains.com/profiler/>

13

Refactoring C# Code – Identifying Code Smells

In this chapter, we will look at problem code and how to refactor it. In the industry, problem code is normally termed **code smell**. It is code that compiles, runs, and does what it is supposed to do. The reason it is problem code is that it becomes unreadable, complex in nature, and makes the code base hard to maintain and extend further down the line. Such code should be refactored as soon as it's feasible to do so. It is technical debt, and in the long run, if you don't deal with it, it will bring the project to its knees. When this happens, you are looking at an expensive redesign and recoding of the application from scratch.

So what is refactoring? Refactoring is the process of taking existing code that works and rewriting it such that the code becomes clean. And as you have already discovered, clean code is easy to read, easy to maintain, and easy to extend.

In this chapter, we will cover the following topics:

- Identifying application-level code smells and how we can address them
- Identifying class-level code smells and how we can address them
- Identifying method-level code smells and how we can address them

After working your way through this chapter, you will have gained the following skills:

- Identifying different kinds of code smell
- Understanding why the code is classed as code smell
- Refactoring code smells so they become clean code

We'll start our look at refactoring code smells by looking at application-level code smells.

Technical requirements

You will need the following prerequisites for the chapter:

- Visual Studio 2019
- PostSharp

For the code files of the chapter, you can use this link: <https://github.com/PacktPublishing/Clean-Code-in-C-/tree/master/CH13>.

Application-level code smells

Application-level code smells are problem code scattered through the application and affect every layer. No matter what layer of the software you find yourself in, you will see the same problematic code appearing over and over again. If you don't address these issues now, then you will find that your software will start to die a slow and agonizing death.

In this section, we will look at the application-level code smells and how we can remove them. Let's start with Boolean blindness.

Boolean blindness

Boolean data blindness refers to the information loss as determined by functions that work on Boolean values. Using a better structure provides better interfaces and classes that keep data, making for a more pleasant experience in working with data.

Let's look at the problem of Boolean blindness via this code sample:

```
public void BookConcert(string concert, bool standing)
{
    if (standing)
    {
        // Issue standing ticket.
    }
    else
    {
        // Issue sitting ticket.
    }
}
```

This method takes a string for the concert name, and a Boolean value indicating whether the person is standing or seated. Now, we would call the code as follows:

```
private void BooleanBlindnessConcertBooking()
{
    var booking = new ProblemCode.ConcertBooking();
    booking.BookConcert("Solitary Experiments", true);
}
```

If someone new to the code saw the `BooleanBlindnessConcertBooking()` method, do you think they would know instinctively what `true` stands for? I think not. They would be blind to what it means. So they would have to either use IntelliSense or locate the method being referred to find the meaning. They are Boolean blind. So how can we cure them of this blindness?

Well, a simple solution would be to replace the Boolean with an enum. Let's start by adding our enum called `TicketType`:

```
[Flags]
internal enum TicketType
{
    Seated,
    Standing
}
```

Our enum identifies two types of ticket types. These are `Seated` and `Standing`. Now let's add our `ConcertBooking()` method:

```
internal void BookConcert(string concert, TicketType ticketType)
{
    if (ticketType == TicketType.Seated)
    {
        // Issue seated ticket.
    }
    else
    {
        // Issue standing ticket.
    }
}
```

The following code shows how to call the newly refactored code:

```
private void ClearSightedConcertBooking()
{
    var booking = new RefactoredCode.ConcertBooking();
    booking.BookConcert("Chrom", TicketType.Seated);
}
```

Now, if that new person came along and looked at this code, they would see that we are booking a concert to see the band `Chrom`, and that we want seated tickets.

Combinatorial explosion

Combinatorial explosion is a *by-product* of the same thing being performed by different pieces of code using different combinations of parameters. Let's look at an example that adds numbers:

```
public int Add(int x, int y)
{
    return x + y;
}

public double Add(double x, double y)
{
    return x + y;
}

public float Add(float x, float y)
{
    return x + y;
}
```

Here, we have three methods that all add numbers. The return types and parameters are all different. Is there a better way? Yes, through the use of generics. By using generics, you can have one single method that is capable of working with different types. And so, we will be using generics to solve our addition problem. This will allow us to have a single addition method that will accept either integers, doubles, or floats. Let's have a look at our new method:

```
public T Add<T>(T x, T y)
{
    dynamic a = x;
    dynamic b = y;
    return a + b;
}
```

This generic method is called with a specific type assigned to `T`. It performs the addition and returns the result. Only one version of the method is required for the different .NET types that can be added together. To call the code for `int`, `double`, and `float` values, we would do the following:

```
var addition = new RefactoredCode.Maths();  
addition.Add<int>(1, 2);  
addition.Add<double>(1.2, 3.4);  
addition.Add<float>(5.6f, 7.8f);
```

We have just eliminated three methods and replaced them with a single method that performs the same task.

Contrived complexity

When you can develop code with simple architecture, but instead implement an advanced and rather complex architecture, this is known as **contrived complexity**. Unfortunately, I have suffered having to work on such systems and it is a proper pain and cause of stress. What you find with such systems is that they tend to have a high turnover of staff. They lack documentation, and no one seems to know the system or has the ability to answer questions by onboarders—the poor souls who have to learn the system to maintain and extend it.

My advice to all super-intelligent software architects is that when it comes to software, **Keep It Simple, Stupid (KISS)**. Remember, the days of permanent employment with jobs for life appear to be a thing of the past now. Oftentimes, programmers are more for chasing the money than showing lifelong loyalty to the business. So with the business relying on the software for revenue, you need a system that is easy to understand, to onboard new staff, to maintain, and to extend. Ask yourself this question: If the systems that you are responsible for suddenly experienced yourself and all staff assigned to them walking out and finding new opportunities, would the new staff who take over be able to hit the ground running? Or would they be left stressed out and scratching their heads?

Also bear in mind that if you have only one person on the team who understands that system and they die, move on to a new location, or retire, where does that leave you and the rest of the team? And even more than that, where does it leave the business?

I cannot stress enough that you really are to KISS. The only reason for creating complex systems and not documenting them and sharing the architectural knowledge is to hold the business over a barrel so they keep you on and you can bleed them dry. Don't do it. In my experience, the more complicated a system is, the quicker it dies a death and has to be rewritten.

In Chapter 12, *Using Tools to Improve Code Quality*, you learned how to use Visual Studio 2019 tools to discover the *cyclomatic complexity* and *Depth of Inheritance*. You also learned how to produce dependency diagrams with ReSharper. Use these tools to discover problem areas in the code, then focus on those areas. Reduce cyclomatic complexity down to a value of 10 or less. And reduce the depth of inheritance on all objects down to no greater than 1.

Then, make sure all classes only perform the tasks that they are meant to. Aim to keep methods small. A good rule of thumb is to have no more than around 10 lines of code per method. As for method parameters, replace long parameter lists with parameter objects. And where you have a lot of `out` parameters, refactor the method to return a tuple or object. Identify any multithreading, and make sure that the code being accessed is thread-safe. You have seen in Chapter 9, *Designing and Developing APIs*, how to replace mutable objects with immutable ones to improve thread-safety.

Also, look for the Quick Tips icons. These will normally suggest one-click refactorings for the line of code they highlight. I recommend you use them. These were mentioned in Chapter 12, *Using Tools to Improve Code Quality*.

The next code smell to consider is the data clump.

Data clump

A **data clump** occurs when you see the same fields appearing together in different classes and parameter lists. Their names usually follow the same pattern. This is normally the sign that a class is missing from the system. The reduction in system complexity will come by identifying the missing class and generalizing it. Don't be put off by the fact that the class may only be small, and never think of a class as being unimportant. If there is a need for a class to simplify the code, then add it.

Deodorant comments

When a comment uses nice words to excuse bad code, this is known as a **deodorant comment**. If the code is bad, then refactor it to make it good and remove the comment. If you don't know how to refactor it to make it good, then ask for help. If there is no one to ask that can help you, then post your code on Stack Overflow. There are some very good programmers on that site that can be a real help to you. Just make sure to follow the rules when posting!

Duplicate code

Duplicate code is code that occurs more than once. Problems that arise from duplicate code include increased maintenance cost per duplication. When a developer is fixing a piece of code, it costs the business time and money. Fixing 1 bug is *technical debt (programmer's pay) x 1*. But if there are 10 duplications of that code, that's *technical debt x 10*. So the more that code is duplicated, the more expensive it is to maintain. Then there is the boredom factor of having to fix the same problem in multiple locations. And the fact that duplication may get overlooked by the programmer doing the bug fix.

It is best to refactor the duplicate code so that only one copy of the code exists. Often, the easiest way to do this is to add it to a new reusable class in your current project and place it in a class library. The benefit of placing reusable code in a class library is that other projects can use the same file.



In the present day, it is best to use the .NET Standard class library for building reusable code. The reason for this is that .NET Standard libraries can be accessed by all C# project types on Windows, Linux, macOS, iOS, and Android.

Another alternative for removing boilerplate code is to use **Aspect-Oriented Programming (AOP)**. We looked at AOP in the previous chapter. You essentially move boilerplate code into an aspect. The aspect then decorates the method it is applied to. When the method is compiled, the boilerplate code is then weaved into place. This enables you only to write code that meets the business requirement inside the method. The aspect applied to the method hides the code that is essential, but not part of what the business has asked for. This coding technique is nice and clean, and it works really well.

You can also write decorators using the decorator pattern, as you also saw in the previous chapter. The decorator wraps concrete class operations in such a way that you can add new code without affecting the expected operation of the code. A simple example would be to wrap the operation in a `try/catch` block as you saw previously in Chapter 11, *Addressing Cross-Cutting Concerns*.

Lost intent

If you can't easily understand the intent of the source code, then it has lost its intent.

The first thing to do is look at the namespace and the class name. These should indicate the purpose of the class. Then, check the contents of the class, and look for code that looks out of place. Once you have identified such code, refactor the code and place it in the right location.

The next thing to do is to look at each of the methods. Are they only doing one thing well or doing multiple things not so well? If yes, then refactor them. For large methods, look for code that can be extracted out into a method. Aim to make the code of the class read like a book. Keep refactoring the code until the intent is clear, and only what is in the class needs to be in the class.

Don't forget to put the tools to work that you learned how to use in [Chapter 12, Using Tools to Improve Code Quality](#). The mutation of variables is the code smell we will look at next.

The mutation of variables

The mutation of variables means they are hard to understand and reason about. This makes them difficult to refactor.

A mutable variable is one that gets changed multiple times by different operations. This makes reasoning about why is the value more difficult. Not only that, but because the variable is mutating from different operations, this makes it difficult to extract sections of code into other small and more readable methods. Mutable variables can also require more checking that adds complexity to the code.

Look to refactor small sections of code by extracting them out to methods. If there is a lot of branching and looping, see if there is an easier way to do things to remove the complexity. If you are using multiple `out` values, consider returning an object or tuple. Aim to remove the mutability of the variable to make it easier to reason about, and know why it is the value that it is, and from where it is getting set. Remember that the smaller the method is that holds a variable, the easier it will be to determine where the variable is getting set, and why.

Look at the following example:

```
[InstrumentationAspect]
public class Mutant
{
    public int IntegerSquaredSum(List<int> integers)
    {
        var squaredSum = 0;
        foreach (var integer in integers)
        {
            squaredSum += integer * integer;
        }
        return squaredSum;
    }
}
```


The method takes a list of integers. It then loops through the integers, squares them, and then adds them to the `squaredSum` variable that is returned when the method exits. Notice the iterations, and the fact that the local variable is getting updated in each iteration. We can improve on this using LINQ. The following code shows the improved, refactored version:

```
[InstrumentationAspect]
public class Function
{
    public int IntegerSquaredSum(List<int> integers)
    {
        return integers.Sum(integer => integer * integer);
    }
}
```

In our new version, we use LINQ. As you know from an earlier chapter, LINQ employs functional programming. As you can see here, there is no loop, and no local variable being mutated.

Compile and run the program, and you will see the following:

Both versions of the code produce the same output.

You will have noticed that both versions of the code have `[InstrumentationAspect]` applied to them. We added this aspect to our reusable library in Chapter 12, *Addressing Cross-Cutting Concerns*. When you run the code, you will find a `Logs` folder in the `Debug` folder. Open the `Profile.log` file in Notepad, and you will see the following output:

```
Method: IntegerSquaredSum, Start Time: 01/07/2020 11:41:43
Method: IntegerSquaredSum, Stop Time: 01/07/2020 11:41:43, Duration:
00:00:00.0005489
Method: IntegerSquaredSum, Start Time: 01/07/2020 11:41:43
Method: IntegerSquaredSum, Stop Time: 01/07/2020 11:41:43, Duration:
00:00:00.0000027
```

The output shows that the `ProblemCode.IntegerSquaredSum()` method was the slowest version, taking **548.9** nanoseconds to run. And that the `RefactoredCode.IntegerSquaredSum()` method was much faster, taking only **2.7** nanoseconds to run.

By refactoring the loop to use LINQ, we avoided mutating a local variable. And we also reduced the time it took to process the calculation by **546.2** nanoseconds. Such a small improvement is not noticeable to the human eye. But if you perform such calculations on big data, then you will experience a noticeable difference.

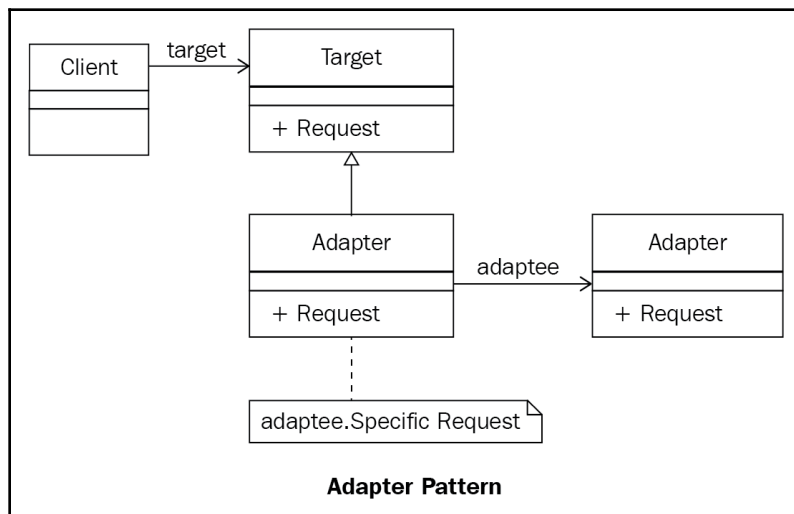
We'll now discuss the oddball solution.

The oddball solution

When you see a problem solved in a different way throughout the source code, this is known as an **oddball solution**. This can happen because of different programmers having their own style of programming, and no standards being put in place. It can also happen through ignorance of the system, in that the programmer does not realize a solution already exists.

A way to refactor oddball solutions is to write a new class that encompasses the behavior that is being repeated in different ways. Add the behavior to the class in the cleanest way that is the most performant. Then, replace the oddball solutions with the newly refactored behavior.

You can also unite different system interfaces using the **Adapter Pattern**:



The `Target` class is the domain-specific interface that is used by `Client`. An existing interface that needs adapting is called `Adaptee`. The `Adapter` class adapts the `Adaptee` class to the `Target` class. And finally, the `Client` class communicates objects that conform to the `Target` interface. Let's implement the adapter pattern. Add a new class called `Adaptee`:

```
public class Adaptee
{
    public void AdapteeOperation()
    {
        Console.WriteLine($"AdapteeOperation() has just executed.");
    }
}
```

The `Adaptee` class is very simple. It contains a method called `AdapteeOperation()` that prints out a message to the console. Now add the `Target` class:

```
public class Target
{
    public virtual void Operation()
    {
        Console.WriteLine("Target.Operation() has executed.");
    }
}
```

The `Target` class is also very simple and contains a virtual method called `Operation()` that prints out a message to the console. We'll now add the `Adapter` class that wires `Target` and `Adaptee` together:

```
public class Adapter : Target
{
    private readonly Adaptee _adaptee = new Adaptee();

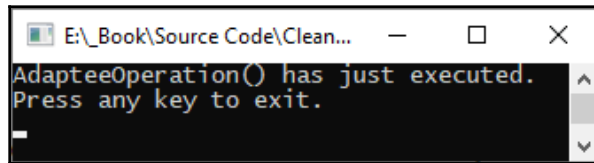
    public override void Operation()
    {
        _adaptee.AdapteeOperation();
    }
}
```

The `Adapter` class inherits the `Target` class. We then create a member variable to hold our `Adaptee` object and initialize it. We then have a single method that is the overridden `Operation()` method of the `Target` class. Finally, we will add our `Client` class:

```
public class Client
{
    public void Operation()
```

```
{  
    Target target = new Adapter();  
    target.Operation();  
}
```

The `Client` class has a single method called `Operation()`. This method creates a new `Adapter` object and assigns it to a `Target` variable. It then calls the `Operation()` method on the `Target` variable. If you call a new `Client().Operation()` method and run the code, you will see the following output:



You can see from the screenshot that the method that gets executed is the `Adaptee.AdapteeOperation()` method. Now that you have successfully learned how to implement the adapter pattern to solve oddball solutions, we will move on to look at shotgun surgery.

Shotgun surgery

Making a single change that requires changes to multiple classes is known as **shotgun surgery**. This can sometimes be down to excessive refactoring of code due to divergent changes being encountered. This code smell increases the propensity for introducing bugs such as those caused by a missed chance. You also increase the possibility of merge conflicts, because the code needs to change in so many areas that programmers end up stepping on each other's toes. The code is that convoluted that it induces cognitive overload in programmers. And new programmers have a steep learning curve because of the nature of the software.

The version control history will provide a history of the changes made to the software over time. This can help you identify all the areas that are changed, every time a new piece of functionality is added or when a bug is encountered. Once these areas have been identified, then you can look to move the changes to a more localized area of the code base. This way, when a change is required, you only have to focus on one area of the program and not many areas. This makes the maintenance of the project a lot easier.

Duplicate code is a good candidate for refactoring into a single class that is appropriately named, and that is placed in the correct namespace. Also, consider all the different layers of your application. Are they really necessary? Can things be simplified? In a database-driven application, is it really necessary to have DTOs, DAOs, domain objects, and the like? Could database access be simplified in any way? These are just some ideas for reducing the size of the code base, and so reducing the number of areas that must be modified to effect a change.

Other things to look at are the level of coupling and cohesion. Coupling needs to be kept to an absolute minimum. One way to accomplish this is to inject dependencies via constructors, properties, and methods. The injected dependencies would be of a specific interface type. We will code a simple example. Add an interface called `IService`:

```
public interface IService
{
    void Operation();
}
```

The interface contains a single method called `Operation()`. Now, add a class called `Dependency` that implements `IService`:

```
public class Dependency : IService
{
    public void Operation()
    {
        Console.WriteLine("Dependency.Operation() has executed.");
    }
}
```

The `Dependency` class implements the `IService` interface. In the `Operation()` method, a message is printed to the console. Now let's add the `LooselyCoupled` class:

```
public class LooselyCoupled
{
    private readonly IService _service;

    public LooselyCoupled(IService service)
    {
        _service = service;
    }

    public void DoWork()
    {
        _service.Operation();
    }
}
```

As you can see, the constructor takes a type of `IService` and stores it in a member variable. The call to `DoWork()` calls the `Operation()` method within the `IService` type. The `LooselyCoupled` class is just that loosely coupled, and it is easy to test.

By reducing coupling, you make classes easier to test. By removing code that does not belong in a class and placing it where it does belong, you improve the readability, maintainability, and extensibility of the application. You lessen the learning curve for anyone coming on board, and there is less chance of introducing bugs when you perform maintenance or new development.

Let's now have a look at solution sprawl.

Solution sprawl

The single responsibility that is implemented within different methods, classes, and even libraries suffer from solution sprawl. This can make code really hard to read and understand. The result is that code becomes harder to maintain and extend.

To fix the problem, move the implementation of the single responsibility into the same class. This way the code is in just one location and does what it needs to. This makes code easy to read and understand. The result is that the code can be easily maintained and extended.

Uncontrolled side effects

Uncontrolled side effects are those issues that raise their ugly heads in production because the quality assurance tests are unable to capture them. When you encounter these problems, the only option you have is to refactor the code so that it is fully testable and variables can be viewed during debugging to make sure they are set appropriately.

An example is passing values by reference. Imagine two threads passing a person object by reference to a method that modifies the person object. A side effect is that unless proper locking mechanisms are in place, each thread can modify the other thread's person object invalidating the data. You saw an example of mutable objects in [Chapter 8, Threading and Concurrency](#).

That concludes our look at application-level code smells. So, now we will move on to look at class-level code smells.

Class-level code smells

Class-level code smells are localized problems with the class in question. The kinds of problems that can plague a class are things like cyclomatic complexity and depth of inheritance, high coupling, and low cohesion. Your aim when writing a class is to keep it small and functional. The methods in the class should actually be there, and they should be small. Only do in the class what needs to be done – no more, no less. Work to remove class dependency and make your classes testable. Remove code that should be placed elsewhere to where it belongs. In this section, we address class-level code smells and how to refactor them, starting with cyclomatic complexity.

Cyclomatic complexity

When a class has a large number of branches and loops, it has an increased cyclomatic complexity. Ideally, the code should have a cyclomatic complexity value of *between 1 and 10*. Such code is simple and without risks. Code with a cyclomatic complexity of 11-20 is complex but low risk. When the cyclomatic complexity of the code is between 21-50, then the code requires attention as it is too complex and poses a medium risk to your project. And if the code has a cyclomatic complexity of more than 50, then such code is high risk and is not testable. A code that has a value above 50 must be refactored immediately.

The goal of refactoring will be to get the cyclomatic value down to between 1-10. Start by replacing `switch` statements followed by `if` expressions.

Replacing switch statements with the factory pattern

In this section, you will see how to replace a `switch` statement with the factory pattern. First, we will need a report enum:

```
[Flags]
public enum Report
{
    StaffShiftPattern,
    EndOfMonthSalaryRun,
    HrStarters,
    HrLeavers,
    EndOfMonthSalesFigures,
    YearToDateSalesFigures
}
```

The `[Flags]` attribute enables us to extract the name of the enum. The `Report` enum provides a list of reports. Now let's add our `switch` statement:

```
public void RunReport(Report report)
{
    switch (report)
    {
        case Report.EndOfMonthSalaryRun:
            Console.WriteLine("Running End of Month Salary Run Report.");
            break;
        case Report.EndOfMonthSalesFigures:
            Console.WriteLine("Running End of Month Sales Figures
Report.");
            break;
        case Report.HrLeavers:
            Console.WriteLine("Running HR Leavers Report.");
            break;
        case Report.HrStarters:
            Console.WriteLine("Running HR Starters Report.");
            break;
        case Report.StaffShiftPattern:
            Console.WriteLine("Running Staff Shift Pattern Report.");
            break;
        case Report.YearToDateSalesFigures:
            Console.WriteLine("Running Year to Date Sales Figures
Report.");
            break;
        default:
            Console.WriteLine("Report unrecognized.");
            break;
    }
}
```

Our method accepts a report and then decides on what report to execute. When I started off in 1999 as a junior VB6 programmer, I was responsible for building a report generator from scratch for the likes of Thomas Cook, ANZ, BNZ, Vodafone, and a few other big concerns. There were many reports, and I was responsible for writing a case statement that was massive that dwarfed this one. But my system worked really well. However, by today's standards, there are much better ways of performing this same code and I would do things very differently.

Let's use the factory method to run our reports without using a switch statement. Add a file called `IReportFactory` as shown:

```
public interface IReportFactory
{
    void Run();
}
```

The `IReportFactory` interface only has one method called `Run()`. This method will be used by the implementing classes to run their reports. We'll only add one report class, called `StaffShiftPatternReport`, which implements `IReportFactory`:

```
public class StaffShiftPatternReport : IReportFactory
{
    public void Run()
    {
        Console.WriteLine("Running Staff Shift Pattern Report.");
    }
}
```

The `StaffShiftPatternReport` class implements the `IReportFactory` interface. The implemented `Run()` method prints a message to the screen. Add a report called `ReportRunner`:

```
public class ReportRunner
{
    public void RunReport(Report report)
    {
        var reportName =
            $"CH13_CodeRefactoring.RefactoredCode.{report}Report,
            CH13_CodeRefactoring";
        var factory = Activator.CreateInstance(
            Type.GetType(reportName) ?? throw new
            InvalidOperationException()
        ) as IReportFactory;
        factory?.Run();
    }
}
```

The `ReportRunner` class has a method called `RunReport`. It accepts a parameter of type `Report`. With `Report` being an enum with the `[Flags]` attribute, we can obtain the name of the report enum. We use this to build the name of the report. Then, we use the `Activator` class to create an instance of the report. If the `reportName` returns null when getting the type, `InvalidOperationException` is thrown. The factory is cast to the `IReportFactory` type. We then call the `Run` method on the factory to generate the report.

This code is definitely much better than a very long `switch` statement. We need to know how to improve the readability of conditional checks within an `if` statement. We'll look at that next.

Improving the readability of conditional checks within an `if` statement

The `if` statements can break the single responsibility and the open/closed principles. See the following example:

```
public string GetHrReport(string reportName)
{
    if (reportName.Equals("Staff Joiners Report"))
        return "Staff Joiners Report";
    else if (reportName.Equals("Staff Leavers Report"))
        return "Staff Leavers Report";
    else if (reportName.Equals("Balance Sheet Report"))
        return "Balance Sheet Report";
}
```

The `GetReport()` class has three responsibilities: the staff joiners report, the staff leavers report, and the balance sheet report. This breaks the SRP because the method should only be concerned with HR reports and it is returning HR and Finance reports. As far as the open/closed principle is concerned, every time a new report is needed we will have to extend this method. Let's refactor the method so we no longer need the `if` statement. Add a new class called `ReportBase`:

```
public abstract class ReportBase
{
    public abstract void Print();
}
```

The `ReportBase` class is an abstract class with an abstract `Print()` method. We will add the `NewStartersReport` class, which inherits the `ReportBase` class:

```
internal class NewStartersReport : ReportBase
{
    public override void Print()
    {
        Console.WriteLine("Printing New Starters Report.");
    }
}
```

The `NewStartersReport` class inherits the `ReportBase` class and overrides the `Print()` method. The `Print()` method prints a message to the screen. Now, we will add the `LeaversReport` class, which is pretty much the same:

```
public class LeaversReport : ReportBase
{
    public override void Print()
    {
        Console.WriteLine("Printing Leavers Report.");
    }
}
```

The `LeaversReport` inherits the `ReportBase` class and overrides the `Print()` method. The `Print()` method prints a message to the screen. We can now call the reports as follows:

```
ReportBase newStarters = new NewStartersReport();
newStarters.Print();

ReportBase leavers = new LeaversReport();
leavers.Print();
```

Both reports inherit the `ReportBase` class, and so can be instantiated and assigned to a `ReportBase` variable. The `Print()` method can then be called on the variable, and the correct `Print()` method will be executed. The code now adheres to the single responsibility principle and the open/closed principle.

The next thing we will look at is a divergent change code smell.

Divergent change

When you need to make a change in one location and find yourself having to change many unrelated methods, then this is known as a **divergent change**. Divergent changes take place within a single class and are the result of a poor class structure. Copying and pasting code is another reason this problem arises.

To fix the problem, move the code causing the problem to its own class. If the behavior and state are shared between classes, then consider implementing inheritance using base classes and subclasses as appropriate.

The benefits of fixing divergent change-related problems include easier maintenance, as changes will be located within a single location. This makes supporting the application a whole load easier. It also removes duplicate code from the system, which just so happens to be the next thing we will be discussing.

Downcasting

When a base class is cast to one of its children, this is known as **downcasting**. This is clearly a code smell as the base class should not know about the classes that inherit it. For example, consider the `Animal` base class. Any type of animal can inherit the base class. But an animal can only be of one type. For example, felines are felines and canines are canines. It would be absurd to cast a feline to a canine and vice versa.

It is even more absurd to downcast an animal to one of its subtypes. That would be like saying a monkey is the same as a camel and is really good at transporting humans and cargo long distances through the desert. This just does not make sense. And so, you should never be downcasting. The upcasting of various animals such as monkeys and camels to the type `Animal` is valid because felines, canines, monkeys, and camels are all types of animals.

Excessive literal use

When using literals, it is very easy to introduce coding errors. An example would be a spelling mistake in a string literal. It is best to assign literals to constant variables. String literals should be placed in resource files for localization. Especially if you plan to deploy your software to different locations around the world.

Feature envy

When a method spends more time processing source code in classes other than the one that it is in, this is known as **feature envy**. We will see an example of this in our `Authorization` class. But before we do, let's have a look at our `Authentication` class:

```
public class Authentication
{
    private bool _isAuthenticated = false;

    public void Login(ICredentials credentials)
    {
        _isAuthenticated = true;
    }
}
```

```
        public void Logout()
        {
            _isAuthenticated = false;
        }

        public bool IsAuthenticated()
        {
            return _isAuthenticated;
        }
    }
```

Our `Authentication` class is responsible for logging people in and out, as well as identifying whether they are authenticated or not. Add our `Authorization` class:

```
public class Authorization
{
    private Authentication _authentication;

    public Authorization(Authentication authentication)
    {
        _authentication = authentication;
    }

    public void Login(ICredentials credentials)
    {
        _authentication.Login(credentials);
    }

    public void Logout()
    {
        _authentication.Logout();
    }

    public bool IsAuthenticated()
    {
        return _authentication.IsAuthenticated();
    }

    public bool IsAuthorized(string role)
    {
        return IsAuthenticated && role.Contains("Administrator");
    }
}
```

As you can see with our `Authorization` class, it is doing more than it is supposed to. There is one method that validates whether the user is authorized to carry a role. The role passed in is checked to see whether it is the administrator role. If it is, then the person is authorized. But if the role is not the administrator role, then the person is not authorized.

However, if you look at the other methods, they are doing no more than calling the same methods in the `Authentication` class. So, in the context of this class, the authentication methods are an example of feature envy. Let's remove the feature envy from the `Authorization` class:

```
public class Authorization
{
    private ProblemCode.Authentication _authentication;

    public Authorization(ProblemCode.Authentication authentication)
    {
        _authentication = authentication;
    }

    public bool IsAuthorized(string role)
    {
        return _authentication.IsAuthenticated() &&
            role.Contains("Administrator");
    }
}
```

You will see that the `Authorization` class is a lot smaller now, and only does what it needs to. There is no longer any feature envy.

Next up, we will look at an inappropriate intimacy code smell.

Inappropriate intimacy

A class engages in inappropriate intimacy when it relies on the implementation details held in a separate class. Does the class that has this reliance really need to exist? Can it be merged with the class that it relies on? Or is there shared functionality that is better off being extracted into its own class?

Classes should not rely on each other as this causes coupling, and it can also affect cohesion. A class should ideally be self-contained. And classes should really know as little about each other as possible.

Indecent exposure

When a class reveals its internal details, this is known as **indecent exposure**. This breaks the OOP principle of *encapsulation*. Only that which should be public should be public. All other implementations that don't need to be public should be hidden by using the appropriate access modifiers.

Data values should not be public. They should be private, and they should only be modifiable via constructors, methods, and properties. And they should only be retrievable via properties.

The large class (aka the God object)

The large class, also known as the `God` object, is all things to all parts of the system. It is a large, unwieldy class that simply does far too much. When you attempt to read the object, the intent of the code may be clear when you read the class name and see what namespace it is in, but then when you come to look at the code, the intent of the code can become lost.

A well-written class should have the name of its intent and should be placed in the appropriate namespace. The contents of the class should follow the company coding standards. Methods should be kept as small as possible, and method parameters should be kept to the absolute bare minimum. Only the methods that belong in the class should be in the class. Member variables, properties, and methods that don't belong in the class should be removed and placed in the correct files in the correct namespace.

To keep classes small and focused, don't inherit classes if there is no need. If there is a class that has five methods, and you will only ever use one of them, is it possible to move that method out into its own reusable class? Remember the single responsibility principle. A class should only have a single responsibility. For example, a file class should only handle operations and behaviors associated with files. A file class should not be performing database operations. You get the idea.

When writing a class, your aim is to make it as small, clean, and readable as you can.

The lazy class (aka the freeloader and the lazy object)

A **freeloading** class is one that hardly does anything to be useful. When you encounter such classes, you can merge their contents with other classes that have the same kind of intentions.

You can also attempt to collapse the inheritance hierarchy. Remember that the ideal depth of inheritance is 1. And so, if your classes have a larger value for their depth of inheritance, then they are good candidates for moving back up the inheritance tree. You may also want to consider using inline classes for really small classes.

The middleman class

The middleman class does no more than delegate functionality to other objects. In situations like this, you can get rid of the middleman and deal with the objects that carry out the responsibility directly.

Also, remember that you need to keep the depth of inheritance down. So if you cannot get rid of the class, look to merge it with existing classes. Look at the overall design of that area of code. Could it all be refactored in some way to reduce the amount of code and the number of different classes?

The orphan class of variables and constants

It is not really good practice to have a lone class that holds variables and constants for multiple different parts of the application. When you encounter such a situation, it can be hard for the variables to have any real meaning and their context can be lost. It is better to move constants and variables to areas that use them. If constants and variables will be used by multiple classes, then they should be assigned to a file within the root of the namespace they will be used in.

Primitive obsession

Source code that uses primitive values rather than objects for certain tasks such as range values and formatted strings such as credit cards, postcodes, and phone numbers suffers from primitive obsession. Other signs include constants used for field names, and information stored inappropriately stored in constants.

Refused bequest

When a class inherits from another class but does not use all its methods, then this is known as **refused bequest**. A common reason for this happening is when the subclass is completely different from the base class. For example, a `building` base class is used by different building types, but then a `car` object inherits `building` because it has properties and methods to do with windows and doors. This is clearly wrong.

When you encounter this, consider whether a base class is necessary. If it is, then create one and then inherit from it. Otherwise, add the functionality to the class that was inherited from the wrong type.

Speculative generality

A class that is programmed with functionality that is not needed now but may be needed in the future is suffering from speculative generality. Such code is dead code and adds maintenance overhead as well as code bloat. It is best to remove these classes when you see them.

Tell, Don't Ask

The *Tell, Don't Ask* software principle informs us as programmers that we are to bundle data with the methods that will operate on that data. Our objects must not ask for data and then operate on it! They must tell the logic of an object to perform a specific task on that object's data.

If you find objects that contain logic and that ask other objects for data to carry out their operations, then combine the logic and the data into a single class.

Temporary fields

Temporary fields are member variables that are not needed for an object's entire lifetime.

You can perform refactoring by removing the temporary fields and the methods that operate upon them to their own class. You will end up with clearer code that is well organized.

Method-level smells

Method-level code smells are problems within the method itself. Methods are the work-horses that either make software function well or poorly. They should be well organized and do only what they are expected to do—no more and no less. It is important to know the kinds of problems and issues that can arise because of poorly constructed methods. We will address what to look out for in terms of method-level code smells, and what we can do to address them. We'll start with the black sheep method first.

The black sheep method

Out of all the methods in the class, a black sheep method will be noticeably different. When you encounter a black sheep method, you must consider the method objectively. What is its name? What is the method's intent? When you have answered these questions, then you can decide to remove the method and place it where it truly belongs.

Cyclomatic complexity

When a method has too many loops and branches, this is known as cyclomatic complexity. This code smell is also a class-level code smell, and we have already seen how we can reduce the problems with branching when we looked at replacing `switch` and `if` statements. As for loops, they can be replaced with LINQ statements. LINQ statements have the added benefit of being a functional code since LINQ is a functional query language.

Contrived complexity

When a method is unnecessarily complex and can be simplified, this complexity is termed contrived complexity. Simplify the method to make sure that its contents are human-readable and understandable. Then, look to refactor the method and reduce the size to the smallest number of lines that is practical.

Dead code

When a method exists but is not used, this is known as dead code. The same goes for constructors, properties, parameters, and variables. They should be identified and removed.

Excessive data return

When a method returns more data than is needed by each client that calls it, this code smell is known as excessive data return. Only the data that is required should be returned. If you find that there are groups of objects with different requirements, then you should maybe consider writing different methods that appeal to both groups and only return what is necessary to those groups.

Feature envy

A method that has feature envy spends more time accessing data in other objects than it does in its own object. We have already seen this in action when we looked at feature envy under class-level code smells.

A method should be kept small, and most of all, its main functionality should be localized to that method. If it is doing more in other methods than its own, then there is scope for moving some of the code out of the method and into its own method.

Identifier size

Identifiers can be either too short or too long. Identifiers should be descriptive and succinct. The main thing to consider when naming variables is the context and location. In a localized loop, a single letter may be appropriate. But if the identifier is at the class level, then it will need a human-understandable name to give it context. Avoid using names that lack context, and that are ambiguous or cause confusion.

Inappropriate intimacy

Methods that rely too heavily on implementation details in other methods or classes display inappropriate intimacy. These methods need to be refactored and possibly even removed. The main thing to bear in mind is that the methods use the internal fields and methods of another class.

To perform refactoring, you can move the methods and fields to where they actually need to be used. Alternatively, you can extract the fields and methods into a class of their own. Inheritance can replace delegation when the subclass is being intimate with the superclass.

Long lines (aka God lines)

Long lines of code can be very hard to read and decipher. This makes it difficult for programmers to debug and refactor such code. Where it is possible, the line can be formatted so that any periods and any code after a comma appears on a new line. But such code should also be refactored to make it small.

Lazy methods

A lazy method is one that does very little work. It may delegate its work to other methods, and it may simply call a method on another class that does what it is supposed to. If any of these are the case, then it may pay to get rid of the methods and place code within the methods where it is needed. You could, for instance, use an inline function such as a lambda.

Long methods (aka God methods)

A long method is one that has outgrown itself. Such methods may lose their intent and perform more tasks than they are expected to. You can use the IDE to select parts of the method, and then select extract method or extract class to move portions of the method to their own method and even their own class. A method should only be responsible for doing a single task.

Long parameter lists (aka too many parameters)

Three or more parameters are classed as the long parameter list code smell. You can tackle this problem by replacing the parameters with a method call. An alternative is to replace the parameters with a parameter object.

Message chains

A message chain occurs when a method calls an object that calls another object that calls another object and so on. Previously, you saw how to deal with message chains when we looked at the Law of Demeter. Message chains break this law, as a class should only communicate with its nearest neighbor. Refactor the classes to move the required state and behavior closer to where it is needed.

The middleman method

When all a method does is delegate work out to others to complete, it is a middleman method and can be refactored and removed. But if there is functionality that can't be removed, then merge it in the area that it is being used in.

Oddball solutions

When you see multiple methods doing the same thing but doing it differently, then this is an oddball solution. Choose the method that best implements the task, and then replace the method calls to the other methods with calls to the best method. Then, delete the other methods. This will leave only one method and one way of implementing the task that can be reused.

Speculative generality

A method that is not used anywhere in the code is known as a speculative generality code smell. It is essentially dead code, and all dead code should be removed from the system. Such code provides a maintenance overhead and also provides unnecessary code bloat.

Summary

In this chapter, you have been introduced to a variety of code smells and how to remove them through refactoring. We have stated that there are application-level code smells that permeate throughout all the layers of the application, class-level code smells that run throughout the class, and method-level code smells that affect the individual methods.

First of all, we covered the application-level code smells, which consisted of Boolean blindness, combinatorial explosion, contrived complexity, data clump, deodorant comments, duplicate code, lost intent, mutation of variables, oddball solutions, shotgun surgery, solution sprawl, and uncontrolled side effects.

We then went on to look at class-level code smells, including cyclomatic complexity, divergent change, downcasting, excessive literal use, feature envy, inappropriate intimacy, indecent exposure, and the large object, also known as the God object. We also covered the lazy class, also known as the freeloader and the lazy object; middleman; orphan classes of variables and constants; primitive obsession; refused bequest; speculative generality; Tell, Don't Ask; and temporary fields.

Finally, we moved on to method-level code smells. We discussed black sheep; cyclomatic complexity; contrived complexity; dead code; feature envy; identifier size; inappropriate intimacy; long lines, also known as the God lines; the lazy method; the long method, also known as the God method; the long parameter list, also known as too many parameters; message chains; middleman; oddball solutions; and speculative generality.

In the next chapter, we will be continuing our look at code refactoring with the use of ReSharper.

Questions

1. What are the three main categories of code smell?
2. Name the different types of application-level code smells.
3. Name the different types of class-level code smells.
4. Name the different types of method-level code smells.
5. What kinds of refactoring are you able to perform in order to clean up various code smells?
6. What is cyclomatic complexity?
7. How can we overcome cyclomatic complexity?
8. What is contrived complexity?
9. How can we overcome contrived complexity?
10. What is a combinatorial explosion?
11. How do we overcome a combinatorial explosion?
12. What should you do when you find deodorant comments?
13. If you have bad code but don't know how to fix it, what should you do?
14. What is a good place to ask questions and get answers when it comes to programming issues?
15. In what ways can a long parameter list be reduced?
16. How can a large method be refactored?
17. What is the maximum length for a clean method?
18. Within what range of numbers should your program's cyclomatic complexity be?
19. What is the ideal depth of inheritance value?
20. What is speculative generality and what should you do about it?
21. If you encounter an oddball solution, what course of action should you take?
22. What refactorings would you perform if you encountered a temporary field?

23. What is a data clump, and what should you do about it?
24. Explain the refused bequest code smell.
25. What law do message chains break?
26. How should message chains be refactored?
27. What is feature envy?
28. How do you remove feature envy?
29. What pattern can you use to replace `switch` statements that return objects?
30. How can we replace `if` statements that return objects?
31. What is solution sprawl, and what can be done to tackle it?
32. Explain the Tell, don't ask! principle.
33. How does the Tell, don't ask! principle get broken?
34. What are the symptoms of shotgun surgery, and how should they be addressed?
35. Explain lost intent and what can be done about it.
36. How can loops be refactored, and what benefits do the refactorings bring?
37. What is a divergent change, and how would you go about refactoring it?

Further reading

- *Refactoring - Improving the Design of Existing Code* by Martin Fowler and Kent Beck.
- <https://refactoring.guru/refactoring>: A good site on design patterns and code smells.
- <https://www.dofactory.com/net/design-patterns>: A very good C#-based site on various design patterns.

14

Refactoring C# Code – Implementing Design Patterns

Half the battle in programming clean code is in the correct implementation and usage of design patterns. Design patterns themselves can become code smells. A design pattern becomes a code smell when it is used to over-engineer something that is rather simple to implement.

You have already seen the use of design patterns in writing clean code and refactoring code smells in the previous chapters of this book. Specifically, we have implemented the adapter pattern, the decorator pattern, and the proxy pattern. These patterns were implemented in the right way to accomplish the task at hand. They were kept simple and they most certainly did not complicate the code. So, when used for their proper purpose, design patterns are really useful in removing code smells, thus leaving your code nice, clean, and fresh.

In this chapter, we will address the **Gang of Four (GoF)** creational, structural, and behavioral design patterns. Design patterns are not set in stone and you don't have to be strict in their implementation. But having code samples can help you transition from just having head knowledge to having the practical skills needed to correctly implement and use design patterns.

In this chapter, we will be covering the following topics:

- Implementing creational design patterns
- Implementing structural design patterns
- Overview of behavioral design patterns

By the end of this chapter, you will have the following skills:

- The ability to understand, describe, and program different creational design patterns
- The ability to understand, describe, and program different structural design patterns
- An understanding of an overview of behavioral design patterns

We will begin our overview of GoF design patterns by addressing creational design patterns.

Technical requirements

- Visual Studio 2019
- A Visual Studio 2019 .NET Framework console application as your working project
- The complete source code for this chapter: https://github.com/PacktPublishing/Clean-Code-in-C-/tree/master/CH14/CH14_DesignPatterns

Implementing creational design patterns

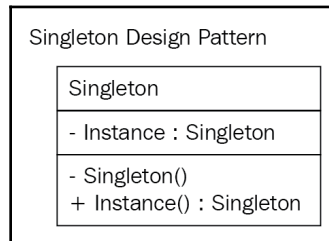
From a programmer's perspective, we use creational design patterns when we perform object creation. Patterns are selected based on the task at hand. There are five creational design patterns:

- **Singleton:** The singleton pattern ensures that only one instance of an object will exist at the application level.
- **Factory method:** A factory pattern is used to create objects without using the class to be used.
- **Abstract factory:** Without the specification of their concrete classes, groups of related or dependent objects are instantiated by the abstract factory.
- **Prototype:** Specifies the type of prototype to create, and then creates copies of the prototype.
- **Builder:** Separates object construction from its representation.

We will now begin implementing each of these patterns, starting with the singleton design pattern.

Implementing the singleton pattern

The singleton design pattern only allows one instance of a class with global access to it. Use the singleton pattern when all operations within a system must be coordinated by exactly one object:



The participant in this pattern is **singleton**—a class that is responsible for managing its own instance and ensures that there is only one instance of itself running in the entire system.

We are now going to implement the singleton design pattern:

1. Add a folder called **Singleton** to the **CreationalDesignPatterns** folder. Then, add a class called **Singleton**:

```

public class Singleton {
    private static Singleton _instance;

    protected Singleton() { }

    public static Singleton Instance() {
        return _instance ?? (_instance = new Singleton());
    }
}
  
```

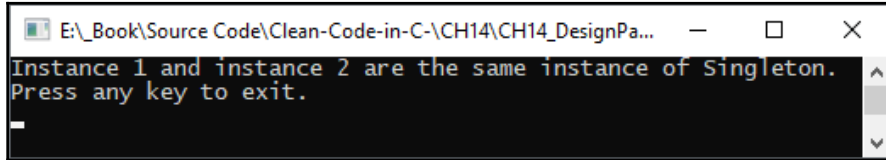
2. The **Singleton** class stores a static copy of an instance of itself. You cannot instantiate the class because the constructor is marked as protected. The **Instance()** method is static. It checks to see whether an instance of the **Singleton** class exists. If it does, then it is returned. If it does not exist, then the instance is created and returned. Now, we'll add the code to call it:

```

var instance1 = Singleton.Instance();
var instance2 = Singleton.Instance();

if (instance1.Equals(instance2))
    Console.WriteLine("Instance 1 and instance 2 are the same
instance of Singleton.");
  
```

3. We declare two instances of the `Singleton` class, and then compare them to see whether they are the same instance. You can see the output in the following screenshot:

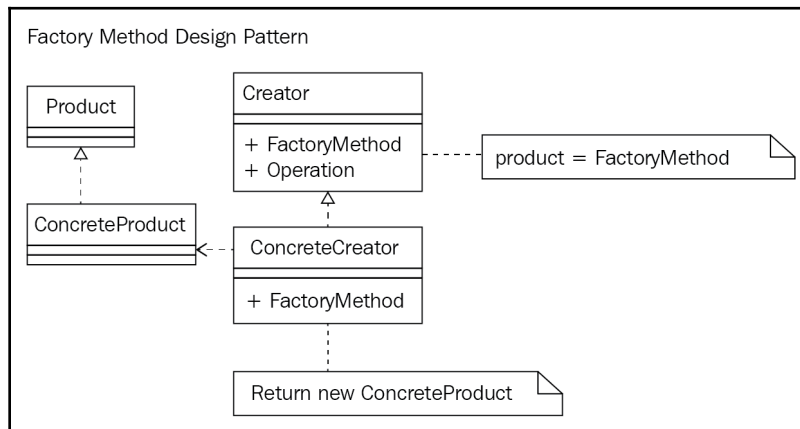


```
E:\_Book\Source Code\Clean-Code-in-C-\CH14\CH14_DesignPa...
Instance 1 and instance 2 are the same instance of Singleton.
Press any key to exit.
```

As you can see, we have a working class that implements the singleton design pattern. Next up, we'll tackle the factory method design pattern.

Implementing the factory method pattern

The factory method design pattern creates objects that let their subclasses implement their own object creation logic. Use this design pattern when you want to keep object instantiation in a single place and need to generate a specific group of related objects:



The participants in this project are as follows:

- **Product:** The abstract product created by the factory method
- **ConcreteProduct:** Inherits the abstract product
- **Creator:** An abstract class with an abstract factory method
- **Concrete Creator:** Inherits the abstract creator and overrides the factory method

We will now implement the factory method:

1. Add a folder to the `CreationalDesignPatterns` folder called `FactoryMethod`. Then, add the `Product` class:

```
public abstract class Product {}
```

2. The `Product` class defines the objects that are created by the factory method. Add the `ConcreteProduct` class:

```
public class ConcreteProduct : Product {}
```

3. The `ConcreteProduct` class inherits the `Product` class. Add the `Creator` class:

```
public abstract class Creator {  
    public abstract Product FactoryMethod();  
}
```

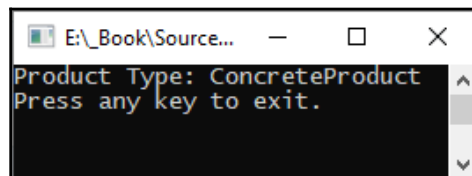
4. The `Creator` class will be inherited by the `ConcreteFactory` class, which will implement `FactoryMethod()`. Add the `ConcreteCreator` class:

```
public class ConcreteCreator : Creator {  
    public override Product FactoryMethod() {  
        return new ConcreteProduct();  
    }  
}
```

5. The `ConcreteCreator` class inherits the `Creator` class and overrides the `FactoryMethod()`. A new `ConcreteProduct` class is returned by the method. The following code demonstrates the factory method in use:

```
var creator = new ConcreteCreator();  
var product = creator.FactoryMethod();  
Console.WriteLine($"Product Type: {product.GetType().Name}");
```

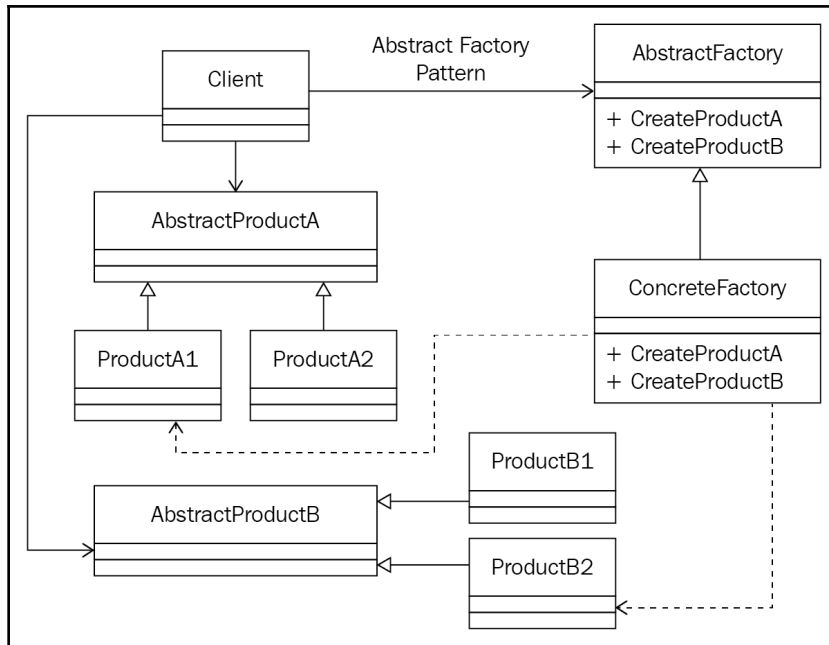
We have created a new instance of the `ConcreteCreator` class. Then, we called the `FactoryMethod()` to create a new product. The name of the product created by the factory method is then output to the console window, as shown:



Now that we know how to implement the factory method design pattern, we will move on to implementing the abstract factory design pattern.

Implementing the abstract factory pattern

Without the specification of their concrete classes, groups of related or dependent objects, referred to as families, are instantiated using the abstract factory design pattern:



The participants in this pattern are as follows:

- **AbstractFactory**: The abstract factory, which is implemented by concrete factories
- **ConcreteFactory**: Creates concrete products
- **AbstractProduct**: The abstract product that concrete products will inherit
- **Product**: Inherits **AbstractProduct** and is created by the concrete factory

We will now start implementing the pattern:

1. Add a folder to the project called `CreationalDesignPatterns`.
2. Add a folder to the `CreationalDesignPatterns` folder called `AbstractFactory`.
3. In the `AbstractFactory` folder, add the `AbstractFactory` class:

```
public abstract class AbstractFactory {  
    public abstract AbstractProductA CreateProductA();  
    public abstract AbstractProductB CreateProductB();  
}
```

4. `AbstractFactory` contains two abstract methods for creating abstract products. Add the `AbstractProductA` class:

```
public abstract class AbstractProductA {  
    public abstract void Operation(AbstractProductB productB);  
}
```

5. The `AbstractProductA` class has a single abstract method, which performs an operation on `AbstractProductB`. Now, add the `AbstractProductB` class:

```
public abstract class AbstractProductB {  
    public abstract void Operation(AbstractProductA productA);  
}
```

6. The `AbstractProductB` class has a single abstract method, which performs an operation on `AbstractProductA`. Add the `ProductA` class:

```
public class ProductA : AbstractProductA {  
    public override void Operation(AbstractProductB productB) {  
        Console.WriteLine("ProductA.Operation(ProductB)");  
    }  
}
```

7. `ProductA` inherits `AbstractProductA` and overrides the `Operation()` method, which interacts with `AbstractProductB`. The `Operation()` method in this example prints out a console message. Do the same for the `ProductB` class:

```
public class ProductB : AbstractProductB {  
    public override void Operation(AbstractProductA productA) {  
        Console.WriteLine("ProductB.Operation(ProductA)");  
    }  
}
```

8. **ProductB inherits AbstractProductB and overrides the Operation() method, which interacts with AbstractProductA. The Operation() method in this example prints out a console message. Add the ConcreteFactory class:**

```
public class ConcreteProduct : AbstractFactory {
    public override AbstractProductA CreateProductA() {
        return new ProductA();
    }

    public override AbstractProductB CreateProductB() {
        return new ProductB();
    }
}
```

9. **ConcreteFactory inherits the AbstractFactory class and overrides the two product creation methods. Each method returns a concrete class. Add the Client class:**

```
public class Client
{
    private readonly AbstractProductA _abstractProductA;
    private readonly AbstractProductB _abstractProductB;

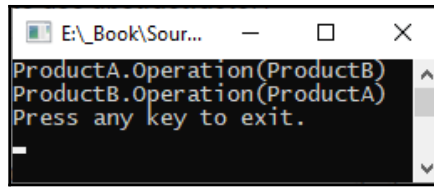
    public Client(AbstractFactory factory) {
        _abstractProductA = factory.CreateProductA();
        _abstractProductB = factory.CreateProductB();
    }

    public void Run() {
        _abstractProductA.Operation(_abstractProductB);
        _abstractProductB.Operation(_abstractProductA);
    }
}
```

10. **The Client class declares two abstract products. Its constructor takes an AbstractFactory class. Inside the constructor, both declared abstract products are assigned their respective concrete products by the factory. The Run() method executes Operation() on both products. The following code executes our abstract factory example:**

```
AbstractFactory factory = new ConcreteProduct();
Client client = new Client(factory);
client.Run();
```

11. **Run the code and you will see the following output:**



```

E:\_Book\Sour...
ProductA.Operation(ProductB)
ProductB.Operation(ProductA)
Press any key to exit.

```



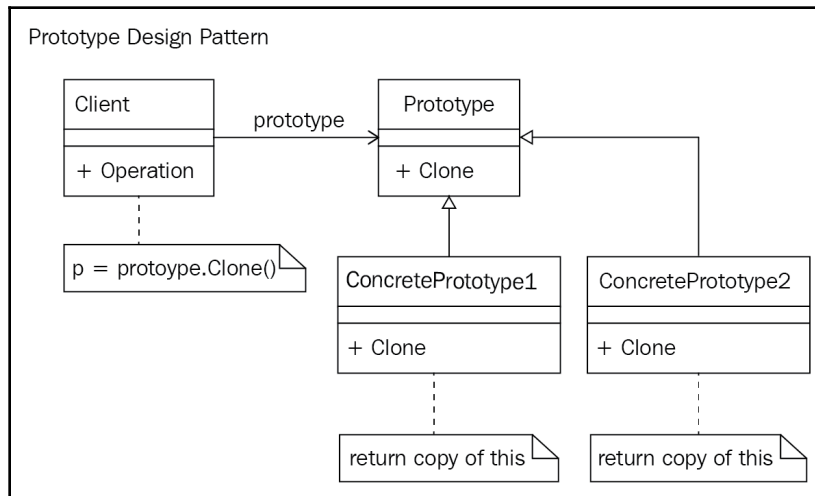
A good reference implementation of the abstract factory is the ADO.NET 2.0 `DbProviderFactory` abstract class. An article called *Abstract Factory Design Pattern in ADO.NET 2.0* by Moses Soliman on C# Corner is a nice write-up on `DbProviderFactory` about the implementation of the abstract factory design pattern. Here is the link:

<https://www.c-sharpcorner.com/article/abstract-factory-design-pattern-in-ado-net-2-0/>.

We have successfully implemented the abstract factory design pattern. Now, we will implement the prototype pattern.

Implementing the prototype pattern

The prototype design pattern is used to create an instance of a prototype, and then to create new objects by cloning the prototype. Use this pattern when the cost of creating objects directly is expensive. With this pattern, you can cache the object and return a clone when needed:



The participants in the prototype design pattern are as follows:

- **Prototype:** An abstract class that provides a method for cloning itself
- **ConcretePrototype:** Inherits the prototype and overrides the `Clone()` method to return a memberwise clone of the prototype
- **Client:** Requests new clones of the prototype

We will now implement the prototype design pattern:

1. Add a folder called `Prototype` to the `CreationalDesignPatterns` folder, and then add the `Prototype` class:

```
public abstract class Prototype {
    public string Id { get; private set; }

    public Prototype(string id) {
        Id = id;
    }

    public abstract Prototype Clone();
}
```

2. Our `Prototype` class must be inherited. Its constructor requires an identifying string to be passed in that is stored at the class level. A `Clone()` method is provided, which the subclass will override. Now, add the `ConcretePrototype` class:

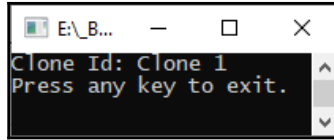
```
public class ConcretePrototype : Prototype {
    public ConcretePrototype(string id) : base(id) { }

    public override Prototype Clone() {
        return (Prototype) this.MemberwiseClone();
    }
}
```

3. The `ConcretePrototype` class inherits from the `Prototype` class. Its constructor takes an identifying string and passes that string into the constructor of the base class. It then overrides the clone method to provide a shallow copy of the current object by calling the `MemberwiseClone()` method and returning the clone that is cast to the type of `Prototype`. Now for the code that demonstrates the prototype design pattern in use:

```
var prototype = new ConcretePrototype("Clone 1");
var clone = (ConcretePrototype)prototype.Clone();
Console.WriteLine($"Clone Id: {clone.Id}");
```

Our code creates a new instance of the `ConcretePrototype` class with an identifier of "Clone 1". We then clone the prototype and cast it to the `ConcretePrototype` type. Then, we print the clone's identifier to the console window, as shown:



As we can see, the clone has the same identifier as the prototype that it was cloned from.

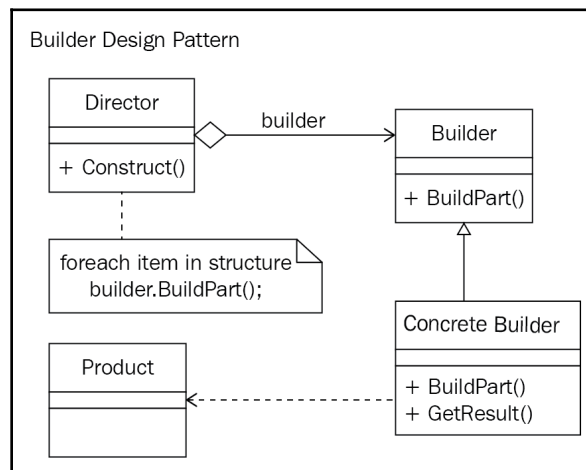


For a very detailed article of a real-world example, refer to an excellent article called *Prototype Design Pattern with Real-World Scenario*, by Akshay Patel, on C# Corner. Here is the link: <https://www.c-sharpcorner.com/UploadFile/db2972/prototype-design-pattern-with-real-world-scenario624/>.

We will now implement our final creational design pattern called the builder design pattern.

Implementing the builder pattern

The builder design pattern separates the object's construction from its representation. As a result, you can use the same construction method to create different representations of the object. Use the builder design pattern when you have a complex object that needs to be built up and connected in stages:



The participants in the builder design pattern are as follows:

- **Director:** A class that receives a builder via its constructor, and then calls each of the build methods on the builder object
- **Builder:** An abstract class that provides abstract build methods and an abstract method for returning the built object
- **ConcreteBuilder:** A concrete class that inherits the Builder class, overrides the builder methods to actually build the object, and overrides the result method to return the fully built object

Let's start implementing our final creational design pattern—the builder design pattern:

1. Start by adding a folder called `Builder` to the `CreationalDesignPatterns` folder. Then, add the `Product` class:

```
public class Product {
    private List<string> _parts;

    public Product() {
        _parts = new List<string>();
    }

    public void Add(string part) {
        _parts.Add(part);
    }

    public void PrintPartsList() {
        var sb = new StringBuilder();
        sb.AppendLine("Parts Listing:");
        foreach (var part in _parts)
            sb.AppendLine($"- {part}");
        Console.WriteLine(sb.ToString());
    }
}
```

2. In our example, the `Product` class keeps a list of parts. These parts are strings. The list is initialized in the constructor. Parts are added by the `Add()` method, and when our object is fully constructed, we can call the `PrintPartsList()` method to print the list of parts that make up the object to the console window. Now, add the `Builder` class:

```
public abstract class Builder
{
    public abstract void BuildSection1();
    public abstract void BuildSection2();
}
```

```
        public abstract Product GetProduct();  
    }  
}
```

3. Our `Builder` class will be inherited by concrete classes that will override its abstract methods to build the object and return it. We'll now add the `ConcreteBuilder` class:

```
public class ConcreteBuilder : Builder {  
    private Product _product;  
  
    public ConcreteBuilder() {  
        _product = new Product();  
    }  
  
    public override void BuildSection1() {  
        _product.Add("Section 1");  
    }  
  
    public override void BuildSection2() {  
        _product.Add("Section 2");  
    }  
  
    public override Product GetProduct() {  
        return _product;  
    }  
}
```

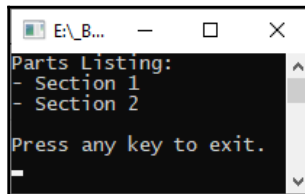
4. Our `ConcreteBuilder` class inherits the `Builder` class. The class stores the instance of the object to be constructed. The build methods are overridden and parts are added to the product via the product's `Add()` method. The product is returned to the client via the `GetProduct()` method call. Add the `Director` class:

```
public class Director  
{  
    public void Build(Builder builder)  
    {  
        builder.BuildSection1();  
        builder.BuildSection2();  
    }  
}
```

5. The `Director` class is a concrete class that takes a `Builder` object via its `Build()` method and calls the build methods on the `Builder` object to build the object. All we need now is the code to demonstrate the builder design pattern in action:

```
var director = new Director();  
var builder = new ConcreteBuilder();  
director.Build(builder);  
var product = builder.GetProduct();  
product.PrintPartsList();
```

6. We create a director and builder. Then, the director builds the product. The product is then assigned, and its parts list is printed out to the console window, as shown:



Everything is working as it should be.

In .NET Framework, the `System.Text.StringBuilder` class is an example of the builder design pattern in the real world. Using string concatenation with the plus (+) operator is slower than using the `StringBuilder` class when concatenating five or more lines. String concatenation with the + operator is faster than `StringBuilder` when you have less than five concatenation lines, but slower when you have more than five lines to concatenate. The reason for this is that each time you create a string with the + operator, you are recreating the string since strings are immutable on the heap. But `StringBuilder` allocates buffer space on the heap. Then, characters are written to the buffer space. For only a small number of lines, the + operator is faster because of the overhead of creating the buffer when using the string builder. But when there are more than five lines, there is a noticeable difference when using `StringBuilder`. In big data projects where there may be hundreds of thousands or even millions of string concatenations taking place, the string concatenation strategy that you decide to employ will either perform fast or sluggishly. Let's create a simple demonstration. Create a new class called `StringConcatenation`, and then add the following code:

```
private static DateTime _startTime;  
private static long _durationPlus;  
private static long _durationSb;
```

The `_startTime` variable holds the current start time of the method execution. The `_durationPlus` variable holds the duration of the method execution as the number of ticks when using the `+` operator to concatenate, and `_durationSb` holds the duration of the operation as the number of ticks for the `StringBuilder` concatenation. Add the `UsingThePlusOperator()` method to the class:

```
public static void UsingThePlusOperator()
{
    _startTime = DateTime.Now;
    var text = string.Empty;
    for (var x = 1; x <= 10000; x++)
    {
        text += $"Line: {x}, I must not be a lazy programmer, and should
continually develop myself!\n";
    }
    _durationPlus = (DateTime.Now - _startTime).Ticks;
    Console.WriteLine($"Duration (Ticks) Using Plus Operator:
{_durationPlus}");
}
```

The `UsingThePlusOperator()` method demonstrates the time taken when concatenating 10,000 strings using the `+` operator. The time taken to process the string concatenation is stored as the number of ticks fired. There are 10,000 ticks per millisecond. Now, add the `UsingTheStringBuilder()` method:

```
public static void UsingTheStringBuilder()
{
    _startTime = DateTime.Now;
    var sb = new StringBuilder();
    for (var x = 1; x <= 10000; x++)
    {
        sb.AppendLine(
            $"Line: {x}, I must not be a lazy programmer, and should
continually develop myself!"
        );
    }
    _durationSb = (DateTime.Now - _startTime).Ticks;
    Console.WriteLine($"Duration (Ticks) Using StringBuilder:
{_durationSb}");
}
```

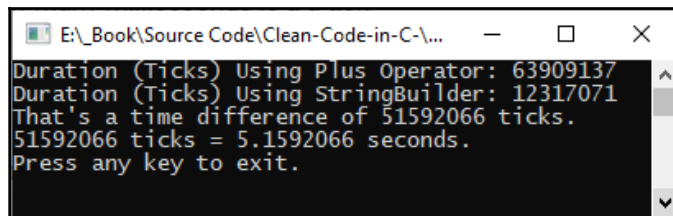
This method is the same as the previous one, except we perform string concatenation using the `StringBuilder` class. We'll now add the code to print out the time difference, called `PrintTimeDifference()`:

```
public static void PrintTimeDifference()
{
    var difference = _durationPlus - _durationSb;
    Console.WriteLine($"That's a time difference of {difference} ticks.");
    Console.WriteLine($"{difference} ticks =
{TimeSpan.FromTicks(difference)} seconds.\n\n");
}
```

The `PrintTimeDifference()` method calculates the time difference by subtracting the `StringBuilder` ticks from the `+` ticks. The difference in ticks is then printed to the console, followed by a line that translates the ticks into seconds. Here is the code to test our methods so that we can see the time difference in the two concatenation methods:

```
StringConcatenation.UsingThePlusOperator();
StringConcatenation.UsingTheStringBuilder();
StringConcatenation.PrintTimeDifference();
```

When you run the code, you will see the times and time difference in the console window, as shown:



As you can see from the screenshot, `StringBuilder` is much faster. With small amounts of data, you don't really see a difference with the naked eye. But the difference is noticeable to the naked eye when the data lines being processed greatly increase in number.

Another example that comes to mind for using the builder pattern is report construction. If you consider banded reports, the bands are essentially sections that need to be built up from various sources. So, you could have the main part, and then each subreport as a different part. The final report would be the amalgamation of these various parts. So, you could have code like the following to build a report:

```
var report = new Report();
report.AddHeader();
report.AddLastYearsSalesTotalsForAllRegions();
```

```
report.AddLastYearsSalesTotalsByRegion();  
report.AddFooter();  
report.GenerateOutput();
```

Here, we are creating a new report. We start by adding the header. Then, we add last year's sales figures, combined for all regions, followed by last year's sales figures, broken down by region. We then add a footer to the report and complete the process by generating the report output.

So, you've seen the default implementation of the builder pattern from the UML diagram. Then, you implemented string concatenation using the `StringBuilder` class, which helps you build strings in a performant manner. Finally, you learned how the builder pattern can be useful in building up the sections of a report and generating its output.

Well, that concludes our implementations of the creational design patterns. We will now move on to implementing some structural design patterns.

Implementing structural design patterns

As programmers, we use structural patterns to improve the overall structure of our code. So, when code is encountered that lacks structure and is not at its cleanest, we can use the patterns mentioned in this section to restructure the code and make it clean. There are seven structural design patterns:

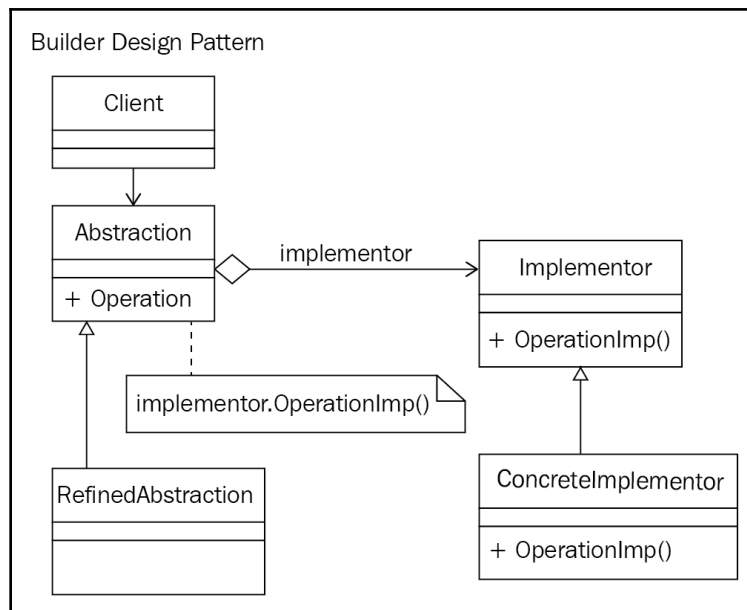
- **Adapter:** Use this pattern to enable classes with incompatible interfaces to work cleanly together.
- **Bridge:** Use this pattern to loosely couple code by decoupling an abstraction from its implementation.
- **Composite:** Use this pattern to aggregate objects and provide a uniform way of working with individual and object compositions.
- **Decorator:** Use this pattern to keep the interface the same while dynamically adding new functionality to the object.
- **Facade:** Use this pattern to simplify larger and more complex interfaces.
- **Flyweight:** Use this pattern to conserve memory and pass shared data between objects.
- **Proxy:** Use this pattern between a client and an API to intercept calls between the client and the API.

We have already touched on the adapter, decorator, and proxy patterns in previous chapters, so they won't be covered again in this chapter. Now, we'll start implementing our structural design patterns, starting with the bridge pattern.

Implementing the bridge pattern

We use the bridge pattern to decouple abstractions from their implementations so that they are not bound at compile time. Both the abstraction and implementation can vary without impacting the client.

Use the bridge design pattern if you require runtime binding of the implementation or sharing of the implementation between multiple objects, if a number of classes exist as a result of interface coupling and various implementations, or if there is a need for orthogonal class hierarchies to be mapped:



The participants of the bridge design pattern are as follows:

- **Abstraction**: An abstract class that contains abstract operations
- **RefinedAbstraction**: Inherits the **Abstraction** class and overrides the `Operation()` method

- **Implementor:** An abstract class with an abstract `Operation()` method
- **ConcreteImplementor:** Inherits the `Implementor` class and overrides the `Operation()` method

We will now implement the bridge design pattern:

1. Start by adding the `StructuralDesignPatterns` folder to the project, and then in that folder, add the `Bridge` folder. Then, add the `Implementor` class:

```
public abstract class Implementor {  
    public abstract void Operation();  
}
```

2. The `Implementor` class has just a single abstract method, called `Operation()`. Add the `Abstraction` class:

```
public class Abstraction {  
    protected Implementor implementor;  
  
    public Implementor Implementor {  
        set => implementor = value;  
    }  
  
    public virtual void Operation() {  
        implementor.Operation();  
    }  
}
```

3. The `Abstraction` class has a protected field that holds the `Implementor` object, which is set via the `Implementor` property. A virtual method called `Operation()` calls the `Operation()` method on the `implementor`. Add the `RefinedAbstraction` class:

```
public class RefinedAbstraction : Abstraction {  
    public override void Operation() {  
        implementor.Operation();  
    }  
}
```

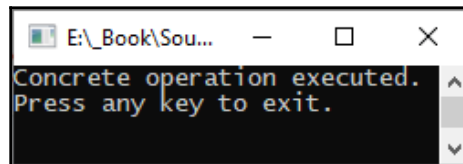
4. The `RefinedAbstraction` class inherits the `Abstraction` class and overrides the `Operation()` method to call the `Operation()` method on the implementor. Now, add the `ConcreteImplementor` class:

```
public class ConcreteImplementor : Implementor {  
    public override void Operation() {  
        Console.WriteLine("Concrete operation executed.");  
    }  
}
```

5. The `ConcreteImplementor` class inherits the `Implementor` class and overrides the `Operation()` method to print out a message to the console. The code to run the bridge design pattern example is as follows:

```
var abstraction = new RefinedAbstraction();  
abstraction.Implementor = new ConcreteImplementor();  
abstraction.Operation();
```

We create a new `RefinedAbstraction` instance and then set its implementor to a new instance of `ConcreteImplementor`. Then, we call the `Operation()` method. The output from our example bridge implementation is as follows:

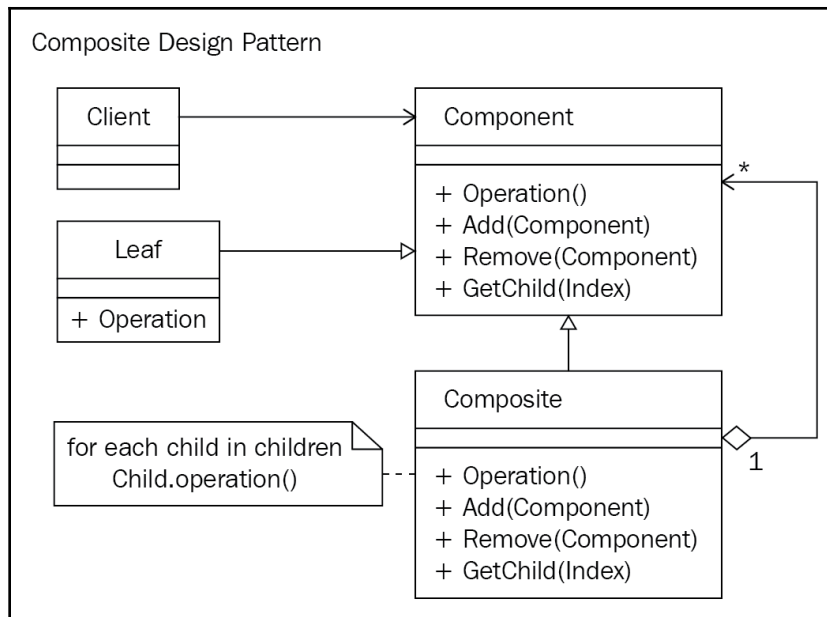


As you can see, we successfully executed the concrete operation in the concrete implementor class. The next pattern we will look at is the composite design pattern.

Implementing the composite pattern

With the composite design pattern, objects are composed of tree structures to represent part-whole hierarchies. This pattern enables you to treat individual objects and compositions of objects in a uniform manner.

Use this pattern when you need to ignore the differences between individual objects and object compositions, when you need tree structures to represent hierarchies, and when a hierarchical structure requires generic functionality across its structure:



The participants in the composite design pattern are as follows:

- **Component:** Composed objects interface
- **Leaf:** A leaf in the composition that has no children
- **Composite:** Stores child components and performs operations
- **Client:** Manipulates compositions and leaves via the component interface

It's time to implement the composite pattern:

1. Add a new folder called `Composite` to the `StructuralDesignPatterns` class. Then, add the `IComponent` interface:

```
public interface IComponent {
    void PrintName();
}
```

2. The `IComponent` interface has a single method, which will be implemented by both leaves and composites. Add the `Leaf` class:

```
public class Leaf : IComponent {
    private readonly string _name;

    public Leaf(string name) {
        _name = name;
    }
}
```

```

    }

    public void PrintName() {
        Console.WriteLine($"Leaf Name: {_name}");
    }
}

```

3. The `Leaf` class implements the `IComponent` interface. Its constructor takes a name and stores it, and the `PrintName()` method prints the name of the leaf to the console window. Add the `Composite` class:

```

public class Composite : IComponent {
    private readonly string _name;
    private readonly List<IComponent> _components;

    public Composite(string name) {
        _name = name;
        _components = new List<IComponent>();
    }

    public void Add(IComponent component) {
        _components.Add(component);
    }

    public void PrintName() {
        Console.WriteLine($"Composite Name: {_name}");
        foreach (var component in _components) {
            component.PrintName();
        }
    }
}

```

4. The `Composite` class implements the `IComponent` interface in the same way that the leaf does. Additionally, `Composite` stores a list of components that are added via the `Add()` method. Its `PrintName()` method prints out its own name, followed by the names of each of the components in the list. Now, we'll add the code to test our composite design pattern implementation:

```

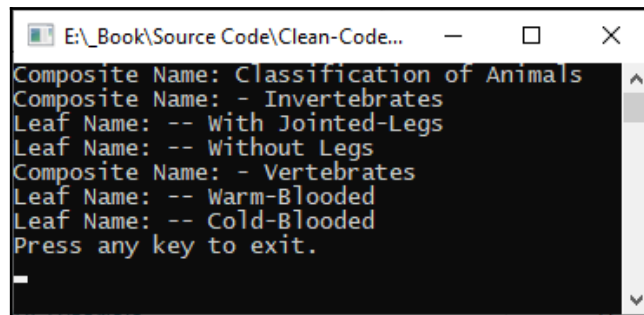
var root = new Composite("Classification of Animals");
var invertebrates = new Composite("+ Invertebrates");
var vertebrates = new Composite("+ Vertebrates");

var warmBlooded = new Leaf("-- Warm-Blooded");
var coldBlooded = new Leaf("-- Cold-Blooded");
var withJointedLegs = new Leaf("-- With Jointed-Legs");
var withoutLegs = new Leaf("-- Without Legs");

```

```
invertebrates.Add(withJointedLegs);  
invertebrates.Add(withoutLegs);  
  
vertebrates.Add(warmBlooded);  
vertebrates.Add(coldBlooded);  
  
root.Add(invertebrates);  
root.Add(vertebrates);  
  
root.PrintName();
```

5. As you can see, we create our composites and then our leaves. We then add the leaves to the appropriate composites. Then, we add our composites to the root composite. Finally, we call the root composite's `PrintName()` method, which will print the root's name, along with the names of all the components and leaves in the hierarchy. You can see the output, as follows:



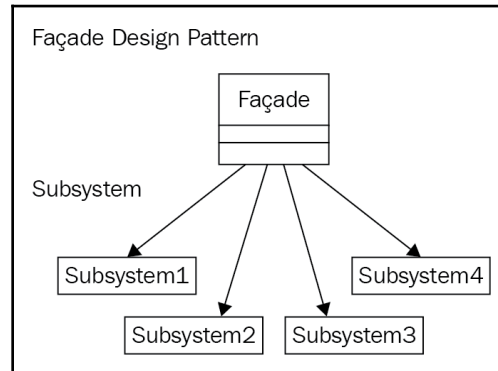
```
E:\_Book\Source Code\Clean-Code...  
Composite Name: Classification of Animals  
Composite Name: - Invertebrates  
Leaf Name: -- With Jointed-Legs  
Leaf Name: -- Without Legs  
Composite Name: - Vertebrates  
Leaf Name: -- Warm-Blooded  
Leaf Name: -- Cold-Blooded  
Press any key to exit.  
_
```

Our composite implementation is working as expected. The next pattern we will implement is the façade design pattern.

Implementing the façade pattern

The façade pattern is designed to make using API subsystems easier to use. Use this pattern to hide a large and complex system behind a much simpler interface for your clients to use. The main reason that programmers will implement this pattern is that the system they are having to use or work on is too complex and very hard to understand.

Other reasons why this pattern is employed include if too many classes are dependent on one another, or simply because programmers don't have access to the source code:



The participants in the façade pattern are as follows:

- **Facade:** The simple interface, which acts as a *go-between* between the client and a more complex system of subsystems
- **Subsystem Classes:** The subsystem classes, which are directly removed from client access and are directly accessed by the façade

We are now going to implement the façade design pattern:

1. Add a folder called **Facade** to the **StructuralDesignPatterns** folder. Then, add the **SubsystemOne** and **SubsystemTwo** classes:

```
public class SubsystemOne {  
    public void PrintName() {  
        Console.WriteLine("SubsystemOne.PrintName()");  
    }  
}  
  
public class SubsystemOne {  
    public void PrintName() {  
        Console.WriteLine("SubsystemOne.PrintName()");  
    }  
}
```

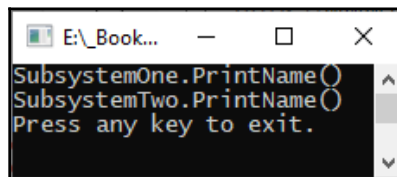
2. These classes have a single method that prints the class name and method name to the console window. Now, let's add the Facade class:

```
public class Facade {  
    private SubsystemOne _subsystemOne = new SubsystemOne();  
    private SubsystemTwo _subsystemTwo = new SubsystemTwo();  
  
    public void SubsystemOneDoWork() {  
        _subsystemOne.PrintName();  
    }  
  
    public void SubsystemTwoDoWork() {  
        _subsystemTwo.PrintName();  
    }  
}
```

3. The Facade class creates member variables for each system that it has knowledge of. It then provides a series of methods that will access various portions of each of the subsystems when requested to do so. We will add the code to test our implementation:

```
var facade = new Facade();  
facade.SubsystemOneDoWork();  
facade.SubsystemTwoDoWork();
```

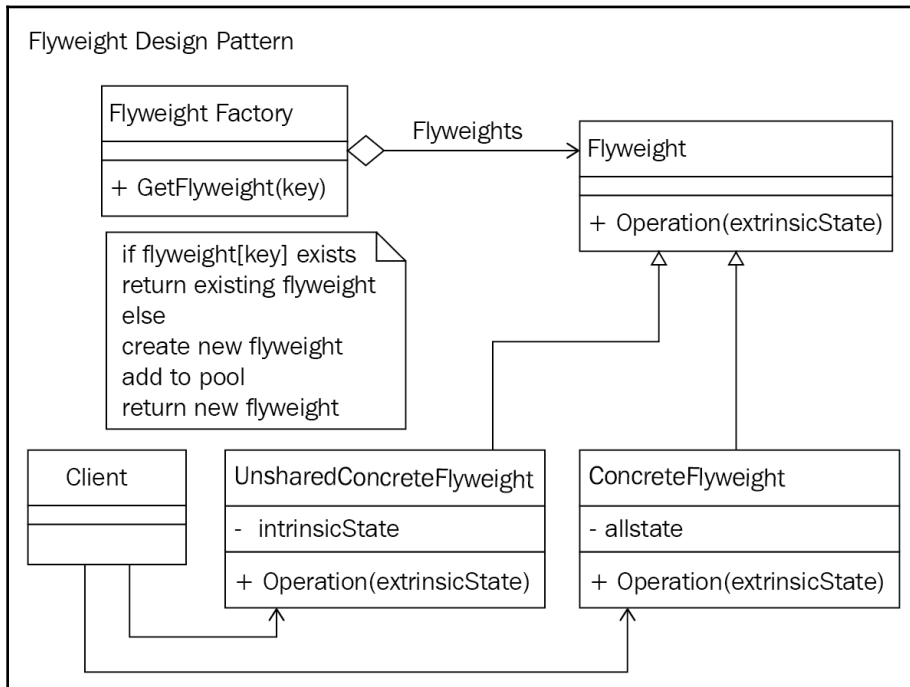
4. All we have to do is create a Facade variable, and then we can call the methods that execute method calls in the subsystems. You should see the following output:



Time to look at our final structural pattern called the flyweight pattern.

Implementing the flyweight pattern

The flyweight design pattern is used to efficiently process a large number of fine-grained objects by reducing the overall object count. Use this pattern to increase performance and reduce the memory footprint by reducing the number of objects that you create:



The participants in the flyweight design pattern are as follows:

- **Flyweight**: Provides an interface for flyweights so that they can receive an extrinsic state and act on it
- **ConcreteFlyweight**: A sharable object that adds storage for the intrinsic state
- **UnsharedConcreteFlyweight**: Used when flyweights don't need to be shared
- **FlyweightFactory**: Correctly manages flyweight objects and shares them properly
- **Client**: Maintains flyweight references and computes or stores the extrinsic state of flyweights



Extrinsic state means that it is not part of the essential nature of the object and that it originates externally to the object. **Intrinsic state** means that the state belongs to the object and is essential to the object.

Let's implement the flyweight design pattern:

1. Start by adding the `Flyweight` folder to the `StructuralDesignPatterns` folder. Now, add the `Flyweight` class:

```
public abstract class Flyweight {  
    public abstract void Operation(string extrinsicState);  
}
```

2. This class is abstract and contains an abstract method called `Operation()`, which is passed in the extrinsic state of the flyweight:

```
public class ConcreteFlyweight : Flyweight  
{  
    public override void Operation(string extrinsicState)  
    {  
        Console.WriteLine($"ConcreteFlyweight: {extrinsicState}");  
    }  
}
```

3. The `ConcreteFlyweight` class inherits the `Flyweight` class and overrides the `Operation()` method. The method outputs the method name and its extrinsic state. Now, add the `FlyweightFactory` class:

```
public class FlyweightFactory {  
    private readonly Hashtable _flyweights = new Hashtable();  
  
    public FlyweightFactory()  
    {  
        _flyweights.Add("FlyweightOne", new ConcreteFlyweight());  
        _flyweights.Add("FlyweightTwo", new ConcreteFlyweight());  
        _flyweights.Add("FlyweightThree", new ConcreteFlyweight());  
    }  
  
    public Flyweight GetFlyweight(string key) {  
        return ((Flyweight)_flyweights[key]);  
    }  
}
```

4. In our particular flyweight example, we store our flyweight objects in a *hashtable*. Three flyweight objects are created in our constructor. Our `GetFlyweight()` method returns the flyweight for the specified key from the hashtable. Now, add the client:

```
public class Client
{
    private const string ExtrinsicState = "Arbitrary state can be
anything you require!";

    private readonly FlyweightFactory _flyweightFactory = new
FlyweightFactory();

    public void ProcessFlyweights()
    {
        var flyweightOne =
        _flyweightFactory.GetFlyweight("FlyweightOne");
        flyweightOne.Operation(ExtrinsicState);

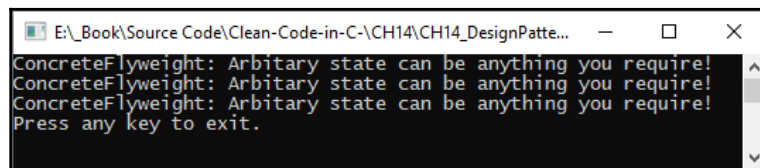
        var flyweightTwo =
        _flyweightFactory.GetFlyweight("FlyweightTwo");
        flyweightTwo.Operation(ExtrinsicState);

        var flyweightThree =
        _flyweightFactory.GetFlyweight("FlyweightThree");
        flyweightThree.Operation(ExtrinsicState);
    }
}
```

5. An extrinsic state can be anything you require it to be. In our example, we are using a string. We declare a new flyweight factory, add three flyweights, and execute the operation on each of them. Let's add the code to test our implementation of the flyweight design pattern:

```
var flyweightClient = new
StructuralDesignPatterns.Flyweight.Client();
flyweightClient.ProcessFlyweights();
```

6. The code creates a new `Client` instance, and then calls the `ProcessFlyweights()` method. You should see the following:

A screenshot of a Windows console window. The title bar shows the file path "E:_Book\Source Code\Clean-Code-in-C\CH14\CH14_DesignPatte...". The console output consists of three lines, each starting with "ConcreteFlyweight: " followed by the string "Arbitrary state can be anything you require!". After the third line, the prompt "Press any key to exit." is displayed. The console has a black background with white text.

Well, that's it for the structural patterns. Now it is time for us to look at implementing behavioral design patterns.

Overview of behavioral design patterns

As a programmer, your behavior on the team is governed by your methods of communication and interaction with other team members. The objects we program are no different. As programmers, we determine how objects will behave and communicate with other objects through the use of behavioral patterns. These behavioral patterns are as follows:

- **Chain of responsibility:** A sequential pipeline of objects that process an incoming request.
- **Command:** Encapsulates all the information that will be used to call a method at some point in time within an object.
- **Interpreter:** Provides interpretation of a given grammar.
- **Iterator:** Use this pattern to access an aggregate object's elements sequentially without exposing its underlying representation.
- **Mediator:** Use this pattern to have objects communicate with each other via an intermediary.
- **Memento:** Use this pattern to capture and save the object's state.
- **Observer:** Use this pattern to observe and be notified of changes in the object state of the object being observed.
- **State:** Use this pattern to alter the behavior of an object when its state changes.
- **Strategy:** Use this pattern to define a catalog of encapsulated algorithms that are interchangeable.
- **Template method:** Use this pattern to define an algorithm and the steps that can be overridden in subclasses.
- **Visitor:** Use this pattern to add new operations to existing objects without modifying them.

Due to the constraints of this book, we don't have enough pages left to cover the behavioral design patterns. With that in mind, I will direct you to the following books, which you can use to further your knowledge of design patterns. The first book is called *Design Patterns in C#: A Hands-on Guide with Real-World Examples*, by Vaskaring Sarcar, and published by Apress. The second book is called *Design Patterns in .NET: Reusable Approaches in C# and F# for Object-Oriented Software Design*, by Dmitri Nesteruk, also published by Apress. Published by Packt, the third book is called *Hands-On Design Patterns with C# and .NET Core*, by Gaurav Aroraa and Jeffrey Chilberto.

Between these books, you will not only come to understand all the patterns, but you will also gain exposure to real-world examples, which will help you transition from simply having head knowledge to having the practical skills to use design patterns in a reusable way in your own projects.

That's it for our look at design pattern implementations. Before we summarize what we've learned, I'll leave you with some final thoughts on clean code and refactoring.

Final thoughts

There are two types of software development—**brownfield development** and **greenfield development**. The majority of the code we work on throughout our careers will be brownfield development, which is the maintenance and extension of existing software, while greenfield development is the development, maintenance, and extension of new software. With greenfield software development, you are afforded the opportunity to write clean code from the start, and I encourage you to do just that.

Make sure that projects are properly planned before you work on them. Then, employ the tools available to you to develop clean code with confidence. When it comes to brownfield development, you are best off spending time getting to know the system inside out before you maintain or extend it. Unfortunately, you may not always be in a situation where time affords you such luxury. So, there may be times when you will set about writing the code you need, not realizing that code already exists to do the task you are implementing. Keeping the code that you do write clean and well-structured will make for easier refactoring later on in the project.

Regardless of whether the project you are working on is a brownfield or greenfield project, it is down to you to ensure that you follow the company procedures. They are there for good reasons, those reasons being harmony between the development team and a clean code base. When you encounter unclean code within the code base, you should look to refactor it immediately.

If the code is too complex to change immediately, and if too many changes across layers are necessary, then the change must be logged as technical debt on the project to be addressed at a later date after proper planning.

At the end of the day, whether you call yourself a software architect, software engineer, software developer, or anything else, for that matter, your bread and butter is your *programming skills*. Bad programming can be detrimental to your current position, and can even negatively impact your ability to find new positions. So, employ every resource you have to ensure that your current code leaves a lasting good impression of your level of ability. I once heard someone say the following:

"You are only as good as your last programming assignment!"

It is important when architecting systems not to be *too clever* and build overly complex systems. Keep the depth of inheritance of your programs to no greater than 1, and do your best to reduce loops through utilizing functional programming techniques such as LINQ.

You saw in Chapter 13, *Refactoring C# Code – Identifying Code Smells*, how LINQ is more performant than a `foreach` loop. Try to also reduce the complexity of your software by limiting the number of pathways through your computer program from the beginning to the end. Reduce boilerplate code by removing the boilerplate code to aspects that can be weaved into the code at compile time. This reduces the number of lines in your methods to only those lines that are the required business logic. Keep classes small and focused on only one responsibility. Also, keep methods to 10 lines of code or fewer. Classes and methods must only perform a single responsibility.

Learn to keep the code you write simple so that it is easy to read and reason about. Understand the code you write. If you can easily understand your code, then you're fine. Now, ask yourself this: *after working on another project and coming back to this one, would you still understand the code with little or no effort?* When code is hard to understand, then it must be refactored and simplified.

Failure to do this can result in a bloated system that dies a slow and agonizing death. Use documentation comments to document publicly accessible code. For hidden code, only use succinct and meaningful comments when the code does not adequately make sense by itself. Use patterns for common code that would often be repeated so that you **Don't Repeat Yourself (DRY)**. Indentation within Visual Studio 2019 is automatic, but the default indentation is not the same across different document types. Therefore, it is a good idea to make sure all documentation types have the same levels of indentation. Use the standard naming recommendations as suggested by Microsoft.

Give yourself programming challenges to solve without copying and pasting other people's source code. Use benchmarking (profiling) to rewrite the same code with the aim of reducing processing time. Test your code often to ensure it is behaving and doing what it is supposed to. Finally, practice, practice, and then practice some more.

We all change our programming styles over time. Some programmers' code will deteriorate over time if they are within a team of programmers that adopts a lot of poor practices. Other programmers' code will improve over time if they are within a team of programmers that adopts a lot of best practices. Don't forget, just because code compiles and does what it is meant to, it does not necessarily mean that it is the cleanest or most performant code.

Your aim as a computer programmer is to write clean and efficient code that is easy to read, reason, maintain, and expand. Practice implementing TDD and BDD, along with the software paradigms of KISS, SOLID, YAGNI, and DRY.

Consider checking out some old code from GitHub to use as a training opportunity in migration of old .NET versions to new .NET versions, and refactoring the code to make it clean and performant, as well as adding documentation comments to produce API documentation for the development team. This is good practice for honing your personal computer programming skills. By doing this, you can often come across some rather clever code that you can personally learn from. Other times, it can be a case of wondering what the programmer was thinking at the time! But either way, improving your clean coding skills at every opportunity you have will only work toward making you a stronger and better programmer.

Another saying that I believe to be true in the field of programming is as follows:

"To become a true expert computer programmer, you have to push yourself beyond what you are currently capable of doing."

So, no matter how expert you or your peers consider you to be, always remember that you can do even better. Therefore, keep pushing forward and upping your game. Then, when you retire, you can look back on your career with a righteous pride in your wonderful accomplishments as a computer programmer!

Let's now summarise what we have learned in this chapter.

Summary

In this chapter, we covered several creational, structural, and behavioral design patterns. You used the knowledge that you gained in this chapter to look at legacy code and understand its goal. Then, you used the patterns that you learned to implement in this chapter to refactor existing code and make it easier to read, reason, maintain, and extend. By using the patterns in this book, and the many others that are available to you, you can refactor existing code and write clean code from the start.

You also used the creational design patterns to solve real-world problems and to improve the efficiency of your code. Use structural design patterns to improve the overall structure of code and improve relations between objects. Also, use behavioral design patterns to improve communication between objects whilst maintaining the decoupling of those objects.

Well, this is the end of the chapter, and I thank you for taking the time to read this book and work through the code examples. Remember, software should be a joy to work with. As such, we don't need unclean code causing problems for our business, its development and support teams, and for the customers of the software. So, think about the code you are writing, and always strive to be a better programmer than you are today—no matter how many years you have been in the industry. There is an old saying: *no matter how good you are, you can always do better!*

Let's test your knowledge on the contents of this chapter, and then I will leave you with some further reading. Happy clean coding in C#!

Questions

1. What are GoF patterns and why would we use them?
2. Explain what creational design patterns are used for and list them.
3. Explain what structural design patterns are used for and them.
4. Explain what behavioral design patterns are used for and list them.
5. Is it possible to overuse design patterns and call code smells?
6. Describe the singleton design pattern and when you'd use it.
7. Why would we use factory methods?
8. What design pattern would you use to hide the complexity of a system that is large and difficult to use?
9. How can you minimize memory usage and share common data between objects?
10. What pattern is used to decouple an abstraction from its implementation?
11. How can you construct multiple representations of the same complex object?
12. If you have an item that requires various stages of manipulation to get it into the required state, what pattern would you use and why?

Further reading

- *Refactoring: Improving the Design of Existing Code*, by Martin Fowler
- *Refactoring at Scale*, by Maude Lemaire
- *Software Development, Design, and Coding: With Patterns, Debugging, Unit Testing, and Refactoring*, by John F. Dooley
- *Refactoring for Software Design Smells*, by Girish Suryanarayana, Ganesh Samarthayam, and Tushar Sharma
- *Refactoring Databases: Evolutionary Database Design*, by Scott W. Ambler and Pramod J. Sadalage
- *Refactoring to Patterns*, by Joshua Kerievsky
- *C#7 and .NET Core 2.0 High Performance*, by Ovais Mehboob Ahmed Khan
- *Improving Your C# Skills*, by Ovais Mehboob Ahmed Khan, John Callaway, Clayton Hunt, and Rod Stephens
- *Patterns of Enterprise Application Architecture*, by Martin Fowler
- *Working Effectively with Legacy Code*, by Michael C. Feathers
- <https://www.dofactory.com/products/dofactory-net>: C# Design Pattern Framework for RAD by dofactory
- *Hands-On Design Patterns with C# and .NET Core*, by Gaurav Aroraa and Jeffrey Chilberto
- *Design Patterns Using C# and .NET Core*, by Dimitris Loukas
- *Design Patterns in C#: A Hands-on Guide with Real-World Examples*, by Vaskaring Sarcar

Assessments

Chapter 1

1. One outcome of bad code is that you can end up with a really badly written piece of code that is hard to understand. This can often lead to programmer stress and software that is buggy, hard to maintain, and hard to test and extend.
2. One outcome of good code is that it is easy to read and understand, as you know the programmer's intent. This leads to less stress for programmers who must debug the code, test it, and extend it.
3. When you break a large project up into modular components and libraries, each module can be worked on by separate teams concurrently. Small modules are easy to test, code, document, deploy, extend, and maintain.
4. **DRY** stands for **Don't Repeat Yourself**. Look for repeatable code, and refactor it so that you remove duplicate code. The advantage of this is smaller programs, because if such code contains bugs, you only have to change it in one place.
5. **KISS** means simple code that will not confuse programmers, especially if you have juniors on your team. **KISS** code is easy to read and write tests for.
6. **S** is the **Single Responsibility Principle**, **O** is the **Open/Closed Principle**, **L** is **Liskov Substitution**, **I** is the **Interface Segregation Principle**, and **D** is the **Dependency Inversion Principle**.
7. **YAGNI** is short for **You Aren't Going to Need It**. In other words, don't add code you don't need. Only add the code you absolutely need, and no more.
8. Occam's Razor is the principle that states: *Entities must not be multiplied without necessity. Deal only in facts. Only make assumptions if absolutely necessary.*

Chapter 2

1. The two roles in the peer code review are reviewer and reviewee.
2. The project manager agrees on the people that will be involved in the peer code review.

3. You can save your reviewer time and effort prior to requesting a peer code review by making sure your code and tests all work, that you perform code analysis on your project and fix any issues raised, and that your code adheres to the company coding guidelines.
4. When reviewing code, look out for naming, formatting, programming styles, potential bugs, correctness of code and tests, security, and performance issues.
5. The three categories of feedback are positive, optional, and critical.

Chapter 3

1. We can place our code in individual source files in folder structures and wrap classes, interfaces, structs, and enums in namespaces that map to the folder structure.
2. A class should have only one responsibility.
3. You can comment in your code for document generators using XML comments placed directly above the public member to be documented.
4. Cohesion is the logical grouping together of code that works on the same responsibility.
5. Coupling refers to the dependencies between classes.
6. Cohesion should be high.
7. Coupling should be low.
8. You can use DI and IoC to design for change.
9. **DI** stands for **Dependency Injection**.
10. **IoC** stands for **Inversion of Control**.
11. Immutable objects are type-safe and so can be safely passed between threads.
12. Objects should expose methods and properties and hide data.
13. Data structures should expose data and have no methods.

Chapter 4

1. Methods with no parameters are called niladic methods.
2. Methods with only one parameter are called monadic methods.
3. Methods with two parameters are called dyadic methods.
4. Methods with three parameters are called triadic methods.

5. Methods with more than three parameters are called polyadic methods.
6. You should avoid duplicate code. It is not a productive way to program, can make programs unnecessarily large, and has the propensity to proliferate the same exception throughout your codebase.
7. Functional programming is a software coding methodology that treats computations as the mathematical evaluation of computations that does not modify state.
8. The advantages of functional programming include safe code in multithreaded applications and smaller, more meaningful methods that are easy to read and understand.
9. Input and output can be a problem for functional programs as it relies on side-effects. Functional programming does not allow for side-effects.
10. WET code is the opposite of DRY in that code is written each time it is needed. This produces duplication, and the same exception can occur in multiple locations within a program, making maintenance and support more difficult.
11. DRY code is the opposite of WET in that code is only ever written once and is reused wherever it is needed. This reduces the code base and exception footprint, thus making programs easier to read and maintain.
12. You DRY out WET code by removing duplicate code using refactoring.
13. Long methods are cumbersome and prone to exceptions. The smaller they are, the easier they are to read and maintain. There is also less chance of the programmer introducing bugs, especially of a logical nature.
14. To avoid having to use try/catch blocks, you can write argument validators. You would then call the validators at the top of your method. If the parameters fail validation, then the appropriate exception is thrown, and the method is not executed.

Chapter 5

1. A checked exception is an exception that is checked at compile time.
2. An unchecked exception is an exception that is not checked or simply ignored at compile time.
3. An overflow exception is raised when high-order bits cannot be assigned to the destination type. In checked mode, `OverflowException` is raised. In unchecked mode, high-order bits that cannot be assigned are simply ignored.
4. An attempt made to access a property or a method on a null object.

5. Implement a `Validator` class and an `Attribute` class that checks the parameter for null, and that throws `ArgumentNullException`. You would use the `Validator` class at the top of your methods so that you don't get halfway through the method before the exception is raised.
6. **Business Rule Exception (BRE).**
7. BREs are bad practice because they expect exceptions to be raised in order to control program flow.
8. Correct programming should never control the flow of a computer program by expecting exceptions as output. So, given that BREs are bad as they expect exceptional output and use it to control program flow, a better solution is to use conditional programming. With a conditional program, you use Boolean logic. Boolean logic allows for two possible paths of execution, and never raises exceptions. Conditional checks are explicit and make the programs easier to read and maintain. You can also easily extend such code, whereas with BREs, you can't.
9. First, start with error trapping for known types of exceptions such as `ArgumentNullException` and `OverflowExceptions` using known exception types in the Microsoft .NET Framework. But when these are insufficient and don't provide enough data for your particular situation, then you would write and use your own custom exceptions and apply meaningful exception messages.
10. Your custom exception must inherit from `System.Exception`, and implement three constructors: the default construct, a constructor that accepts a text message, and a constructor that accepts a text message and an inner exception.

Chapter 6

1. A good unit test must be atomic, deterministic, repeatable, and fast.
2. A good unit test must not be inconclusive.
3. Test-driven development.
4. Behavioral-driven development.
5. A small unit of code whose only purpose is to test a single unit of code that only does one thing.
6. A fake object used by the unit test to test the public methods and properties of a real object, but without testing the method or property dependencies.
7. A fake object is the same as a mock object.
8. `MSTest`, `NUnit`, and `xUnit`.

9. Rhino Mocks and Moq.
10. SpecFlow.
11. Unnecessary comments, dead code, and redundant tests.

Chapter 7

1. The testing of a complete system from end to end. This can be performed manually, automatically, or by using both methods.
2. Integration testing.
3. Manual testing of all features, all our unit tests should pass, and we should write automation tests to test the commands and data that are passed between two modules.
4. Factories are classes that implement the factory method pattern whose intention is to allow the creation of objects without specifying their classes. We would use them in the following scenarios:
 1. The class is unable to anticipate the type of object that must be instantiated.
 2. The subclass must specify the type of object to instantiate.
 3. The class controls the instantiation of its objects.
5. DI is a method of producing loosely coupled code that is easy to maintain and extend.
6. Using a container makes the management of dependency objects easy.

Chapter 8

1. A thread is a process.
2. One.
3. Background threads and foreground threads.
4. The background thread.
5. The foreground thread.
6. `Thread.Sleep(500);`
7. `var thread = new Thread(Method1);`

8. Set `IsBackground` equal to `true`.
9. A deadlock is a situation when two threads are blocked and waiting on the other thread to release the resource.
10. `Monitor.Exit(objectName);`
11. Multiple threads using the same resource generate different outputs based on the timings of each thread.
12. Use the TPL with `ContinueWith()`, and use `Wait()` to wait until the task has finished before exiting the method.
13. Using a member variable that is shared by other methods, and passing in reference variables.
14. Yes.
15. The `ThreadPool`.
16. It is an object that cannot be modified once it has been constructed.
17. They allow you to safely share data between threads.

Chapter 9

1. Application Programming Interface.
2. Representational State Transfer.
3. Uniform interface, client-server, stateless, cacheable, layered system, optional executable code.
4. **Hypermedia as the Engine of Application State (HATEOAS).**
5. `RapidApi.com`.
6. Authorization and authentication.
7. Claims are statements that an entity makes about itself. These claims are then validated against a data store. They are particularly useful in role-based security to check whether the entity making the claim is authorized in regard to that claim.
8. Making API requests and examining their responses.
9. Because you can change your data store in keeping with your requirements.

Chapter 10

1. The correct partitioning of software into logical namespaces, interfaces, and classes, which aids the testing of software.
2. By understanding APIs, you can KISS your code and keep it DRY by not reinventing the wheel and writing code that already exists. This saves time, energy, and money.
3. Structs.
4. Third-party APIs are written by software developers, and so subject to human error that introduces bugs. By testing third-party APIs, you can be confident they work as expected, and if not, then you can have the code fixed or write a wrapper for it.
5. Your APIs are prone to errors. By testing them in keeping with the specification and its acceptance criteria, you can be sure you are delivering what the business wants at the agreed level of quality ready for public release.
6. The specification and acceptance criteria provide the normal program flow. From them, you can determine what to test in regard to the normal flow of execution, and you can determine what exceptional circumstances will be encountered and test for them.
7. Namespaces, interfaces, and classes.

Chapter 11

1. Cross-cutting concerns are those concerns that not part of the business requirements that form the core concerns, but that must be addressed in all areas of the code. **AOP** stands for **Aspect-Oriented Programming**.
2. An aspect is an attribute that, when applied to a class, method, property, or parameter, injects code at compile time. You apply an aspect in square brackets before the item it is being applied to.
3. An attribute gives semantic meaning to an item. You apply an attribute in square brackets before the item it is being applied to.
4. Attributes give the code semantic meaning, while aspects remove the boilerplate code so that it is injected at compile time.
5. When the code is being built, the compiler will insert the boilerplate code that the aspect hides from the programmer. This process is known as code weaving.

Chapter 12

1. Code metrics are several source code measurements that enable us to identify how complex our software is, and how maintainable it is. Such measurements enable us to identify areas of code that can be made less complex and more maintainable through refactoring.
2. Cyclomatic complexity, maintainability index, depth of inheritance, class coupling, lines of source code, and lines of executable code.
3. Code analysis is the static analysis of source code with the intention of identifying design flaws, issues with globalization, security problems, issues with performance, and interoperability problems.
4. Quick actions are single commands identified by a screwdriver or lightbulb that will suppress warnings, add using statements, import missing libraries and add the using statements, correct errors, and implement language usage improvements aimed at simplifying code and reducing the number of lines in a method.
5. JetBrains' dotTrace utility is a profiling tool used for the purpose of profiling source code and compiled assemblies to identify potential issues with the software. With it you can perform sampling, tracing, line-by-line, and timeline profiling. You can profile execution time, thread time, real-time CPU instructions, and thread cycle time.
6. JetBrains' ReSharper utility is a code refactoring tool that helps developers identify and fix code issues and implement language features to improve and speed up the programmer's programming experience.
7. The decompilation of source code can be used to retrieve lost source code, generate PDBs for debugging, and for learning. You can also use the decompiler to see how well you have obfuscated your code to make it hard for hackers and other people to steal your code secrets.

Chapter 13

1. Application-level, class-level, and method-level.
2. Boolean blindness, combinatorial explosion, contrived complexity, data clump, deodorant comments, duplicate code, lost intent, mutation of variables, oddball solution, shotgun surgery, solution sprawl, and uncontrolled side effects.

3. Cyclomatic complexity, divergent change, downcasting, excessive literal use, feature envy, inappropriate intimacy, indecent exposure, large class (also known as God object), lazy class (also known as freeloader and lazy object), middleman class, an orphan class of variables and constants, primitive obsession, refused bequest, speculative generality, Tell, don't ask!, and temporary field.
4. Black sheep, cyclomatic complexity, contrived complexity, dead code, excessive data return, feature envy, identifier size, inappropriate intimacy, long line aka God line, lazy method, long method (God method), long parameter list (too many parameters), message chains, middleman method, oddball solutions, and speculative generality.
5. Use LINQ instead of loops. Make classes responsible for only one thing. Make methods do only one thing. Replace long lists of parameters with parameter objects. Use creational design patterns to improve the efficiency of expensive object creation and utilization. Keep methods to 10 lines or less. Use AOP to remove boilerplate code from methods. Decouple objects and make them testable. Make code highly cohesive.
6. A value that represents the amount of branching and looping.
7. Reduce the amount of branching and looping that takes place until such time as the cyclomatic complexity value becomes 10 or less.
8. Making things more complicated than they need to be.
9. **Keep It Simple, Stupid (KISS).**
10. The same thing is done by different methods with different parameter combinations.
11. Create generic methods that can perform the same task on the different data types so that you only have the one method with one set of parameters.
12. Fix the bad code and remove the comment.
13. Ask for help.
14. Stack Overflow.
15. A long parameter list can be replaced with a parameter object.
16. Refactor it into smaller methods that do only one thing, and remove boilerplate code into aspects using AOP.
17. No more than 10 lines.
18. 0-10; anything beyond that and you are asking for trouble.
19. One.
20. Variables, classes, properties, and methods that are not used. Get rid of them.
21. Choose the best method of implementation, and then refactor the code to use just that method of implementation.

22. Refactor the temporary field and the methods that operate on it into their own class.
23. The same set of variables used in different classes. Refactor the variables into a class of their own, and then reference the class.
24. A class inherits from another class but does not use all its methods.
25. The Law of Demeter.
26. Only allow classes to speak to their immediate neighbors.
27. A class or method spending too much time inside another class or method.
28. Refactor dependencies in their own class or method.
29. The factory method.
30. Inherit from a base class, and then create the new class that inherits from the base.
31. Single responsibility is implemented in different methods of different classes across different layers of the application. Refactor the responsibility into its own class so that it is only in a single location.
32. The data should be placed in the same object that operates on it.
33. When you create an object that asks another object for data so that it can perform operations on it.
34. A single change requires changes in multiple locations. Remove duplication, remove the coupling, and improve cohesion.
35. Lost intent is when the reason for the class or method is unclear because there are lots of unrelated items clumped together. Refactor the code so that all methods are in the right class. That way, the intent of the class and the methods becomes clear.
36. You can refactor loops with LINQ queries. LINQ is a functional language that does not alter location variables and can perform much faster than loops.

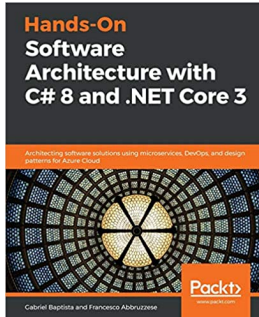
Chapter 14

1. **GoF** is short for **Gang-of-Four** patterns. These are 23 patterns that are grouped into creational, structural, and behavioral design patterns. They are considered the foundation of all software design patterns. They work together to produce clean object-oriented code.
2. Creational patterns enable abstraction and inheritance to provide an object-oriented way of removing code duplication and improving performance when object creation is expensive. The creational patterns are abstract factory, factory method, singleton, prototype, and builder.

3. Structural patterns enable the correct management of relationships between objects. We can use structural patterns to enable incompatible interfaces to work together, decouple abstractions from their implementations, and improve performance. The structural patterns are adapter, bridge, composite, decorator, façade, flyweight, and proxy.
4. Behavioral patterns govern how objects interact and communicate with each other. We can use them to produce pipelines, encapsulate commands and information to be executed at a future point in time, mediate between objects, observe state changes in objects, and more. The behavior patterns are chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method, and visitor.
5. Yes.
6. The singleton only allows a single instance of an object throughout the lifetime of the application. The object is globally accessible to all objects that need it. We use this pattern when we need to ensure we have one centralized point of object creation and object access.
7. We use factory methods when we have a need to create objects without specifying the exact class to be instantiated.
8. Façade.
9. Use the flyweight design pattern.
10. Bridge.
11. Use the builder pattern.
12. You would use the chain of responsibility pattern, as you can have a pipeline of handlers, each of which performs a task. If they are unable to handle the task, the handlers pass the task to their successor to handle.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

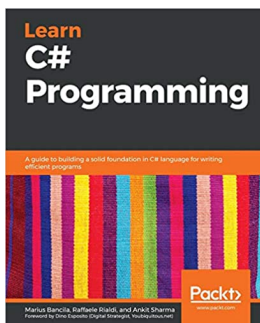


Hands-On Software Architecture with C# 8 and .NET Core 3

Francesco Abbruzzese, Gabriel Baptista

ISBN: 978-1-78980-093-7

- Overcome real-world architectural challenges and solve design consideration issues
- Apply architectural approaches like Layered Architecture, service-oriented architecture (SOA), and microservices
- Learn to use tools like containers, Docker, and Kubernetes to manage microservices
- Get up to speed with Azure Cosmos DB for delivering multi-continental solutions
- Learn how to program and maintain Azure Functions using C#
- Understand when to use test-driven development (TDD) as an approach for software development
- Write automated functional test cases for your projects



Learn C# Programming

Marius Bancila, Raffaele Rialdi, Ankit Sharma

ISBN: 978-1-78980-586-4

- Get to grips with all the new features of C# 8
- Discover how to use attributes and reflection to build extendable applications
- Utilize LINQ to uniformly query various sources of data
- Use files and streams and serialize data to JSON and XML
- Write asynchronous code with the async-await pattern
- Employ .NET Core tools to create, compile, and publish your applications
- Create unit tests with Visual Studio and the Microsoft unit testing frameworks

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

- abstract factory pattern
 - implementing 422, 423, 424, 425
- Adapter Pattern
 - using 395
- admin module (subsystem) 179, 180
- advice 334
- agnostic RAML design specification
 - C# API, generating from 277, 278, 280, 281
- API development
 - Swagger for 262, 263, 264
- API documentation
 - quality, need for 261, 262
- API endpoints 258
- API proxy 254, 255, 256
- API Workbench
 - installing, by MuleSoft 272, 273
- Application Lifecycle Management (ALM) 360
- Application Programming Interfaces (APIs), design guidelines
 - API documentation, quality need for 261, 262
 - immutable structs, passing instead of mutable objects 265, 266, 267
 - well-defined software boundaries 259, 260, 261
- Application Programming Interfaces (APIs)
 - about 251, 252, 253, 254, 374
 - design guidelines 256, 257, 258, 259
 - designing, with RAML 271
 - project, creating 274, 276
 - testing 269, 270, 271
- application-level code smells
 - about 387
 - Boolean blindness 387, 388, 389
 - combinatorial explosion 389, 390
 - contrived complexity 390, 391
 - data clump 391
 - deodorant comment 391
 - duplicate code 392
 - lost intent 392, 393
 - mutation of variables 393, 394, 395
 - oddball solution 395, 396, 397
 - shotgun surgery 397, 398, 399
 - solution sprawl 399
 - uncontrolled side effects 399
- arguments 100
- Aspect-Oriented Programming (AOP), with PostSharp
 - architectural framework, extending 337, 338
 - aspect, developing 334
 - behaviors, injecting before and after method execution 334, 335, 336, 337
 - framework, extending 334
- Aspect-Oriented Programming (AOP)
 - about 33, 327, 392
 - with PostSharp 333
- Atom
 - installation link 272
 - installing, by MuleSoft 272, 273
- attribute programming 327
- Azure Active Directory (Azure AD) 288
- Azure Key Vault
 - Morningstar API key, storing 286, 287, 288, 289
- Azure
 - dividend calendar ASP.NET Core web application, creating 289, 290
 - home page, reference link 296
 - reference link 286

B

- background threads 206
- bad code
 - versus good code 9
- bad coding practices

- about 9
- bad naming conventions 13
- code, that is difficult to read 19, 20
- code, that is tightly coupled 20, 21
- commented-out lines 12
- comments, that excuse bad code 11
- comments, that state the obvious 10, 11
- directly expose information 24
- exceptions, using to control program flow 18, 19
- Finalize() method, using 22
- improper indentation 10
- improper organization, of namespaces 12
- Keep It Simple, Stupid (KISS) 22, 23
- large classes, lack of regions 23
- lost-intention code 23, 24
- low cohesion 21
- methods 16, 17
- methods, with more than 10 lines of code 17
- methods, with more than two parameters 18
- multiple jobs, classes 13, 14, 15, 16
- objects, left hanging around 21, 22
- over-engineering 22
- behavioral design patterns
 - overview 445, 446
- Behavioral-Driven Development (BDD) 33, 131
- binary
 - reference link 218
- Boolean blindness 387, 388, 389
- bounded context 259
- bridge pattern
 - implementing 434, 435, 436
- brownfield development 446
- builder pattern
 - implementing 427, 428, 429, 430, 431, 432, 433
- business requirement
 - designing 75
- Business Rule Exceptions (BREs)
 - about 18, 117, 118, 119, 120
 - used, for handling conditions within program 120, 121

C

C# API

- generating, from agnostic RAML design

- specification 277, 278, 280, 281
- C# exception handling
 - best practices 127, 128
- C# programming
 - multiple parameters, removing 100, 101
- central processing unit (CPU) 358
- checked exceptions 110, 111, 112, 113, 114
- class coupling 361
- class-level code smells, cyclomatic complexity
 - readability, improving of conditional checks within if statement 403, 404
 - switch statements, replacing with factory pattern 400, 401, 402, 403
- class-level code smells
 - about 400
 - cyclomatic complexity 400
 - divergent change 404, 405
 - downcasting 405
 - feature envy 405, 407
 - freeloading class 408, 409
 - God object 408
 - inappropriate intimacy 407
 - indecent exposure 408
 - literal, using 405
 - middleman class 409
 - orphan class, of variables and constants 409
 - primitive obsession 409
 - refused bequest 410
 - speculative generality 410
 - Tell, Don't Ask 410
 - temporary fields 410
- classes
 - organizing 62, 64, 65
 - responsibility 65, 66, 67
- code analysis
 - performing 363, 364, 365, 366
- code block 98
- code cleanup
 - performing 360, 361, 362, 363
- code metrics
 - calculating 360, 361, 362, 363
- code review feedback
 - effects 48, 49, 50
 - providing 57
 - providing, as reviewer 57, 58

- responding to 57
- responding to, as reviewee 58
- code review process 39, 40
- code review, aspects
 - about 51, 55
 - architectural guidelines 54
 - business requirements 51
 - company's coding guidelines 51
 - design patterns 54
 - formatting 52
 - naming conventions 51
 - performance 54
 - security 54
 - testing 52, 53
- code review
 - leading 41, 42
 - performing 55, 56
 - preparing for 40, 41
 - pull request, issuing 43, 44, 45
 - pull request, responding to 46, 47, 48
- code smell 386
- coding conventions
 - about 33
 - reference link 33
- coding methodologies
 - need for 31
- coding principles
 - about 32
 - need for 31
- coding standards
 - about 31, 32
 - need for 31
 - references 32
- cohesion 70
 - high cohesion, example 74, 75
 - low cohesion, example 73
- combinatorial explosion 389, 390
- comments
 - removing 169, 170
- Common Language Runtime (CLR) 21
- composite pattern
 - implementing 436, 437, 438, 439
- concurrency 204
- contrived complexity 390, 391
- core concerns 326

- coupling
 - low coupling, example 72, 73
 - tight coupling, example 71
- creational design patterns
 - abstract factory pattern 422, 423, 424, 425
 - builder pattern 427, 428, 429, 430, 431, 432, 433
 - factory method pattern 420, 421, 422
 - implementing 418
 - prototype pattern 425, 426, 427
 - singleton design pattern 419, 420
- critical feedback 58
- cross-cutting concerns reusable library
 - about 338
 - caching concern, adding 338, 339
 - configuration settings concern, adding 354, 355
 - exception-handling concern, adding 343, 344
 - instrumentation concern, adding 355, 356
 - logging capabilities, adding 340, 341
 - logging concern, adding 341, 342
 - resource pool concern, adding 353, 354
 - security concern, adding 344, 345, 346, 347
 - transaction concern, adding 353
 - validation concern, adding 348, 349, 350, 352
- custom exceptions
 - building 125, 126, 127
- cyclomatic complexity 360

D

- data clump 391
- data structure
 - data, exposing 88
 - example 89
- data transfer objects (DTOs) 93
- data
 - exposing, in data structure 88
 - hiding, in objects 87
- dead code
 - about 169
 - removing 169, 170
- deadlock example
 - coding 220, 221, 222, 223, 224, 225
- deadlocks
 - about 219
 - preventing 219

- decorator design pattern 327, 329, 330
- deodorant comment 391
- Dependency Injection (DI)
 - about 79, 194, 195, 196, 197, 199, 200
 - example 79, 80, 81
 - types 194
- Depth of Inheritance 361
- divergent change 404, 405
- dividend calendar API, securing with Morningstar
 - API key
 - authentication, adding 300, 301, 302, 303, 304
 - authentication, setting up 300
 - authorization, adding 304, 305, 306, 307
 - authorization, setting up 300
 - repository, setting up 297, 298, 299
- dividend calendar API, securing
 - Morningstar API key, using 297
- dividend calendar API
 - about 284, 285
 - throttling 319, 320, 321, 322, 323
- dividend calendar ASP.NET Core web application
 - creating, in Azure 289, 290
 - publishing 291, 292, 293, 294, 295, 296, 297
- dividend calendar code
 - adding 311, 312, 313, 314, 315, 316, 317, 318, 319
- document generation
 - commenting 67, 69, 70
- Domain-Driven Design (DDD) 259
- domains 259
- Don't Repeat Yourself (DRY) 32, 35, 99, 261, 447
- downcasting 405
- DRY code
 - duplication, removing 99
- duplicate code 392
- dyadic methods 100

E

- E2E testing, subsystems
 - admin module (subsystem) 179, 180
 - login module (subsystem) 175, 176, 177, 178, 179
 - test module (subsystem) 181, 182
- E2E testing
 - about 173, 174, 175

- subsystems 175
- encapsulation
 - example 87, 88
- End-to-end (E2E)
 - about 173
 - used, for testing three-module system 182, 183, 184, 185
- exceptions 123, 124

F

- factories
 - implementing, with factory method pattern 185, 186, 187, 188, 189, 190, 191, 192, 193, 194
- factory method pattern
 - implementing 420, 421, 422
 - used, for implementing factories 185, 186, 187, 188, 189, 190, 191, 192, 193, 194
- façade pattern
 - implementing 439, 440, 441
- feature envy 405
- flyweight pattern
 - implementing 442, 443, 444, 445
- foreground threads 206
- functional programming
 - about 92, 93, 94, 95
 - code, indenting 97, 98, 99
 - keeping, methods small 95, 96, 97

G

- Gang-of-Four (GoF) 54, 417
- Garbage Collection (GC) 21
- God method 102
- God object 102
- good code
 - versus bad code 9
- good coding practices
 - about 24
 - API documentation comments 25
 - classes, that only do one job 27
 - code, that is loosely coupled 29
 - code, that is readable 28
 - Dispose() method, using 29, 30
 - exceptions, using properly 28
 - Finalize() method, avoiding 30
 - good naming conventions 26, 27

- high cohesion 29
- meaningful comments 25
- methods, that do one thing 27
- methods, with 4 lines 27
- methods, with less than 10 lines 27
- methods, with no more than two parameters 28
- namespaces, using for proper organization 26
- proper indentation 25
- regions, using in large classes 31
- right level of abstraction 30
- good-quality code 358, 359
- greenfield development 446

H

- Her Majesty's Revenue and Customs (HMRC) 219
- high cohesion
 - about 70
 - example 74, 75
- HTTP status codes
 - URL 257
- Human Resources Index (ihridx) 13
- Hypermedia as the Engine of Application State (HATEOAS) 253
- HyperText Transfer Protocol (HTTP) 367

I

- immutability
 - demonstrating 236, 238
- immutable data structure type
 - about 85
 - example 86
- immutable object type
 - about 85
 - example 86
- immutable types 234
- indecent exposure 408
- input/output (I/O) 358
- integration testing 173, 202
- Interface-Oriented Programming (IOP) 76, 77, 78, 79
- Interlocked class
 - using 243, 244, 245, 246, 247
- Intermediate Language (IL) 359
- International Resource Identifier (IRI) 256
- Internet Information Services (IIS) 368

Inversion of Control (IoC)

- about 79
- example 81, 82

J

- JavaScript Object Notation (JSON) 285
- JetBrains dotTrace profiler
 - using 367, 368, 369, 370, 371, 372
- JetBrains ReSharper
 - using 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382

K

- Keep It Simple, Stupid (KISS) 22, 23, 32, 34, 261, 390

L

- Law of Demeter
 - about 83
 - example 83, 84, 85
- login module (subsystem) 175, 176, 177, 178, 179
- London Stock Exchange (LSE) 127
- low cohesion
 - about 70
 - example 73
- low coupling
 - example 72, 73

M

- Maintainability Index 360
- Market Identification Code (MIC) 313
- method-level code smells
 - about 411
 - black sheep method 411
 - contrived complexity 411
 - cyclomatic complexity 411
 - dead code 411
 - excessive data return 412
 - feature envy 412
 - God lines 413
 - God methods 413
 - identifier size 412
 - inappropriate intimacy 412
 - lazy method 413

- long parameter lists 413
- message chain 413
- middleman method 414
- oddball solution 414
- speculative generality 414
- methods
 - exposing, in objects 87
- Model-View-ViewModel (MVVM) 333
- modularity 33
- modularization 200, 201, 202
- Monadic methods 100
- Moq 152, 153, 154, 155, 156, 157
- Morningstar API key
 - security, testing 308, 309, 310, 311
 - storing, in Azure Key Vault 286, 287, 288, 289
 - used, for securing dividend calendar API 297
- Morningstar API
 - accessing 286
 - reference link 286
- MSTest Framework
 - installing 137, 139, 140, 141, 142, 143, 144, 145
- MuleSoft
 - Atom and API Workbench, installing by 272, 273
- multi-threaded applications
 - general recommendations, from Microsoft 247, 248
- multi-threading 204
- mutability
 - about 234
 - demonstrating 234, 236
- Mutual Exclusion Objects (mutexes)
 - about 212
 - benefits 213
 - disadvantages 213
 - using, with synchronous threads 212, 214, 215

N

- Niladic methods 100
- normal program flow
 - used, for handling conditions within program 121, 122, 123
- NullPointerExceptions
 - avoiding 114, 115, 116, 117
- NUnit 145, 146, 148, 150, 151, 152

O

- Object-Oriented Programming (OOP) 19, 361
- objects
 - data, hiding 87
 - methods, exposing 87
- Occam's razor 36
- oddball solution 395, 396, 397
- optional feedback 58

P

- Peer Review (PR) 57
- Performance, Availability, Security, Scalability, Maintainability, Accessibility, Deployability, and Extensibility (PASSMADE) 358
- polyadic methods 100
- positive feedback 58
- PostSharp
 - Aspect-Oriented Programming (AOP) with 333
 - URL 333
- principle of least astonishment (POLA) 85
- process 204
- prototype pattern
 - implementing 425, 426, 427
- proxy pattern 331, 332, 333
- pull request
 - issuing 43, 44, 45
 - responding to 46, 47, 48

Q

- Quick Action tool
 - using 366, 367

R

- race conditions
 - preventing 225, 226, 227
- RAML
 - Application Programming Interfaces (APIs), designing with 271
 - URL 271, 281
- redundant tests
 - removing 169, 170
- refused bequest 410
- repository pattern 297
- Representational State Transfer (REST) 252

- resource 258
- REST services
 - constraints 253
- RESTful APIs 252

S

- semaphores
 - about 215
 - advantages 215
 - used, for working with parallel threads 215, 216, 217
- shotgun surgery 397, 398, 399
- Single Responsibility Principle (SRP)
 - about 14, 65, 92
 - implementing 103, 104, 105, 106, 107
- Single Responsibility Principle (SRP)SRP
 - implementing 102
- Single Responsibility Principle, Open-Closed Principle, Liskov Substitution, Interface Segregation Principle, and Dependency Inversion Principle (SOLID)
 - about 32, 35, 36, 261
 - principles 35
- Single-Page Applications (SPAs) 259
- singleton design pattern
 - implementing 419, 420
- Software Development Life Cycle (SDLC) 56
- SpecFlow
 - about 157, 159, 160, 162
 - URL 162
- ssynchronized method dependencies 243
- static constructors
 - about 228
 - adding, to sample code 229, 230
- static methods
 - about 228
 - adding, to sample code 230, 231, 232, 233
- structural design patterns
 - bridge pattern 434, 435, 436
 - composite pattern 436, 437, 438, 439
 - façade pattern 439, 440, 441
 - flyweight pattern 442, 443, 444, 445
 - implementing 433
- Structured Query Language (SQL) 367
- synchronized method dependencies 242

- synchronous threads
 - mutex, using with 212, 214, 215

T

- Task Parallel Library (TPL)
 - about 208, 209
 - Parallel.For() 210, 211
 - Parallel.Invoke() 209, 210
- Telerik JustDecompile
 - using 382, 383, 384
- test module (subsystem) 181, 182
- Test-Driven Development (TDD)
 - about 33, 64, 131
 - methodology practice 162, 163, 165, 166, 168, 169
- testing tools
 - about 137
 - Moq 152, 153, 154, 155, 156, 157
 - MSTest Framework 137, 139, 140, 141, 142, 143, 144, 145
 - NUnit 145, 146, 148, 150, 151, 152
 - SpecFlow 157, 159, 160, 162
- third-party APIs
 - testing 268, 269
- thread parameters
 - adding 207, 208
- thread pool
 - number of processors and threads, limiting 217, 218
 - using 208
- thread safety 234, 238, 239, 241, 242
- ThreadPool.QueueUserWorkItem() method 212
- threads
 - ackground threads 206
 - foreground threads 206
 - life cycle 205, 206, 207
- three-module system
 - testing, with End-to-end (E2E) 182, 183, 184, 185
- tight coupling
 - about 70
 - example 71
- time slicing 204
- time the software is functional (tsf) 358
- total time it is expected to function (ttof) 358

triadic methods 100

type initializer 228

U

unbounded priority inversion 215

unchecked exceptions 110, 111, 112, 113, 114

unit testing

 need for 131, 132, 133, 135, 136, 137

Universal Windows Platform (UWP) 368

unmanaged code (COM) 21

User Acceptance Testing (UAT) 56

W

WET code

 duplication, removing 99, 100

Windows Communication Foundation (WCF) 368

wrapper class 269

Write Every Time (WET) 99

Y

You Ain't Gonna Need It (YAGNI) 32, 34, 261