packt

1ST EDITION

Real-World Implementation of C# Design Patterns

Overcome daily programming challenges using elements of reusable object-oriented software

BRUCE M. VAN HORN II

Foreword by Van Symons, CTO Visual Storage Intelligence

Real-World Implementation of C# Design Patterns

BIRMINGHAM—MUMBAI

Real-World Implementation of C# Design Patterns

Copyright © 2022 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.

Group Product Manager: Gebin George

Publishing Product Manager: Gebin George

Senior Editor: Kinnari Chohan

Technical Editor: Jubit Pincy

Copy Editor: Safis Editing

Project Coordinator: Manisha Singh

Proofreader: Safis Editing

Indexer: Subalakshmi Govindhan

Production Designer: Roshan Kawale

Marketing Coordinator: Sonakshi Bubbar

First published: September 2022

Production reference: 1270922

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80324-273-6

www.packt.com

For my beautiful wife Karina, and my children Kitty and Phoebe, who helped me truly understand Ruth 1:16. In memory and gratitude to my mother and father, who paid for my first three computers, back when normal people didn't own them. In memory of Dr. Charles Gettys for motivating me by telling me I would probably fail, and then teaching me everything he knew about computers. For Van Symons, who saw past my wheelchair and gambled on me when nobody else would, and who gave me a place and the opportunity to do the best work of my career. For my students at Southern Methodist University and Richland College, from whom I've learned as much as I've taught. For my work family at Clear Technologies/Visual Storage Intelligence. In memory of my student, Tom, who really could type ActionScript code with his feet. These are the people who inspire me and for whom I am deeply grateful. Above all, I am thankful to my Lord and Savior Jesus Christ who makes all things possible (Philippians 4:13), along with His Father who invented all the electrons and silicon, without which none of this computer stuff would exist (Genesis 1:14–19).

Foreword

I have known and worked with Bruce M. Van Horn II for the past 10 years. Bruce has been the lead software developer for Visual Storage Intelligence, an analytical software technology for analyzing infrastructure hardware and software to ensure the optimum use of on-

premises and cloud resources is maximized. Bruce is a leader in creating and leveraging innovative software technologies to meet demanding customer needs in a fast-paced IT environment.

In this book, Bruce teaches you how to leverage software engineering techniques known as design patterns to solve real-world problems in an efficient and effective manner. Using patterns will help you solve problems effectively without reinventing the wheel. It will also keep your code healthy and stable.

Learning patterns are a must-have requirement in the career of any software engineer that we've hired at our start-ups. There is no question as to whether you should learn patterns; the question is really, where can you find a resource that teaches patterns in a way that is easy to understand and apply? There are many books on patterns in the marketplace but most have a very academic focus. This book is very different and somewhat unique. Instead of a dry treatment that reads like a doctoral thesis, Bruce presents you with the story of two sisters who form a tech start-up. Problems are encountered, requirements change, and at one point, so does the direction of the business. These are all normal and frequent occurrences, and the software that runs the company must be able to adapt. Most books cover development topics in a vacuum. All the designs are perfect, and the code always runs on the first try. That isn't real life.

The story in this book follows a more realistic arc. The characters get into jams and use patterns to get out of them. Oftentimes, knowing when, where, and how to make use of these patterns can be critical to business success, and that's what you're going to learn in this book.

Bruce is not only a brilliant software engineer but also one of the best technology educators I have met in my 45-year career. He has the unique ability to not only understand the technology but also understand both the best places to make use of it, and also the places where specific technologies are not appropriate for the issues being analyzed.

Bruce shows you how to leverage the most important Gang of Four patterns in simple-to-understand examples, as well as how to use the patterns to solve problems in your daily work. Reading this book made software design patterns come to life and helped me understand how to leverage patterns in a real-world business context.

Van Symons

CTO Visual Storage Intelligence

Contributors

About the author

Bruce M. Van Horn II is an architect and lead developer for Clear Technology's Visual Storage Intelligence SaaS product. He has over 30 years' experience writing software and over 25 years' experience teaching software development at a university level. He currently teaches full stack software development at Southern Methodist University's code boot camp. He is a **Certified ScrumMaster** (**CSM**). Ten years ago, he was diagnosed with dermatomyositis and was told he would never walk, speak, or eat nachos again. Through faith and hard work, he beat the odds and today enjoys riding bicycles with his daughters, Katherine (Kitty) and Phoebe. He eats nachos regularly with Karina, the love of his life.

About the reviewers

Alexej Sommer is a professional .NET developer with expertise in a variety of technologies, including ASP.NET Core, WPF, UWP, Xamarin, and Azure.

He is also MCP/MCSD- and Azure-certified. He's a winner of Microsoft's Most Valuable Professional 2016–2019 in the Windows Development category. At present, he sometimes participates in conferences as a speaker.

Sarita Nag was born in India and she earned her master's degree in computer science from KIIT University, Bhubaneswar, Orissa.

Sarita began her career as Software Engineer from Thomson Reuters and since then, she worked on many multinational companies and is currently working at FISERV.

She is an experienced software engineer with passion for developing innovative programs that expedite the efficiency and effectiveness of organizational success. Well-versed in technology and writing code to create systems that are reliable and user-friendly.

She has 10+ years of experience in various phases of Software Development Life cycle (SDLC) and Agile methodologies such as Analysis, Design, Development, Testing, Deployment, and Maintenance in various domains like Tax and accounting, Financial, Communication Media, and Customer relationship.

Table of Contents

Preface

Part 1: Introduction to Patterns (Pasta) and Antipatterns (Antipasta)

1

There's a Big Ball of Mud on Your Plate of Spaghetti

Technical requirements

No battle plan survives first contact with the enemy

The Stovepipe system

The Big Ball of Mud

Time

Cost

Experience

Skill

Visibility

Complexity

Change

Scale

The Golden Hammer

A throwaway code example

How can patterns help?

Summary

Questions

Further reading

2

Prepping for Practical Real-World Applications of Patterns in C#

Becoming a prepper

Technical requirements

Spaghetti < lasagna < ravioli – software evolution explained with pasta

Spaghetti code

Lasagna code

Ravioli – the ultimate in pasta code

The foundational principle – writing clean code

You should write code that is readable by humans

Establishing and enforcing style and consistency

Limiting cognitive load

Terse is worse

Comment but don’t go overboard

Creating maintainable systems using SOLID principles

The Single Responsibility principle

The Open-Closed Principle

The Liskov Substitution principle

The Interface Segregation principle

The Dependency Inversion principle

Measuring quality beyond the development organization

Code reviews

Overall design

Functionality

Summary

Further reading

Part 2: Patterns You Need in the Real World

3

Getting Creative with Creational Patterns

Technical requirements

The following story is fictitious

The initial design

No pattern implementation

The Simple Factory pattern

The Factory Method pattern

The Abstract Factory pattern

The Builder pattern

The Object Pool pattern

The Singleton pattern

Summary

Questions

Further reading

4

Fortify Your Code With Structural Patterns

Technical requirements

B2B (back to bicycles)

The Decorator pattern

The Façade pattern

The Composite pattern

The Bridge pattern

Summary

Questions

Further reading

5

Wrangling Problem Code by Applying Behavioral Patterns

Technical requirements

Meanwhile, back at the bicycle factory

The Command pattern

Applying the Command pattern

Coding the Command pattern

Testing the Command pattern's code

The Iterator pattern

Applying the Iterator pattern

Coding the Iterator pattern

Trying out the new iterator

The Observer pattern

Applying the Observer pattern

Coding the Observer pattern

The Strategy pattern

Applying the Strategy pattern

Coding the Strategy pattern

Summary

Questions

Part 3: Designing New Projects Using Patterns

6

Step Away from the IDE! Designing with Patterns Before You Code

Technical requirements

A bad day at the agency

Bumble Bikes factory – Dallas, Texas

A physical rehabilitation clinic – Dallas, Texas

Designing with patterns

The first pass

The seat

The frame

Wheels and casters

The motor for the powered chair

The steering mechanism for the powered chair

The battery for the powered chair

The track drive system for the Texas Tank

Adding patterns

The first design meeting

The second pass

The Builder pattern

The Singleton pattern

The Composite pattern

The Bridge pattern

The Command pattern

Summary

Questions

Further reading

7

Nothing Left but the Typing – Implementing the Wheelchair Project

The crack of noon

Setting up the project

Wheelchair components

Finishing the wheelchair base classes

Finishing up the composite

Implementing the Builder pattern

Another refactor

Adding concrete component classes

Wrapping up the Builder pattern

Adding the Singleton pattern

Painting the chairs with the Bridge pattern

Summary

Questions

Further reading

8

Now You Know Some Patterns, What Next?

Patterns we didn’t discuss

Prototype

Adapter

Flyweight

Chain of Responsibility

Proxy

Interpreter

Mediator

Memento

State

Template Method

Visitor

Patterns beyond the realm of OOP

Software architecture patterns

Data access patterns

Creating your own patterns

Name and classification

The problem description

The solution description

Consequences of using the pattern

Not everybody likes patterns

Summary

Sundance Square – Fort Worth, Texas

Further reading

Appendix 1

A Brief Review of OOP Principles in C#

Technical requirements

A quick background of C#

C# is a general-purpose language

C# is purely and fully object-oriented

C# uses a static, strong type system

C# has automatic bounds checking and detection for uninitialized variables

C# supports automated garbage collection

C# code is highly portable

Language mechanics in C#

Variables in C#

Classes

Encapsulation

C# auto-implemented properties

Accessor logic with backing variables

Inheritance

Interfaces

Defining interfaces

Implementing interfaces

IDEs for C# development

Visual Studio

VS Code

Rider

Summary

Further reading

Appendix 2

A Primer on the Unified Modeling Language (UML)

Technical requirements

The structure of a class diagram

Classes

Interfaces

Enumerations

Packages

Connectors

Inheritance

Interface realization

Composition

Association

Aggregation

Directed association

Dependency

Notes

Best practices

Less is more – don’t try to put everything in one big diagram

Don’t cross the lines

The most direct path for lines leads to a mess

Parents go above children

Keep your diagrams neat

Summary

Further reading

Index

Other Books You May Enjoy

Preface

This is a book about design patterns written in the context of the C# coding language. I know what you're probably thinking. What does that even mean? Well, I could tell you but I'd spoil the rest of the book, and trust me, it's a doozy!

Design Patterns are a codified set of best practices for software problems that come up so often that we can learn to recognize them and immediately know how to solve them. The solutions for these recurring problems found in patterns have been used for decades and they have proven to be effective.

Patterns also become a battle language for developers because they are so pervasive. This idea comes from the popular television and movie franchise, *Star Trek*. In *Star Trek*, the warrior race known as the Klingons has two languages. They have the regular Klingon language that they learn in Klingon kindergarten and an abbreviated version they use during combat. The phrase "*Load torpedo tubes 1 and 2 and fire a full spread*" can be reduced to one or two words. All Klingons know what that phrase means, and they win battles because they are a few seconds faster than their linguistically-challenged foes. Similarly, you can say "*Just use the decorator pattern*." Any developer who has studied patterns will understand what to do next.

Patterns are not specific to the C# language. However, in order to learn patterns effectively, you need an implementation language. The original book on patterns was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, collectively known as the **Gang of Four**, or **GoF** for short. Their book, *Design Patterns: Elements of Reusable Object-Oriented Software*, is often referred to as *the GoF book*.

The GoF book was published in 1994, which makes it a technological dinosaur. While it may be old, the patterns explained within its pages are still very relevant. My criticism of the GoF book is it is written in a very academic format, and its implementation language is one you don't really see anymore. The implementation language is an important detail. The GoF book covers 23 patterns, which is the same number of herbs and spices found in KFC's original recipe. That can't be a coincidence. This book doesn't cover all 23 patterns, which is fine, because too much fried chicken isn't very good for you. I've focused on patterns you'll need every day in most of the projects you work on. I cover the remainder briefly in *Chapter 8* at the end of the book.

There are some books out there that try to use pseudocode in order to be generic. I posit that such books are not useful to most people. If you're like me, and you probably are in this regard, you'd wish for a code example that is easy to read, isn't overly clever, and has a firm basis in the real world. Books and blog sites that try to teach you pat-

terns with phrases such as "`Class A` inherits from `Class B`, which depends on `Class C`" are too vague to be useful. Equally annoying are books and sites that try to show you the pattern in 20 languages. They usually do a poor job with all 20 instead of focusing on doing a good job in just one. I am only using C# and the **Unified Modeling Language** (**UML**).

If you've never heard of UML before, don't let it scare you. UML is a convention used to create diagrams. There are 14 types of UML diagrams. I only use one diagram type: class diagrams. I've included a primer in *Appendix 2* to help you if you are new to UML.

This book is about the real world, or at least a close facsimile. I've written this book using the same techniques that are used in real software projects. There's realistic code in the book that solves legitimate business problems. There are also design mistakes included, along with thoughtful ways out of trouble.

Another issue with academic books is that they are wordy and hard to read. I have strived to take a dry subject and avoid being boring. I realize that nobody intentionally writes a boring book. Unfortunately, many have succeeded. I think the main cause of this boredom is how a lot of authors in technology view the task of writing a book. I think a lot of authors write books to prove how smart they are. These tend to be very academic. They are built to impress other academics. That's

great! The world needs academics. Most developers are not professional academics. I dare say, a great many developers have never taken a formal CS course. My aim in writing this book isn't to prove I'm smart or capable. My wife would tell you that I am obnoxiously self-deprecating. Instead, my aim is to help you over the wall that is keeping you from moving to the next level in your coding. I had to climb this wall by myself, and for me, it wasn't easy. However, if I can do it, you can too, with my help.

Instead of a dry academic treatment, this book presents a story that could actually happen in the real world. There's just enough light science fiction involved to make it "just a story." The circumstances in the story, though, are very real, and you'll recognize them, even if you have a few years in the field.

On that point, I want to make a few things clear. The story in this book is fiction. While I am a **fantastic** software engineer, I am not a roboticist or a mechanical engineer, nor am I a qualified bicycle mechanic. At several points in the coming story, you may need to suspend your disbelief if these subjects are in your wheelhouse.

Who this book is for

This book is for anyone who wants to become a better software developer. I wish I could get away with just saying that, but I probably can't. Let me add a few thoughts on who will benefit from this book.

The easy answer is what I would call *mid-level* developers. These are developers who have a few years of experience with C# and are very comfortable with the basic principles of object-oriented programming. Ideally, you've seen a few UML class diagrams.

Another beneficiary of this book is a student who is learning C#. If you're even remotely comfortable with basic OOP concepts, such as inheritance and composition, and you know your way around an **integrated development environment** (**IDE**), I want you to read this book. Sure, the mid-level developer might have an easier time, but learning patterns and SOLID principles will give you a strong foundation. You might avoid picking up some bad habits, or correct those you've already learned.

I also encourage you to read this book if you're a recent graduate of either a university or a code boot camp. If you haven't done a lot of C# work, but you've worked in other languages such as Java, C++, Python, or JavaScript, you are invited to read this book as well. I've tried to give you a bit of a boost by including a lengthy primer on C#

and object-oriented programming concepts in *Appendix 1* of this book.

There is one group I want to reach most of all.

I especially want to encourage self-taught developers like myself. Those in this camp tend to have learned only what is absolutely necessary as a means of surviving your current sprint. If your teachers are YouTube and the blogosphere, chances are you will readily recognize the anti-patterns found in *Chapter 1*, because by now, you've probably committed every sin relevant to software engineering. I only know this because I have too. As such, I know you stand to benefit the most from reading this book.

As I said, this book is for anyone who wants to become a better software developer. I guess I should have just stuck with that.

What this book covers

Chapter 1, *There's a Big Ball of Mud on Your Plate of Spaghetti*: Before we dive into patterns, let's dive into why we need them. The world of software development is very messy, but it doesn't have to be. The mess comes from a set of degenerative forces in our work that you will surely recognize.

Chapter 2, *Prepping For Practical Real-World Applications of Patterns in C#*: To defeat the degenerative forces mentioned in *Chapter 1*, you're going to have to step up your game. This chapter presents some rules and principles. If you can adhere to them, you will have the discipline needed to use design patterns to their greatest effect.

Chapter 3, *Getting Creative with Creational Patterns*: Now that you are thoroughly prepared, this chapter introduces our story. It covers patterns designed to make the instantiation of your classes more robust and more flexible. After reading this chapter, you'll never look at the `new` keyword the same way.

Chapter 4, *Fortify Your Code with Structural Patterns*: This chapter covers techniques you can use to structure your classes for maximum flexibility while honoring the SOLID principles covered in *Chapter 2*.

Chapter 5, *Wrangling Problem Code by Applying Behavioral Patterns*: Got algorithms? You need a flexible set of patterns in order to maximize their effectiveness and flexibility. You need behavioral patterns.

Chapter 6, *Step Away from the IDE! Designing with Patterns Before You Code*: In this chapter, we consider ways to design our code with patterns before we write a single line in our IDE. After an unfortunate turn of events in our story, we find our company drastically and rapidly changing direction. We need a new product design, and we need it last week! Let's draw our designs in UML first! This saves a lot of time and energy and prevents the possibility of some pointy-haired boss telling us to ship a prototype.

Chapter 7, *Nothing Left but the Typing – Implementing the Wheelchair Project*: In the last chapter, we came up with an elegant set of design diagrams. In this chapter, we do the typing. You'll implement the same patterns you learned earlier in the book, but this time, you'll use them in concert with each other on a real-world project.

Chapter 8, *Now You Know Some Patterns. What Next?*: We've had a lot of fun learning patterns so far, but this is only the tip of the iceberg. There are patterns everywhere! They aren't limited to the practice of OOP. In this chapter, we cover the GoF patterns we didn't cover in our story.

Chapter 9, *Appendix 1* – *A Brief Review of OOP Principles in C#*: This appendix is designed for those who are new to C# or maybe haven't used it in a while, or are coming from another language.

Chapter 10, *Appendix 2* – *A Primer on the Unified Modeling Language*: The Unified Modeling Language is a documentation convention used by software developers. It defines the structure of the pattern design diagrams used throughout the book. While UML has 14 different diagram types, we really only use class diagrams. Most presentations on patterns have two diagrams. I draw a generic one, and a second diagram that mirrors the project code. This appendix shows you the conventions used in the diagrams.

To get the most out of this book

To get the most out of this book you should be familiar with C#. You need to be competent in using one of the three popular IDEs: Visual Studio, Rider, or Visual Studio Code. You should also understand basic object-oriented programming principles such as abstraction, inheritance, encapsulation, and composition.

I don't spend very much time covering how to use your IDE in this book. However, I do include *Appendix 1*, which covers how to create a project just in case you're rusty. This book isn't designed to be a step-by-step guide through a set of projects. The code in the sample

projects doesn't matter. We're focusing on the structure of the code, rather than the content of the classes.

The projects in this book are all either command-line or library projects. We won't be working with any frontend or user interface code. This is done to reduce the level of noise in the projects. I want you focused on the structure of the classes, not what is inside them, nor even what the program is really doing.

I used Windows 10 to create the code in this book. If you want to follow along with the code in the book, you can probably use macOS or Linux. However, I don't cover those operating systems explicitly, nor do I test the sample code in operating systems other than Windows.

If you intend to code along with our book's heroes, you'll need to set up your computer with an appropriate IDE, and .NET Core 6 or later. I used Rider as my IDE, but I verified the code in Visual Studio 2022 and Visual Studio Code.

Software/hardware covered in the book	**Operating system requirements**
C# 10	Windows
.NET Core 6	Windows

Rider, Visual Studio, or Visual Studio Code	Windows

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

I strongly encourage you to type the code from the book by hand. You'll learn more by typing it out, by making mistakes, and then fixing them yourself.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/Kpackt/Real-World-Implementation-of-C-Design-Patterns. Please note that GitHub won't allow us to use the # character in C# so the name of the repository is slightly misleading. If there's an update to the code, it will be updated in the 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!

This book has a companion website developed by the author. You can find it at https://csharppatterns.dev.

Download the images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://packt.link/3KWzG.

Conventions used

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

`Code in text`: 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: “Kitty starts by creating an abstraction for what the builders will be producing – that is, the product. She creates an interface called **`IBicycleProduct`**.”

A block of code is set as follows:

```
public interface IBicycleProduct
{
public IFrame Frame { get; set; }
public ISuspension Suspension { get; set; }
```

```
  public IHandlebars Handlebars { get; set; }
  public IDrivetrain Drivetrain { get; set; }
  public ISeat Seat { get; set; }
  public IBrakes Brakes { get; set; }
  }
```

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

```
  namespace WheelchairProject;
  public abstract class WheelchairComponent
  {
```

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

```
  $ dotnet build
  $ dotnet run hillcrest
```

TIPS OR IMPORTANT NOTES

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, email us at customercare@packtpub.com and mention the book title in the subject of your message.

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 and fill in the form.

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.

Share Your Thoughts

Once you've read *Real-World Implementation of C# Design Patterns*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we’re delivering excellent quality content.

Part 1: Introduction to Patterns (Pasta) and Antipatterns (Antipasta)

We will begin *Part 1* by describing what patterns are and where they came from and giving an overview of how they work and why you'd want to learn them.

This part covers the following chapters:

- *Chapter 1, There's a Big Ball of Mud on Your Plate of Spaghetti*
- *Chapter 2, Prepping for Practical Real-World Applications of Patterns in C#*

1

There's a Big Ball of Mud on Your Plate of Spaghetti

Welcome to what is potentially your last day on the job. Your project is about to be canceled. Your customers are angry. Your boss is freaking out. Your boss's boss is in Antigua, but when she comes back next week, heads will probably roll. There's no way to sugar-coat it. You might want to update your résumé and brush up on your algorithms, so that you're ready for an imminent job search.

How did it come to this? We had a plan. The hardware architecture was simple. The first few releases went off without a hitch and our users were delighted. Our client even presented a whole new set of feature requests and signed a contract extension. How have we found ourselves at the precipice of sure and sudden doom?

The situation we have found ourselves in here is far from unique. According to many academic accounts, five out of six software projects are canceled. Others fail by running behind schedule or over budget. Software projects are hard. There's no such thing as a *simple* program. There's really no project that you can knock out in a week, then

ship, and that's the end. It doesn't work that way. This phenomenon is unique to the software industry. Structural engineers who design bridge are pretty much done when the bridge opens for public traffic. Electrical engineers design and test circuits on breadboards, then hand those designs off to be manufactured. Aeronautical engineers, such as my grandfather, who designed power plants (as in, engines) for Beechcraft, generally designed and prototyped engines, but didn't do much beyond that. It was up to others to manufacture the engine, mount it on the aircraft, and others to maintain the engine.

In contrast, software engineers must design, build, test, and often maintain the systems that they develop in a continuous delivery environment. Many projects are never "done." I've been working on the same software project for the last 9 years. We certainly haven't built the perfect project with perfect architecture, but the project has endured. New features are produced, and bugs are discovered and fixed.

What differentiates projects that run continually for many years from the vast majority that get canceled? While there are many ways this can happen, we're going to focus solely on the design and architecture of our software. I'll start with how it very often goes wrong. In keeping with the title of this book, we'll spend some time in this chapter discussing a set of antipatterns. While I haven't directly introduced the concept of patterns yet, I suspect that you can make an educated

guess about what they are, along with what their antithesis is. A **pattern**, for now, is simply a formally explained, abstract, best practice solution to a common development requirement. An **antipattern** is a formal example of what you shouldn't do. Patterns are arguably good. Antipatterns are inarguably bad.

This chapter will present some of the most common antipatterns, including the following:

- Stovepipe systems
- The Big Ball of Mud
- The Golden Hammer

Once we've learned about a few antipatterns, we'll focus in later chapters on principles and patterns designed to combat and correct the circumstances where antipatterns have either taken hold or might soon take hold. In *Chapter 2*, *Prepping for Practical Real-World Applications of Patterns in C#*, I will prepare you for your work with patterns. Software development is an odd business in that we all come to it traveling different roads. I am personally self-taught. I started when I was 12 years old. The only books about computer programming were available for purchase at Radio Shack. There were about a dozen. We didn't have resources such as **Packt Publishing** plying the market with fascinating and useful books on every facet of software development. In 1991, the year that I graduated from university,

a computer science degree would have focused on software development for mainframes using FORTRAN, which is a far cry from the work I do now. The mainframe programming course I took in 1987 was the last class to use punch cards. If you're not sure what they are, go look them up. I'll wait. Are you back? Are you horrified? Me too. The point is, there are a lot of people such as me out there who learned programming out of necessity, and they learned informally.

There are many university-trained software developers out there, but not all software development programs are the same. Computer science programs focus on elements such as mathematical theory and algorithm development but teach only a minimal amount of practice. Software engineering programs, boot camps, and trade schools have more of an engineering focus, where you learn to build software with less of a focus on the theory. Regardless of where you started, *Chapter 2* aims to ensure that you understand the most important formal engineering concepts needed in order to work with patterns. Patterns were created using a set of rules and *Chapter 2* covers those rules.

In *Chapters 3*, *4*, and *5*, we cover patterns in earnest using a story format. I've done this in the hope of creating a learning experience very different than my own, having read some of the more heavy-handed academic treatments of design patterns. The patterns that I have selected for this book come from what is perhaps the seminal work on

patterns in the software industry, *Design Patterns: Elements of Reusable Object-Oriented Software*, by Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. These four authors are known collectively as **The Gang of Four** (**GoF**), and the book that they wrote is colloquially referred to as *The GoF book* or just *GoF*. *The GoF book* contains 23 patterns, broken down into three categories:

- **Creational patterns** deal with the creation of objects beyond simply using the new keyword.
- **Structural patterns** deal with the way that you structure your classes to maximize flexibility, reduce tight coupling, and help you focus on reusability.
- **Behavioral patterns** deal with how objects interact with one another.

The creational patterns I'll be covering in *Chapter 3* include the following:

- The **Simple Factory** (technically not a pattern)
- The **Factory Method pattern**
- The **Abstract Factory pattern**
- The **Builder pattern**
- The **Object Pool pattern**
- The **Singleton pattern**

Within the realm of structural patterns, in *Chapter 4*, I'll be covering the following:

- The **Decorator pattern**
- The **Façade pattern**
- The **Composite pattern**
- The **Bridge pattern**

Practical pattern coverage will conclude in *Chapter 5* with this set of behavioral patterns:

- The **Command pattern**
- The **Iterator pattern**
- The **Observer pattern**
- The **Strategy pattern**

Again, I'll point out that this book is designed to focus on real-world software development. In the real world, we don't always perfectly follow the rules presented in *Chapter 2*. Many books present a perfect experience. Expert authors always present all of their examples as perfect on the first try. I won't be doing that because that isn't in keeping with reality. As we flow through *Chapters 3*, *4*, and *5*, we will find ourselves faced with "gotchas" that you will find in actual practice. There's no way around them. Even if you execute the patterns and strategies in these chapters perfectly, nobody can foretell the future.

In *Chapter 6*, a plot twist in our story arises and we have a chance to rethink everything that we've done up to that point. In *Chapter 6*, we'll design a new system based on an old one using the patterns that we've learned so far. *Chapter 6* is all about creating a design and a plan. In *Chapter 7*, we implement that plan.

I won't be covering every pattern covered by the GoF. Instead, I'll be focusing on the patterns you are most likely to need as a C# .NET software developer working in the field. I've selected my list of patterns based on popularity, usefulness, and complexity. Complex patterns that are not often seen in the wild have been omitted from the main text of the book. That said, I do circle back in *Chapter 8* to give you a rundown on the patterns that I didn't cover, and the usual advice on where to go from there.

This book assumes that you have a few years of experience working with C#. In addition to my day job, I have taught software development at colleges for the last 25 years. I presently teach at Southern Methodist University's Full Stack Code Bootcamp. Some of the programs I taught focused on C#, while others haven't. At SMU we teach JavaScript. If you're coming to this book without recent C# experience, or perhaps with none at all, I've added *Appendix 1* at the end of the book. It is designed to give you what I hope is enough orientation in the C# language to make the rest of the book useful. The truth about patterns is they are language-agnostic. They apply to any ob-

ject-oriented language, and I've even seen some of them shoehorned into languages that are not object-oriented with arguable levels of success.

Technical requirements

This chapter presents some code examples. Most books always present exemplary code that makes sense for you to follow in the creation of the project. The code in this chapter is terrible on purpose. It isn't strictly necessary that you follow along by creating the project, but you're welcome to do so if you'd like.

If this is the case, you'll need the following:

- A computer running the Windows OS. I'm using Windows 10. Since the projects are simple command-line projects, I'm pretty sure everything here would also work on a Mac or Linux, but I haven't tested the projects on these operating systems.
- A supported IDE, such as Visual Studio, JetBrains Rider, or Visual Studio Code with C# extensions. I'm using Rider 2021.3.3.
- Some version of the .NET SDK. Again, the projects are simple enough that our code shouldn't be reliant on any particular version. I happen to use the .NET Core 6 SDK and my code's syntax may reflect that.

- An instance of SQL Server and basic knowledge of SQL. I want to restate that the code in this chapter is designed to be a realistic example of throwaway code. C# and SQL Server go together as peanut butter and jam do, which adds to the realism. Some readers may not be comfortable working in SQL Server, especially without **Entity Framework** (**EF**) used for its presentation. This is the only place in this book where a database is even mentioned. If you have no experience with databases, don't worry. The example is really meant to be read more than tried. If you want to try it, any version of SQL Server should work. I'll be using SQL Server 2019.

You can find the code files for this chapter on GitHub at https://github.com/Kpackt/Real-World-Implementation-of-C-Design-Patterns/tree/main/chapter-1/.

No battle plan survives first contact with the enemy

There's an old saying: *if you fail to plan, you plan to fail*. Only the rankest amateur would dive into a project IDE-first without at least considering how the project ought to be structured. The typical first steps might involve roughing out a package and object structure, or maybe designing the structure of a relational database that will persist the data used by our software. Someone who's got a few projects

under their belt might even draw some diagrams using the **Unified Modeling Language** (**UML**).

We begin by taking a set of user stories and we shape our code into something that on the surface meets the requirements in front of us. Soon, we're in an agile groove swing. We've achieved velocity! We create a feature, show it to the customer, get feedback, revise, and continuously deliver. That's usually how the troubles begin.

Our first major anti-pattern, the stovepipe system, comes from the seminal book on the subject, *AntiPatterns*, by Brown, et al., which I've listed as suggested reading at the end of this chapter.

The Stovepipe system

Once upon a time, in just about any industrialized society, people heated their homes and cooked using a cast-iron *potbelly* stove. These stoves burned coal or wood for fuel. Over time, the exhaust vent for the stove, called the stovepipe, since it was literally a pipe sticking out of the stove, would build up with corrosive deposits, which led to leaky stovepipes. The fumes from a burning stove are potentially life-threatening within a small, enclosed space.

Here's what an actual stovepipe looks like:

Figure 1.1 – A stove with a stovepipe.

The stovepipe required constant maintenance to prevent asphyxiation. This was usually done by the owner of the stove, who was unlikely to be a stove repair professional. The stove was repaired using

tools and materials that were readily available. This made for very ad hoc patch jobs, rather than clean, well-thought-out repairs done with **original equipment manufacturer** (**OEM**) grade materials and the proper tools.

Now, think about how this might relate to a software project. The initial release is designed with great care, with an implementation that perfectly matches the design. The natural tendency during software maintenance is to fix things quickly and get the patched version released and out the door. As with our amateurish stove repairs, our analysis of the holes in the software design and implementation are perfunctory and incomplete. There is pressure to solve this quickly. Everybody is watching you. Every minute the application is *down* costs the company money and you risk losing the *employee of the month* parking spot.

This happens to everyone and everyone generally caves to human frailties. You then implement the quickest, easiest thing you can think of – it's done and the patch is out the door. The crisis is over.

Or is it? Dun dun dun! Small ad hoc fixes have a negative cumulative effect over time, referred to as technical debt, just as the corrosive deposits on a stovepipe do. How can you tell whether the systems that you're working on are stovepipe systems? Let's explore the following:

- Stovepipe systems are monolithic by their very nature. It is not easy to get data in or out of this kind of system, and integrating software built this way into a larger enterprise architecture is cumbersome or impossible.
- Stovepipe systems are very brittle. When you make one of these small ad hoc repairs, you generally find that the fix breaks other parts of the application. Usually, this isn't discovered until after the breaking fix has been released.
- Stovepipe systems can't be easily extended as new business requirements emerge. When a project starts, you're given a set of requirements. You build the software that meets those requirements. After it's released, a new feature is requested that you couldn't possibly have predicted. You realize that there's no way to implement that feature without redesigning the whole app. Anytime you're tempted to throw out the baby with the bathwater and just start over, you're working on a stovepipe system.
- Stovepipe systems built on component architectures are generally incapable of sharing those components with other enterprise applications. The level of code reuse between projects is very low.
- Stovepipe systems are often found on projects with high turnover. This makes sense. You start a new job, replacing the last developer, and you feel pressure to get something working quickly to show your new boss that hiring you wasn't a huge mistake. You do your best to piece something together to fix a problem. You have no

knowledge of the existing architecture or what's been tried, and perhaps failed, in the past. Now amplify this by considering two or three further re-staffing efforts, with several months between each new hire's start dates.

- Stovepipe systems are often indicated when the development team is using new or unfamiliar technologies, stacks, or languages. Given that the same pressure to produce something quickly exists, while the team is simultaneously required to work with tools and languages that they've never used before, this leads to the same pattern of *just getting something working and released*. You will also encounter stovepipe systems in start-ups, corporate acquisitions, and mergers for these same reasons.

Does any of this sound familiar? Naturally, we're not talking about anything you've ever written! Isn't this just a little bit reminiscent of code you've seen other people write? Maybe your competitors? Maybe your students? Even if you own up to writing a stovepipe system, don't beat yourself up. It is far and away the most popular pattern in software development today. Sometimes, a stovepipe system is fine. Remember, not every physical edifice needs to be supported by fluted ivory columns, and there's a very legitimate argument to be made for getting the software to market and worrying about the rest later. However, if your objective is to build software that's still useful and profitable 10 years or more down the road, keep reading. We'll

have those stovepipes replaced with functional, modular, well-constructed systems in no time.

The Big Ball of Mud

Around the same time Brown, et al. were writing their book on antipatterns, another research team was engaged in a similar effort. The product of their work was titled *Big Ball of Mud*, by Foote and Yoder (1997), which I've listed in the *Further reading* section at the end of this chapter. It's their work that inspires the title of this chapter.

The Big Ball of Mud antipattern is remarkably similar to our outline of stovepipe systems. However, the authors go into greater depth on how these systems understandably come to be.

They often start with **throwaway code**. I think this is self-explanatory. It's code that you knock out in a few hours, or even a few weeks, that serves as a rough prototype. It proves to you and perhaps your stakeholders that the problem in front of you is soluble. It might even be good enough to demonstrate to the client. This is where the trap is sprung. The prototype is good enough to publish, so, at the behest of your boss, you do. We'll simulate this later in this chapter in a section titled *A throwaway code example* by intentionally building a prototype that is good enough to ship, but not good enough to survive extension. This rough prototype will do everything asked of us. Right after

this imaginary software project ships its first release, we'll then find ourselves faced with a second factor in the construction of a Big Ball of Mud: **piecemeal growth**. A project manager might refer to this pejoratively as *scope creep*. I have been a consulting software engineer and a corporate software engineer. I can tell you the project management view of scope creep as being something negative is misinformed. While it is a source of frustration from a planning and billing perspective, new requirements coming in after the initial release are the hallmark of a successful system. It is my strongest advice that you begin every project with the idea that it will be wildly successful. This may seem overly optimistic, but it is in fact the worst-case scenario if you have published throwaway code.

Piecemeal growth leads to a strategy called **keeping it working**. Again, this needs little explanation. As bugs and new features are identified, you just fix the offending bits and make the program satisfy the new set of requirements. And, oh, by the way, we need this done by next week.

After the second release, *keeping it working* becomes your daily job description. If your program is really successful, you will start to hire people to help you with *keeping it working*, which naturally amplifies the problems and technical debt as the project continues to grow.

To reiterate, this sounds very similar to our elucidation of a stovepipe system. Foote and Yoder consider in more detail the forces that lead to our unfortunate muddy circumstances. Forces, as in nature, are outside actors that you can rarely ever control. These forces consist of the following:

- Time
- Cost
- Experience
- Skill
- Visibility
- Complexity
- Change
- Scale

Let's talk a little more about each.

Time

You may not be given enough time to seriously consider the long-term ramifications of the architectural choices that you're currently making. Time also limits your project by limiting what you can accomplish with what you're allotted. Most developers and project managers try to get around this by padding their estimates. In my experience, Parkinson's law is true: projects where time estimates are

padded, or even doubled, usually expand to fill or exceed the time allotted.

Cost

Most projects don't have an infinite budget. Those that do are open source projects that have no monetary budget at all, instead substituting it with the time of volunteers, which is itself a cost. Architecture is expensive. The people with the knowledge and experience to develop a sound architecture are rare, though slightly less inaccessible given that you are reading this book. They tend to draw higher salaries, and the expense involved in the effort to create and maintain proper architecture doesn't pay off immediately in the minds of your stakeholders, bosses, or customers.

Good architecture requires time, both on the part of the development staff and the architect, but also domain experts who know the business behind the software. The domain experts are rarely dedicated to the software development effort. They have regular jobs with real requirements and deadlines outside the software project. Involving a business consultant who bills at $250 USD per hour is eating up time that could be billable, but you honestly can't complete the project without this access.

Experience

Software developers are experts in software development. They rarely have expertise in the business domain where they are building solutions. For example, it's rare that someone building a system that quotes insurance policies has worked as an actuary or even an adjuster. Lack of experience in the business domain makes the job of modeling software a process of trial and error, which naturally affects the program's architecture.

Skill

Not all software developers have equal levels of skill. Some are new to the field. Some are slower learners than others. Some learned to use a few golden hammers (more on this later) and refuse to upskill any further. And there's always a superstar on the project that makes everybody else feel as though they're a poser.

Visibility

You can't see inside a working program. Sure, you can fire up a debugger, but normal people can't look around the architecture of your code the same way that they can inspect the architecture of a physical structure such as an office building. For this reason, architecture is neglected. Your boss will not likely give you a fat bonus for your amazing abstractions and interface structures. They will, however,

reward you for shipping early. This leads to a very human, lackadaisical attitude toward how your code is structured.

Complexity

Complex problem domains beget muddy architectures. Imagine modeling a collection of modern light bulbs. That's pretty easy. Properties such as wattage, light output in lumens, and input voltage jump out at you as if they are second nature. Now, imagine modeling a light bulb in 1878. You're in uncharted territory. Thomas Edison patented his first light bulb in 1879 and is famously quoted as saying that he had discovered two thousand ways to not build a light bulb. If the domain is complex or unexplored, you should expect a bumpy ride as far as your architecture is concerned.

Change

Change is the one thing that always remains constant. Foote and Yoder wrote that when we devise an architecture, it is based entirely upon a supposition: a set of assumptions about the future wherein we expect changes and extensions to that architecture to be bound only to the realm of possibilities that we have considered so far. This is all well and good, except another truism invariably surfaces: no battle plan survives first contact with the enemy. The change requests will come in the most inconvenient form, at the most inconvenient time,

and it's your job to deal with that. The easy way out is always the most palatable to the stakeholders but it is what leads to a Big Ball of Mud.

Scale

Creating a system to be used by 100 people in total is a very different problem than creating a system that can process $10{,}000$ requests per second. The style in which you write your code is different. Your reliance on highly performant algorithms is largely absent in a small system, but vital to the success of a large one. Rarely do projects start at the scale typically considered by Google or Amazon. Successful projects must be able to scale up according to how successful they become.

The Golden Hammer

Another important antipattern you should learn to recognize is generally a product of some marketing organization or salesperson in a company outside your own. It happens when some killer app, framework, infrastructure component, or tool is presented as the panacea for all your software development woes. It slices, it dices, it makes julienne fries, and it automatically refactors itself while speeding up the execution of your code.

The antipattern is described as the **Golden Hammer**. Behold a fully-rendered CGI representation in *Figure 1.2*:

GOLDEN
HAMMER
Van Horn

Figure 1.2 – When you're given a golden hammer, everything is a nail.

Silicon snake oil salespeople will visit you, take you out to someplace fancy, and try to convince you that the database tool, platform, or **whatever-as-a-service** (**WaaS**) that they're selling can be the entire basis for your company's software. Consider Microsoft SQL Server for a minute. At its most basic, SQL Server is a relational database. It stores your data in tables that you can query. Related tables of data can be joined and filtered allowing a developer who understands the **Structured Query Language** (**SQL**) to produce reporting data in any format or configuration. This is a common functionality found in every relational database tool from *Microsoft Access* and *SQLite* to *Oracle* and *Microsoft SQL Server*. Since SQL is a standardized language, offering this basic functionality is little more than *table* stakes. Just so we understand each other, all puns are intended.

So, how could Microsoft expect to charge money for something that you can get for free in open source offerings such as *MySQL* and *PostgreSQL*? Granted, SQL Server got started before they did when there were fewer rivals in the marketplace, but SQL Server is one of the most popular platforms for managing data today. This is because SQL Server's value contribution doesn't end with tabular data storage. As the product has grown over the years, new features and ancillary tools have been added. You have the ability to load and ana-

lyze data in novel and sophisticated ways using *SQL Server Analysis Services*. *SQL Server Reporting Services* allows you to create reports using SQL and then present those reports graphically to whoever might need them by emailing the reports as PDFs. It also allows users to access the report on the server and play around with the data without needing to know SQL or have access to the underlying code.

There are supported workflows for working with AI and machine learning projects using R and Python, and you can make bits of code in C# that process in the database such as a native stored procedure. *SQL Server Integration Services* allows you to ingest and publish data to a variety of different databases, software services, and industry formats. This leads to the ability to integrate your software and services with your business partners and customers.

In short, if you tried hard enough, you could probably write a great deal of an application, if not an entire one, solely using SQL Server's ecosystem. SQL Server is the Golden Hammer. Every problem now looks as if it's something that can be solved with SQL Server. I want to point out that I am not vilifying SQL Server. It's a reliable and cost-effective set of tools. I go out of my way to recommend it at parties and my advice is always well received. Note to self: find better parties. I picked on SQL Server because I've seen it happen with this particular tool. If you spend too much time reading the marketing ma-

terial for SQL Server, it would be easy for you to walk away with the same conclusion: that SQL Server is all you need. Maybe it is, but you should only make that decision after you understand the Golden Hammer antipattern, lest you wind up painting yourself into a technological corner.

The Golden Hammer also emerges when a developer learns about some technology that was unknown to them before. They use it. They like it. They're rewarded for it in the form of fast or novel solutions to a problem. Since that worked out so well, and since they've gone to the effort of adding a new skill to their skill set, they try to use that tool or technique to solve every problem that they encounter.

Once, I took over a project that was in trouble. The lead programmer on a small team was being let go and most of his team left shortly after I replaced him. Interpersonal drama aside, I set out to understand the new project and the business domain by going through the existing code base.

I asked around and, as it turned out, the original staff on the project were with a consulting firm. The firm sent a couple of their ace developers over to meet and gather requirements. Upon seeing the prototypes that the client had themselves produced in Excel, using **Visual Basic for Applications** (**VBA**), the consultants concluded that they

could produce *real code* in a *real language* and have a fully converted program running in under a month.

Two years went by with no usable deliverables. The ace developers either grossly underestimated the prototype they were working with or overestimated their capabilities. I think it was a little of both. Most developers look down their noses at VBA. I'll admit that I used to, even though I've written quite a bit of VBA code. The consultants erroneously concluded that VBA is simplistic. They believed anything written in VBA would involve a trivial amount of effort to convert to a language as powerful as C#, backed by the equally powerful .NET Framework and SQL Server.

After a few months with very little progress, the consulting firm pulled their ace developers off the project to work on something else and the project was staffed entirely by junior developers.

Given the antipatterns we've covered so far, you can already see where this story is headed. I inherited this code after two and half years without a viable release. As I went through the existing code, I was able to see where the junior developers had encountered some tool or technique. It was as if I was looking at rings on a tree:

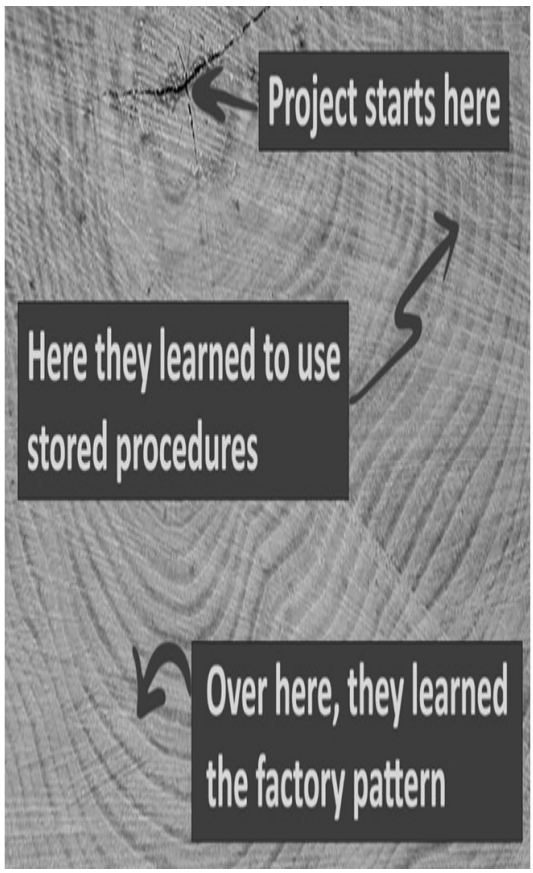


Figure 1.3 – The effects of new developers discovering a golden hammer are analogous to tree rings in their code.

You can tell exactly where they have learned that a stored procedure can be used in SQL Server because, from that point forward, the business logic suddenly moved out of the code and into the database. This is usually a bad idea. It's often done because you can change the business rules without compiling and publishing a new executable, allowing you to make minor or major adjustments. This is roughly akin to working on the engine of an airplane while it's flying at 1,261 knots (about 1,453 mph or 2,336 km/h) at 30,000 feet (9,144 m).

Somewhere else, you can tell they have read a book on patterns because the code changes. Suddenly, everything has interfaces and uses the factory pattern, which we'll cover later in *Chapter 3*. Some of this was good. I could see that they were improving. However, a lot of it was someone picking up a new hammer and using it to bang everything around it into the shape of something useful. This was evident mainly because they were never given a chance to go back and refactor their earlier work. They used different techniques at different points in time. They weren't always using the best tool for the job, but they were motivated by the forces we discussed earlier. They did their best with what they had, as with our stovepipe repair jobs.

A throwaway code example

Let's take a look at some throwaway code. Remember, throwaway code is written quickly, with little thought to architecture. It's good enough to ship but not good enough to survive extension. Consider a program designed to ingest log data from a popular web server, and subsequently analyze and present salient information in the form of a report rendered in HTML. You will be analyzing logs from NGINX (pronounced 'engine-ex'), one of the most popular web server programs in use today. I usually write a user story in an issue tracker, but I'll write it as a Markdown file in lieu, and I'll include it with my project so that I have a record of my requirements:

```
As an IT administrator, I would like to be able t
GIVEN: I have a log file from NGINX on my compute
GIVEN: I have opened a PowerShell terminal windov
GIVEN: The WebLogReporter program is listed withi
THEN: I can run the WebLogReporter command, pass
GIVEN: The program runs without errors.
THEN: I am able to view the output HTML file in r
Acceptance Criteria:
* It's done when I can run the WebLogReporter pro
* It's done when I can run the WebLogReporter pro
* It's done when all the log data are stored in a
```

Your team decides to use C# and SQL Server to read, parse, and store the data for analysis. They decide that, while there are several good templating systems out there, nobody on the team has ever used any of them. Time is short and HTML is simple, so we'll just write our own code to convert our results represented by the results of SQL statements. Let's dive in! The requirements stipulate a console application, so that's the project type I used when creating it in my IDE. I won't be walking you through creating the project. I'm assuming you know how to create a console application using the new project options in Visual Studio.

The input data from an NGINX log looks as follows:

```
127.0.0.1 - - [16/Jan/2022:04:09:51 +0000] "GET /
127.0.0.1 - - [16/Jan/2022:04:09:52 +0000] "GET /
127.0.0.1 - - [16/Jan/2022:04:09:52 +0000] "GET /
```

When you create a console app project in Visual Studio, it creates a file called **`Program.cs`**. We're not going to do anything with **`Program.cs`** yet. I'm going to start by creating a new class file to represent a log entry. I'll call it **`NginxLogEntry`**. I can see in my sample data that we have a date field, so I know that I'm going to need internationalization, owing to the cultural info needed to render a date. So, let's get the basics in place with a **`using`** statement for the globalization package, a namespace, and the class. Visual Studio likes to mark the classes with an internal access modifier. Call me old-fashioned. I always change them to **`public`**, assuming that's appropriate, and in this case, it is:

```
using System.Globalization;
namespace WebLogReporter
{
  public class NginxLogEntry
  {
     //TODO:  the rest of the code will go here
  }
```

```
}
```

With the basics out of the way, let's set up our member variables. Aside from a couple of constructors, that's really all we'll need since this class is designed to represent the line entries in the log.

The fields we're interested in are visually identifiable in the preceding data sample:

- **`ServerIPAddress`** represents the IP address of the web server from which the log was taken.
- **`RequestDateTime`** represents the date and time of each request in the log.
- **`Verb`** represents the HTTP **`verb`** or **`request`** method. We'll be supporting four, though there are many more available.
- **`Route`** represents the route of the request. Our sample is from a RESTful API.
- **`ResponseCode`** represents the HTTP response code for the request. Successful codes are in the 200 and 300 range. Unsuccessful codes are in the 400 and 500 range.
- **`SizeInBytes`** represents the size of the data returned by the request.
- **`RequestingAgent`** represents the HTTP agent used to make the request. This is usually a reference to the web browser used, but in

all the cases in our sample, it is a client written in Python 3 using the popular **`requests`** library.

In addition to our fields, I'll start with an **`enum`** to store the four acceptable values for the HTTP methods, which I've called **`HTTPVerbs`**. The rest are represented with straightforward auto-properties:

```
    public enum HTTPVerbs { GET, POST, PUT, DELET
    public string ServerIPAddress { get; set; }
    public DateTime RequestDateTime { get; set; }
    public HTTPVerbs Verb { get; set; }
    public string Route { get; set; }
    public int ResponseCode { get; set; }
    public int SizeInBytes { get; set; }
    public string RequestingAgent { get; set; }
```

Now that I've got my enumeration and properties in place, I'm going to make a couple of constructors. I want one constructor that allows me to pass in a line from the log. The constructor will parse the line and return a fully populated class with the log line as an input. Here's the top of the first constructor:

```
    public NginxLogEntry(String LogLine)
    {
```

First, I'll take the log line being passed in and split it into a string array, using the **`.Split()`** method, which is part of the **`string`** class:

```
var parts = LogLine.Split(' ');
```

While developing, I run into some corner cases. Sometimes, the log lines don't have 12 fields, as I expect. To account for this, I add a conditional that detects log lines that come in with fewer than 12 parts. This rarely happens but when it does, I want to send them to the console so that I can see what is going on. This is the kind of thing you'd probably take out. I'm embracing my inner stovepipe developer here, so I'm leaving it in:

```
if(parts.Length < 12)
{
  Console.WriteLine(LogLine);
}
```

Now, let's set to work taking apart the line based on the split. It's pretty easy to pick out the server IP address as the first element of the split array:

```
ServerIPAddress = parts[0];
```

We don't care about those two dashes in positions 1 and 2. We can see the date in the third position. Dealing with dates has always been slightly more fun than your average root canal. Think about all the formatting and the parsing needed just to get it into something we know will work with the database code that we'll eventually write. Thankfully, C# handles this with aplomb. We pull out the date parts and we use a custom date format parser. I don't really care about expressing the date in terms of locale, so I'll use **`InvariantCulture`** as the second argument in the date parse:

```
        var rawDateTime = parts[3].Split(' ')[0].Su
        RequestDateTime = DateTime.ParseExact(rawDa
```

Next, we get to work parsing the HTTP verb. It needs to conform to the **`enum`** we defined at the top of the class. I start by pulling the relevant word and making sure it's clean by giving it a quick trim. Then, I cast it to the enumeration type. I probably should have used **`TryParse()`**, but I didn't. It still works with the input sample if I don't, and that's the kind of thinking that lands us in a stovepipe prison later:

```
        var rawVerb = parts[5].Trim().Substring(1);
        Verb = (HTTPVerbs)Enum.Parse(typeof(HTTPVer
```

The **`Route`** value, the **`ResponseCode`** value, and the **`SizeInBytes`** value are just grabbed based on their position. In the latter two cases, I just used **`int.parse()`** to turn them into integers:

```
        Route = parts[6].Trim();
        ResponseCode = int.Parse(parts[8].Trim());
        SizeInBytes = int.Parse(parts[9].Trim());
```

Lastly, I need the **`RequestingAgent`**. The sample data has some pesky double quotes that I don't want to capture, so I'll just use the **`string.replace()`** method to replace them with **`null`**, effectively getting rid of them:

```
        RequestingAgent = parts[11].Replace("\"", r
    }
```

I now have a very useful constructor that does my line parsing for me automatically. Nice!

My second constructor is more standard fare. I'd like to create **`NginxLogEntry`** by simply passing in all the relevant data elements:

```
    public NginxLogEntry(string serverIPAddress,
```

```
      requestDateTime, string verb, string route, i
      responseCode, int sizeInBytes, string request
      {
        RequestDateTime = requestDateTime;
        Verb = (HTTPVerbs)Enum.Parse(typeof(HTTPVer
                verb);
        Route = route;
        ResponseCode = responseCode;
        SizeInBytes = sizeInBytes;
        RequestingAgent = requestingAgent;
      }
    }
  }
```

This class begins as all do – with property definitions. We have a requirement to store the log data in SQL Server. For this, I created a database on my laptop running SQL Server 2019. If you don't have any experience with SQL Server, don't worry. This is the only place it's mentioned. You don't need SQL knowledge to work with patterns in this book. I created a new database called **`WebLogEntries`**, then created a table that matches my object structure. The **Data Definition Language** (**DDL**) to create the table looks as follows:

```
CREATE TABLE [dbo].[WebLogEntries](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [ServerIPAddress] [varchar](15) NULL,
    [RequestDateTime] [datetime] NULL
```

```
    [RequestDateTime] [datetime] NULL,
    [Verb] [varchar](10) NULL,
    [Route] [varchar](255) NULL,
    [ResponseCode] [int] NULL,
    [SizeInBytes] [int] NULL,
    [RequestingAgent] [varchar](255) NULL,
    [DateEntered] [datetime] NOT NULL,
    CONSTRAINT [PK_WebLogEntries] PRIMARY KEY CLU
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE =
      IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
      ALLOW_PAGE_LOCKS = ON, OPTIMIZE_FOR_SEQUENT
      OFF) ON [PRIMARY]
) ON [PRIMARY]
```

As you can see, I have added the ubiquitous auto-incrementing primary key field, simply called **`id`**. I also added a field to track when the record was entered and set its default value to SQL Server's **`GETDATE()`** function, which yields the current date on the server.

Let's move on to the code that reads and writes data with SQL Server. I think most people would use an **Object-Relational Mapper** (**ORM**) such as .NET's EF for this. I prefer to leverage the control and performance I get from working directly with the database. In my IDE, I'll create a second class called **`SQLServerStorage`**. If you're following along, don't forget to add the **`Systems.Data`** package via **NuGet**.

As before, I'll start with the dependencies:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data.SqlClient;
```

Next, I'll set up the class:

```
namespace WebLogReporter
{
  public class SQLServerStorage
  {
      //TODO:  the rest of the code goes here
  }
}
```

Unlike the data class we created earlier, this one is all about the methods. The first method I'll make stores the data in the database using a direct connection. If you've only ever used EF, and you understand SQL (which you should), I highly recommend you try this style and test it for speed against your usual EF-driven code. You'll see a huge difference, especially at scale. I'll get off my proverbial soapbox now and get back to creating the **`StoreLogLine`** method. It takes in the **`NginxLogEntry`** class that we just wrote as its sole input:

```
public void StoreLogLine(NginxLogEntry entry)
{
```

Next, let's connect to the database. I use the **`using`** syntax for this. If you've not used this before (see what I did there?), it's very convenient since it handles the timely closure and destruction of whatever you create. In this case, I'm creating a database connection. Even in throwaway code, there are things you just don't do, such as open a connection to a resource and fail to close it. It's just so rude! This line sets up my connection. I also recommend a strong database password. As usual, I can hide behind the excuse that it's throwaway code. At this point, I've likely repeated this more times than your local government has told you to wear a mask. And as with your local government, it probably won't be the last time you hear it:

```
    using (SqlConnection con = new
           SqlConnection("Server=Localhost;Data
                         WebLogReporter;User
                         Id=SA;Password=P@ssw0
    {
```

Next, I'll build my **Data Manipulation Language** (**DML**) statement for inserting data into the database using the connection we just forged.

I'll use the **`StringBuilder`** class, which is part of **`System.Text`**:

```
        var sql = new StringBuilder("INSERT INTO
                [dbo].[WebLogEntries] (ServerIPAd
                RequestDateTime, Verb, Route, Res
                SizeInBytes, RequestingAgent) VAI
         sql.Append("'" + entry.ServerIPAddress +
         sql.Append("'" + entry.RequestDateTime +
         sql.Append("'" + entry.Verb + "', ");
         sql.Append("'" + entry.Route + "', ");
         sql.Append(entry.ResponseCode.ToString()
         sql.Append(entry.SizeInBytes.ToString() +
         sql.Append("'" + entry.RequestingAgent +
```

Next, let's open the connection, then execute our SQL statement:

```
         con.Open();

         using(SqlCommand cmd = con.CreateCommand(
         {
           cmd.CommandText = sql.ToString();
           cmd.CommandType = System.Data.CommandTy
           cmd.ExecuteNonQuery();
         }

       }
```

```
    }
```

Fabulous! Now that we're writing data, it stands to reason that we should also read it. Otherwise, our class would be really cruddy. Or maybe it wouldn't be? I'll let you mull that over while I type out the next method signature:

```
    public List<NginxLogEntry> RetrieveLogLines()
    {
```

The **`read`** method is going to return a list of **`NginxLogEntry`** instances. This is why we made that second constructor in the **`NginxLogEntry`** class earlier. I'll start by instantiating an empty list to use as the return value. After that I'll make a really simple SQL statement to read all the records from the database:

```
        var logLines = new List<NginxLogEntry>();
        var sql = "SELECT * FROM WebLogEntries";
```

Using the same **`using`** syntax as before, I'll open a connection and read the records:

```
        using (SqlConnection con = new
```

```
            SqlConnection("Server=Localhost;Datab
            WebLogReporter;User Id=SA;Password=P(

        {
          SqlCommand cmd = new SqlCommand(sql, con)
          con.Open();
          SqlDataReader reader = cmd.ExecuteReader(
```

With the **`select`** statement executed, I'll use a reader to get the data out line by line, and for each record, I'll instantiate a **`NginxLogEntry`** class. Since it's supposed to be prototype code, I'm relying on the positions in the dataset for data retrieval. This is not uncommon, but it is fairly brittle. A restructuring of the table will break this code later. But it's throwaway code! See? I told you that you'd hear it again:

```
          while (reader.Read())
          {
            var serverIPAddress = reader.GetString(
            var requestDateTime = reader.GetDateTim
            var verb = reader.GetString(3);
            var route = reader.GetString(4);
            var responseCode = reader.GetInt32(5);
            var sizeInBytes = reader.GetInt32(6);
            var requestingAgent = reader.GetString(
            var line = new NginxLogEntry(serverIPAd
                       requestDateTime, verb, route
                       responseCode, sizeInBytes,
```

```
                        requestingAgent);
```

Now that I've constructed the object using the data from the table, I'll add it to my **`logLines`** list and return the list. The **`using`** statement handles the closure of all the database resources that I created along the way:

```
            logLines.Add(line);
          }
        }
        return logLines;
      }
    }
  }
```

To sum it up, the class has two methods. The first, **`StoreLogLine`**, takes an instance of the **`NginxLogEntry`** class and converts the data into a SQL statement compatible with our table structure. We then perform the **`insert`** operation. Since I used the **`using`** syntax to open the connection to the database, that connection is automatically closed when we leave the scope of the method.

The second operation works in reverse. **`RetrieveLogLines`** executes a select statement that retrieves all our data from the table and con-

verts it into a list of **`NginxLogEntry`** objects. The list is returned at the close of the method.

The last component is the output component. The class is called **`Report`**. Its job is to convert the records requested from the database into an HTML table, which is then written to a file.

I'll set up the class with the dependencies and begin the class with the usual setup:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace WebLogReporter
{
  public class Report
  {
      //TODO: the rest of your code goes here
  }
```

Next, I'll add the method to generate the report:

```
    public void GenerateReport(string OutputPath)
    {
```

I'll now use the **`SQLServerStorage`** class that we made earlier:

```
var database = new SQLServerStorage();
var logLines = database.RetrieveLogLines();
```

I have the data. Now, I'll use another **`StringBuilder`** to generate the HTML. It's table code because this is absolutely not a book on front-end design:

```
var output = new
             StringBuilder("<html><head><t
             Log Report</title></head><body
output.Append("<table><tr><th>Request
               Date</th><th>Verb</th><th>Ro
               <th>Code</th><th>Size</th><t
               </th></tr>");
foreach (var logLine in logLines)
{
  output.Append("<tr>");
  output.Append("<td>" +
      logLine.RequestDateTime.ToString() +
  output.Append("<td>" + logLine.Verb + "</
  output.Append("<td>" + logLine.Route + "<
  output.Append("<td>" +
      logLine.ResponseCode.ToString() + "<
```

```
                logLine.ResponseCode.ToString() + "</
            output.Append("<td>" +
                logLine.SizeInBytes.ToString() + "</t
            output.Append("<td>" +
                logLine.RequestingAgent.ToString() +
            output.Append("</tr>");
        }
        output.Append("</table></body></html>");
```

Finally, we have a wonderful C# one-liner to output the file so that it's ready for viewing in your favorite browser:

```
        File.WriteAllText(OutputPath, output.ToStrin
      }
    }
  }
```

It might be ugly, but it works. I'll say it once again just because I can. It's throwaway code! One trick I advocate when writing throwaway code is to make it so incredibly ugly that nobody in their right mind would put their name to it. I think I've accomplished that here. I just used a string builder to create my HTML. No spaces or formatting. It's basically a minified HTML file, which is, of course, an intended feature and not at all inspired by laziness.

There's one last thing to do before we put this baby to bed. We need to edit the **`Program.cs`** file Visual Studio created as the project's entry point. This file glues all the other pieces together. The most recent editions of most C# IDEs generate the entry point for console apps within the **`Program.cs`** file. This isn't new. What is new is the format this file takes. The new format lacks the usual constructor and class setup we've seen so far in the classes we created from scratch. Behind the scenes, the compiler is generating those definitions for us, but it makes **`Program.cs`** look different from everything else. Rather than present all the usual boilerplate, it's straight to business.

We'll start by using the **`WebLogReporter`** class that we just created:

```
using WebLogReporter;
```

We'll do a perfunctory and minimal test to make sure the right number of arguments were passed in from the command line. We need a path to the log file and an output path. If you don't pass in the right number of arguments, we'll give you a little command-line hint, then exit with a non-zero code, in case this is part of some sequence of automation. I know, it's throwaway code, but I'm not a barbarian:

```
if (args.Length < 2)
{
```

```
    Console.WriteLine("You must supply a path to th
      you want to parse as well as a path for the c
    Console.WriteLine(@"For example: WebLogReporter

      c:\temp\nginx-sample.log c:\temp\report.html"
    Environment.Exit(1);
  }
```

Now, we check whether the log input file exists. If it doesn't, we alert the user to our disappointment and again exit with non-zero code:

```
  if (!File.Exists(args[0]))
  {
    Console.WriteLine("The path " + args[0] + " is
      valid log file.");


    Environment.Exit(1);
  }
```

If they make it this far, we're assuming everything is good and we get to work:

```
  var logFileName = args[0];
  var outputFile = args[1];
  Console.WriteLine("Processing log: " + logFileNam
```

```
int lineCounter = 0;
```

We instantiate **`SQLServerStorageClass`** so we can store our records as we read them in:

```
var database = new SQLServerStorage();
```

Now, we open the input log file, and with a **`foreach`** loop, we take each line and use our parsing constructor in **`NginxLogEntry`** to create an **`NginxLogEntry`** object. We then feed that to our database class. If we encounter a line that's a problem, we write out a message that states where the problem occurred so that we can review it later:

```
foreach(string line in
        System.IO.File.ReadLines(logFileName))
{
  lineCounter++;
  try
  {
    var logLine = new NginxLogEntry(line);
    database.StoreLogLine(logLine);
  }
  catch
  {
    Console.WriteLine("Problem on line " + lineC
  }
```

```
    }
}
```

We've parsed the log data and written it to the database. All that's left is to use the **`Report`** class to write out our HTML:

```
var report = new Report();
report.GenerateReport(outputFile);
Console.WriteLine("Processed " + lineCounter + "
                  lines.");
```

To sum up, the **`Program.cs`** file contains the main program. The current version of C# allows us to dispense with the usual class definition in the main file for the project.

First, we check to make sure the user entered two arguments. It's hardly a bulletproof check, but it's good enough to demo.

Next, after making sure the input log file is a legitimate path, we open the file, read it line by line, and save each line to the database.

Once we've read all our lines, we read back the data from the database and convert it to HTML using the report object.

Your program is complete; you demonstrate it to the customer, and they are delighted! A week later, their boss has lunch with your boss,

and a new requirement comes in stating the client would now like to support logs from two other web server formats: Apache and IIS. They'd also like to select the output from several different formats, including the following:

- HTML (which we have already delivered)
- Adobe **PDFs**
- Markdown
- **JavaScript Object Notation** (**JSON**)

The purpose of the last format, JSON, is that it allows outside clients to ingest that data into other systems for further analysis, such as capturing trend data over time.

While these concise descriptions of the requirements are hardly what we'd want when building a real extension to our program, they are enough to get you thinking.

What would you do?

Have we built a stovepipe system? If not, is there a chance it might evolve into one? Pause for a moment and think about this before reading further.

I believe we have built a stovepipe system. Here's why:

- Our classes are all single-purpose and coupled directly to the web server log formats
- Our software is directly coupled with SQL Server
- Our output to HTML is the only possible output, given that we didn't create an interface or structure to abstract the output operation and format

You might be thinking that the second set of requirements was unknown at the time we created our first release. That's accurate. You might further defend your idea by stating you are not psychic, and there's no way you could have known you would need to extend the program per the second set of requirements. You're right there too. Nobody is prescient. But despite that, you know, if for no other reason than you've read this chapter so far, that any successful program must support extension. This is true because you know your first iteration has now begotten a request for a second and that always entails changes and additions to the requirements. We can never know how the requirements will change, but we do know that they will change.

How can patterns help?

These factors are undoubtedly part of your professional life right now. Perhaps only a few of them are at play at a time. If you stay in soft-

ware development for any length of time, you'll undoubtedly encounter them all at some point. Remember, everything we've talked about so far is an antipattern. All this negative energy needs a counterbalance. You might even be tempted to say that you need to bring balance to the force. Instead of becoming a dark Jedi, maybe something less radical will do. You can learn to use patterns to balance and ultimately defeat the antipatterns and the forces that create and enable them.

I think it's time we formally introduced the concept that I have left to your imagination thus far. I could offer my own definition of patterns but I'd rather stand on the shoulders of giants. From the days since Grace Hopper logged the first bug on the Mark II at Harvard in 1947 (see *Figure 1.4*), programmers and computer scientists have been facing the same problems over and over again. We get a little bit smarter every time and we write down what we did. If you take the distilled experience and knowledge gleaned through hard-won trial and error over the course of seven decades, from the early wartime pioneers to the most recent graduates, you wind up with a set of patterns, which are descriptions of solutions to consistently recurring problems.

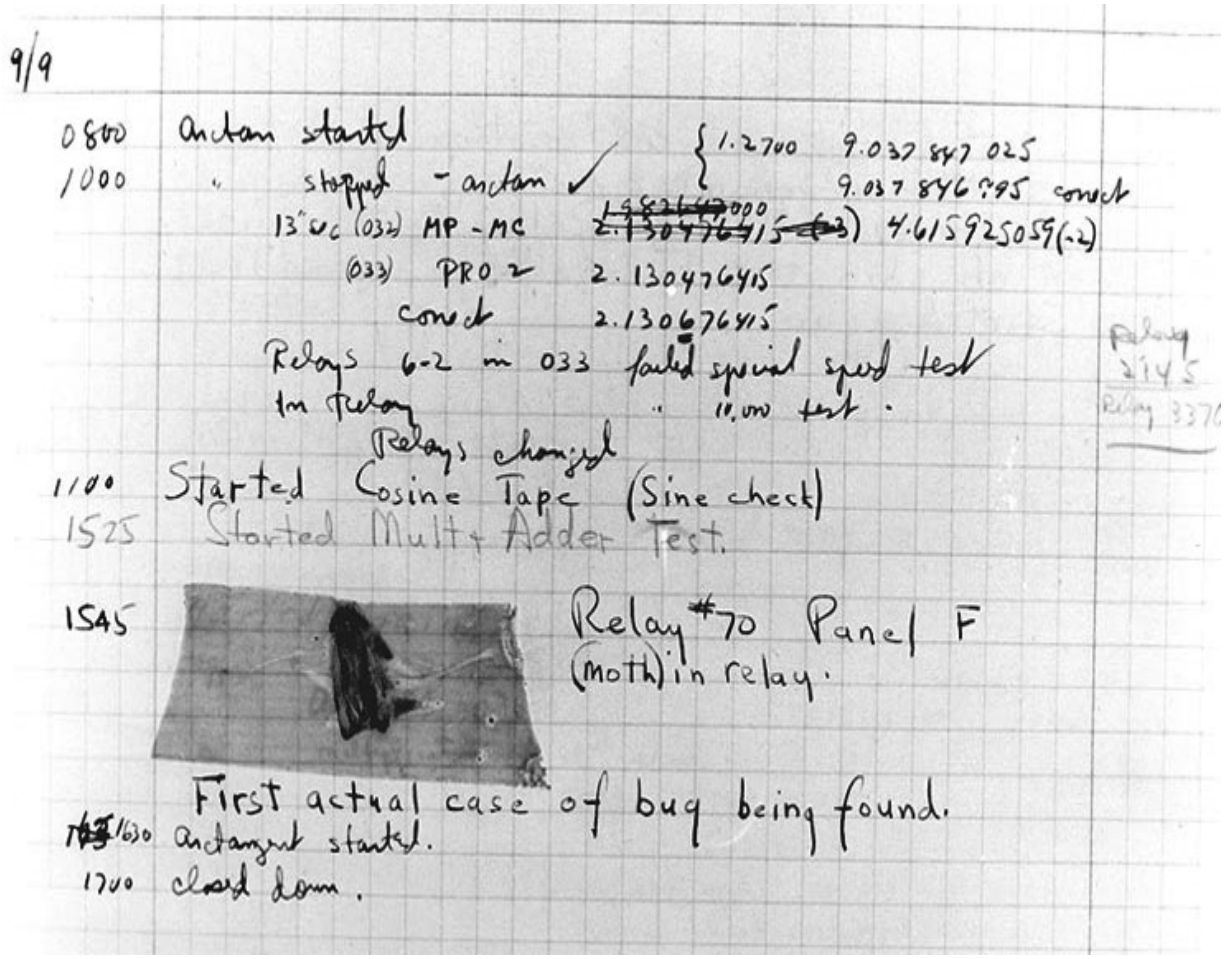


Figure 1.4 – The very first computer bug was literally a bug (moth) that had crawled into the relays of the Mark II computer at Harvard University

The idea of patterns originated in the field of architecture, that is, traditional architecture, with respect to the design and creation of buildings. In 1977, Christopher Alexander documented a pattern language designed to form the basis for the best practices for building towns. The book describes two hundred and fifty-three patterns presented as the paragon of architectural design. The book breaks everything down into objects. I find it fascinating that the language doesn't even

change when you adapt the seminal book on architectural patterns to our explanation of software patterns. I'd characterize the book as being the synthesis of a language used to describe objects in the real world and how to mold spaces and objects together to achieve harmony. As with the motto of the Vulcans in the TV and film franchise *Star Trek*, the goal of the language of patterns is *infinite diversity expressed in infinite combinations*. Alexander himself describes a pattern as a solution to problems that occur frequently enough that every practitioner will likely recognize them. He then describes a solution in a way that isn't directly tied to any implementation. Keeping it flexible in this way, the same solution may be used on a million projects and in a million slightly different ways.

Let's shift focus from the world of building architecture to the realm of software architecture. We revisit the famous GoF. They define a design pattern as an abstraction of a recurring problem that pinpoints the chief elements of design structure, focusing on the idea of creating reusable object-oriented software.

Patterns are going to be the weapon that we can use to overcome the antipatterns and dark forces that prevail in both the loftiest corporate institutions and most hallowed halls in small business.

Are you ready to fight the darkness? Roll up your sleeves and let's get to work!

Summary

This chapter initiated our discussion of patterns by defining antipatterns.

We learned that if we design software only with a mind toward meeting the requirements, we will build a system that can't be extended easily. These systems are called *Stovepipe systems* because over time, they degenerate structurally as an exhaust vent on a coal-burning stove does. You will invariably reach a point where maintaining and extending such a system is untenable. Nobody wants to be the one to tell the bosses that you need six months with no new releases so that you can rebuild the company's cash-cow product. Designing with patterns will help you avoid these kinds of pitfalls.

We also learned about a similar antipattern called *the Big Ball of Mud*. Foote and Yoder described the prevailing forces that we all recognize in our daily work lives: time, cost, experience, skill, visibility, complexity, change, and scale. These forces confound our ability to write good code in the first place. Even if we can write good code for our first release, these forces erode systems over time, such as a tiny stream forming the Grand Canyon.

We saw an example of throwaway code, which is how most projects are born. We now know what bad practice it is to just clean up the

throwaway code and then ship it. It pays to spend the time to architect projects properly and assume the worst-case scenario: that your software will be wildly successful. If it is, you can absolutely count on feature requests and new requirements that you never even imagined when you wrote the throwaway prototype.

Patterns can be thought of as a language that defines common software design elements, coupled with an abstract solution that can be implemented in many different ways. They aren't tied to a particular language or technology stack. As you learn to use patterns, your software will become more robust owing to a stronger foundation. Your projects will be able to support your inevitable success by providing avenues for future features and expansion that you couldn't have conceived of when you started your project.

In the next chapter, I'll prepare you for your patterns journey with the C# programming language. I will cover some popular idioms and practices in relation to object-oriented programming. In particular, pay close attention to the presentation of SOLID methodology, because it is the foundation upon which successful patterns are implemented.

If you are new, or perhaps returning to C# having worked with other object-oriented languages, I want to direct you to *Appendix 1* at the end of the book. I wrote this for my students and colleagues who have an interest in learning about patterns but have focused on other

languages, such as JavaScript or Python. Patterns are language-agnostic. Learning these patterns exclusive of a language is possible. However, I believe that we all learn best by doing, and that means you need an implementation language. The neat thing about programming languages is that they are pretty much all the same. All of them use variables, objects, methods, collections, and loops.

In *Appendix 1*, I'll cover the object-oriented features of the C# language. I've alluded to my idea that newer developers within the field are the most at risk in terms of knowing the language reasonably well but simultaneously clinging to Golden Hammers and using C# to produce stovepipe software. I have a great many students who learn JavaScript from me and who, at my strong encouragement, want to take the next steps in their journey by learning C#. Given how differently inheritance and common structures such as objects and classes work between the two languages, you'll undoubtedly detect my desire to be inclusive. I had originally written this appendix to be *Chapter 2*, but I didn't want to bear the expense of having the C# crowd cry out, "I'm BORED." Even if you're a C# veteran, I encourage you to peruse the chapter. You might find I've explained things a little differently from many other authors, and certainly differently from an academic textbook.

Questions

1. What, in your own words, is a pattern? What is an antipattern?
2. What antipatterns have you seen in your own work before?
3. What is a stovepipe system? Can you point to an example from your own body of work? Don't worry, I won't tell.
4. Can you think of a time when you wielded a Golden Hammer? What was the hammer and what did you perceive as the nails?

Further reading

- Alexander, C. (1977). *A pattern language: towns, buildings, construction*. Oxford University Press
- Brown, W. H., Malveau, R. C., McCormick, H.W. S., & Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.
- Foote, B., & Yoder, J. (1997). *Big Ball of Mud*. Pattern languages of program design, 4, 654–692. Retrieved from http://www.laputan.org/pub/foote/mud.pdf
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Patterns, D. (1995). *Elements of reusable object-oriented software* (Vol. 99.) Reading, Massachusetts: Addison-Wesley
- Johnson, J. (1995). *Creating Chaos*. American Programmer, July 1995
- Nazeer, H. (2020). *A pattern language: towns, buildings, construction (review)*. Journal of Research in Architecture and Planning,

Vol. 29, Second Issue

2

Prepping for Practical Real-World Applications of Patterns in C#

Becoming a prepper

You may think that because I was born in Oklahoma and live in Texas, I could be one of those crazy people who runs around in the woods preparing for the imminent end of the world. Hey, let's try to keep my personal life out of this! However, when I tell you that I want you to become a prepper, it has nothing to do with a conspiracy theory or end-of-times rhetoric. Instead, it has to do with a fundamental transformation that is likely to be happening in your professional life. Unlike the radical political transformations that concern most real-world preppers, ours are hopefully fully beneficial, with the aim to make us better at our craft. This chapter is geared toward preparing you to use patterns. Relying on patterns to improve your code doesn't make sense if the code you're working with doesn't conform to the following conditions:

- Sports a high level of organization
- Structured for modification with minimal risk
- Testable
- Measurable

So, you need to prepare. Think of it as though you were sanding down rough spots on a piece of wood before painting it. The patterns are the paint; if you apply the paint before sanding, all the rough spots are going to stand out.

Consider this chapter an explanation of why the code at your company is the way it is. If you work at Microsoft, Google, Apple, Meta, Twitter, or Amazon, your code is likely in good shape. You need to understand that a high-quality code base doesn't happen by accident, nor is it immune to the same chaotic forces that plague the physical universe. Entropy isn't limited to the concerns of thermodynamics in some distant stellar nebulae. It's happening all the time in your code and you must learn to recognize it and work diligently to fix it or fend it off.

If you work at one of the other 99.99% of the companies in the world that creates software as a normal part of doing business, there might be some unrecognized opportunities for improvement. Let's look at what we're covering in this chapter:

- Spaghetti < lasagna < ravioli – software evolution explained with pasta
- The foundational principle – writing clean code
- Creating maintainable systems using SOLID principles
- Measuring quality

When you reach the end of the chapter, you'll be able to do the following:

- Identify some common metaphors for poorly constructed code, as well as exemplary code.
- Articulate the importance of good basic coding habits as the key to the success of your project and applying code. Patterns make your code better, but only if it's clean code.
- Apply SOLID principles to your code and your design process.
- Identify the most common software metrics so we can understand when our code is more likely good than bad.

Technical requirements

This chapter presents some code samples. They are geared toward demonstrating some of the concepts we're going to be covering. I'm not sure they are exciting enough to make you want to recreate the projects yourself, but you are welcome to do so. Should you decide to try any of this out, you'll need the following:

- A computer running the Windows operating system. I'm using Windows 10.
- A supported IDE such as Visual Studio 2022, JetBrains Rider, or Visual Studio Code with C# extensions.
- .NET Core 6 SDK.

Spaghetti < lasagna < ravioli – software evolution explained with pasta

The first part of any prepping journey involves securing your food supply. Real-world preppers love pasta because it's portable and keeps without refrigeration. So, let's start with pasta.

The title of *Chapter 1* was *Why Is There a Big Ball of Mud on Your Plate of Spaghetti?* I never mentioned spaghetti. I didn't need to. It's such an obvious descriptive metaphor for a chaotic mess that it likely needs little further discussion. Spaghetti code is actually mentioned in the original publication of the *Big Ball of Mud* antipattern. What a lot of developers don't see is that spaghetti code is a symptom. It isn't the disease itself. The real diseases are the antipatterns. It is possible to have multiple infections ransacking your body at the same time. It is equally as possible for your code to be similarly and simultaneously

inflicted with multiple antipatterns. You can gauge your code's health by deciding where your code lies on the pasta spectrum..

Spaghetti code

Spaghetti code is characterized as chaotic, poorly organized, hard to follow, and impossible to maintain. The more spaghetti you have, the worse it is. As soon as spaghetti appears, it tends to spread rapidly. We've also compared code architecture to a physical building's architecture in the previous chapter. There is a well-known principle from the field of criminology called *the broken windows theory*. The theory postulates that outward signs of crime, anti-social behavior, civil disobedience, or visible urban decay encourage further crime, particularly vandalism. A building with one broken window will soon have all its windows broken because it's clear that you can get away with throwing rocks with no repercussions. Who doesn't enjoy breaking Windows? Microsoft does it all the time.

Your code is the same way. If you allow just one broken window, your code is no longer clean. You need to correct the situation immediately or it won't be long before all you have in your code repository is a Big Ball of Mud on your plate of spaghetti.

Maybe running fast and loose has paid off so far. Maybe it allowed you to get your software to market ahead of any competitors. Maybe

your bosses and stakeholders are impressed with the velocity and small turnaround time on your initial releases. All that feels pretty good. What's really happening is that you're being rewarded for throwing rocks at your own windows. You're sacrificing long-term profit for short-term gain.

You need to get out of the spaghetti production business entirely.

Lasagna code

The logical step up from spaghetti code is lasagna. Most characterizations of lasagna code say that it is the object-oriented version of spaghetti. I don't necessarily agree. Lasagna itself consists of many of the same ingredients found in spaghetti. It's still noodles, tomato sauce, spices, and often meat or vegetables. The difference is in how it's organized. Instead of hundreds of intertwined chaotic structures of noodles that can only be tamed by twirling them on a spoon, lasagna uses big noodles that serve as boundaries between discrete layers. Many of the layers are composed of the same stuff as the other layers.

In software, layered code fixes the spaghetti code into many small classes. Using lasagna as a metaphor for software cropped up around 1982, as the industry was beginning to move to a new hardware architecture. We were moving away from big-iron monolithic

mainframes to something called **client-server**. The mainframes of the day required a financial commitment similar to that of the GDP of several small European countries combined. Client-server systems, on the other hand, were affordable for a reasonably well-funded small business. The systems consisted of a powerful server, servicing multiple clients, taking the form of less powerful computers. Does this sound familiar? It's the same model used by the internet. The difference is that networking technology is now universal, exponentially better, faster, and more reliable. Spoiler alert: **cloud computing** is not a new concept. It's just a marketing term for work done on other people's computers.

It's worth noting that the 1980s also brought us the first forays into object-oriented programming. I doubt Java was even a glimmer in James Gosling's eye then, and the inventor of C#, Anders Hejlsberg, was still in college. However, Bjarne Stroustrup began working on **C++** in 1979 while working at AT&T Bell Labs. He based C++ on the C language. C is a strictly procedural language designed to control telecommunications switching networks. Stroustrup reimagined this language by bolting on object orientation by way of classes.

C++ was released to the world in 1985 when Stroustrup published a textbook on the language. Those of you who were not yet born now understand that the 80s brought us more than just hairbands, Madonna wannabes, *The Karate Kid*, and Valley girls. It brought a whole

new way of thinking about and designing software. It was a mere co-incidence that we looked fabulous while doing it.

The hardware shift toward client-server necessitated a change in software design practices, as we considered what pieces should run on servers and which on clients. We started seeing something that we call **separation of concerns** today. When done well, separation of concerns is a really good idea. When done poorly, it creates lasagna.

Lasagna doesn't usually begin as lasagna. A software developer thoughtfully lays out a layered architecture that separates components of the overall system into layers called *tiers*. Here's an example of a common four-tier architecture design:

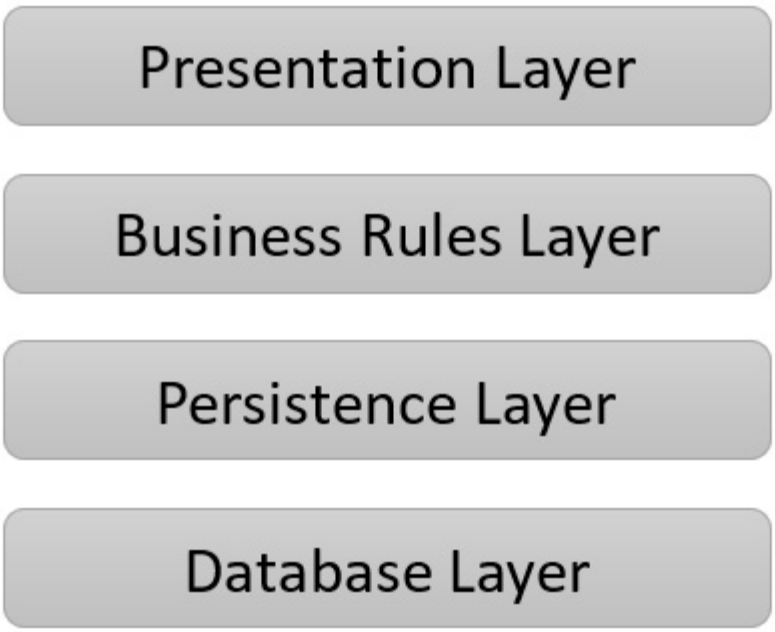


Figure 2.1 – A common four-tiered architecture

This is exactly the kind of architecture we see in modern mobile applications. The presentation layer is the app you downloaded from the app store, which runs on your phone, while the remaining tiers run invisibly in the cloud.

Since the separation exists at the hardware level, it makes sense to think about making software in a set of layers that mirrors the systems running the program. The presentation layer is the **view**, or the part of the program interacting with the users. The business rules layer implements the logic that drives what the presentation layer displays. It draws information from the persistence layer, which is responsible for storing and retrieving data. Some of this might be stored as files, or in memory, but some data will undoubtedly be at the lowest level in a database. All communication between the layers should move through each layer above it or below it, depending on the direction of flow for the data. As these layers become more and more complicated, it is easy for functionality to start bleeding out of the intended layer into other layers.

Imagine you need a quick fix for the way something is displayed in the UI. Maybe you want to show something in one way to an administrative user, and another way to a regular user. The right way to effect this change might be to alter a rule in the business rule layer, but it's easier and faster to slap something directly into the UI. You've just broken a window. It's no big deal; you tell yourself you'll fix it later.

In other scenarios, lasagna can be the result of an overcorrection. Take a programmer who writes spaghetti code in a procedural language such as C, then give them an object-oriented language such as C++ or C#. Now, add a book on patterns. The natural inclination of the programmer will likely be the same as yours the first time you realized that you could use different fonts in *PowerPoint*. One font is good; therefore,, all 25 on your computer must be better! Naturally, you start creating presentations that try to use as many fonts as possible. Patterns are an amazing tool. They are meant to guide you toward clean architecture the way fonts, used sparingly, should add emphasis, clarity, and aesthetics to a presentation. Patterns aren't meant to be a prison.

The lasagna problem isn't limited to large, multi-service programs. You'll see it in smaller ones too, in the form of layers of inheritance. Inheritance is a foundational feature of object-oriented programming. If a language doesn't support inheritance, it's not in the club, and it can't call itself object-oriented. Consider what the **`NginxLogLine`** class from the project in *Chapter 1*, *There's a Big Ball of Mud on Your Plate of Spaghetti*, might look like after a few iterations of requirements gathering, coding, and releases. The use case I'm presenting here is pure fiction and doesn't actually relate to the release history of NGINX's log structure. I need a believable series of complications to help make my example work.

Let's say the code in *Chapter 1* represents the first release. The client was thrilled and started coming up with new requirements, as the software was exposed to different business users in the real world. A shortlist of new requirements might be the following:

- It was noted by some of the IT professionals using the program that while it works perfectly with the latest version of the web server software, it breaks when working with older versions because the log structure has changed several times over the last few years. We need to support the last major revision's log format, as well as the most current.
- Users based in Asia noted that the text formatting in their log files is not supported because their text files are encoded using double-byte formats. We need to support international text formats.
- Another important IT group can't even use your software because they don't use NGINX exclusively. They also use Apache and they noted they have the same constraints. The log formats have changed over time, so they have several formats to consider, and they too need support for single- and double-byte text implementations.

Let me reiterate my earlier disclaimer. While I do know my way around a web server log, I am by no means a web server log aficionado. As a reminder, this is a fictitious cautionary tale. Any resemblance of our requirements to actual software requirements is purely coinci-

dental. I further promise under the penalty of perjury that no cute and fluffy animals will be harmed in the equally fictitious production of said software. My lawyer wants me to add more, but my editor wants me to get on with it already and I learned early that my editor is always right.

Next, let's suppose since there was zero inheritance used in the original code that maybe it was because we were new to C# and didn't fully understand how to use inheritance. Now, suppose we bought a book from a publisher other than Packt. It was written by a university professor who has never published software in a commercial setting. This is common. They've never heard a pointy-haired boss who wielded godlike power over their livelihood tell them to clean up the prototype and ship it next week. At no point have they faced a deadline unrealistically dictated by a marketing department. In short, this book would be by an author who doesn't work in the real world of software development.

This kind of author may have presented a muddy, overly academic picture of C#'s inheritance model, along with the academically perfect way to structure your code. They are basing their estimation on the available academic literature reviews, also written by university professors who have never actually worked in the field. Your young, sharp, supple, but impressionable mind might be persuaded by all the

author’s academic credentials to make something similar to what follows:

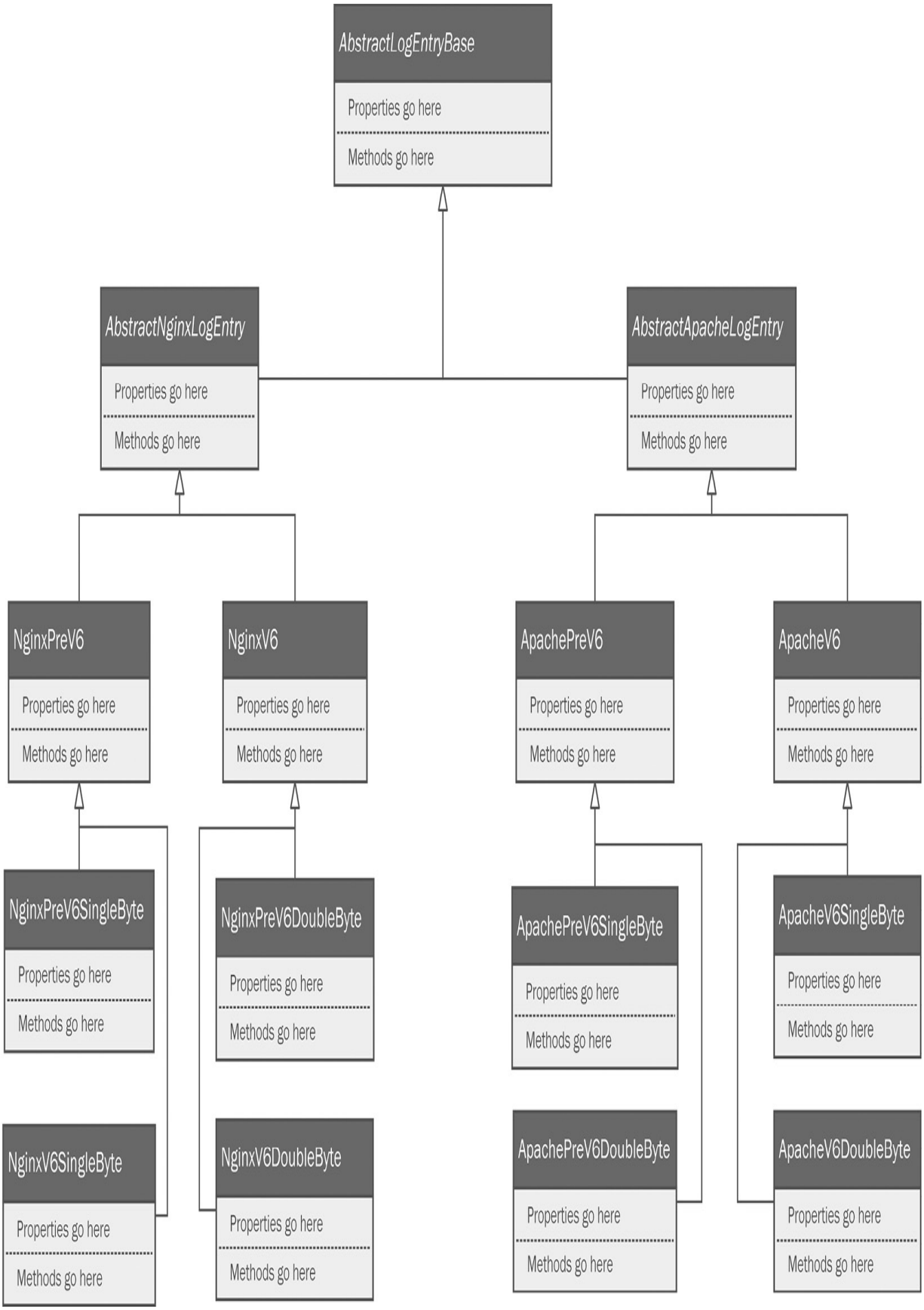

AbstractLogEntryBase
Properties go here
Methods go here
AbstractNginxLogEntry
Properties go here
Methods go here
AbstractApacheLogEntry
Properties go here
Methods go here
NginxPreV6
Properties go here
Methods go here
NginxV6
Properties go here
Methods go here
ApachePreV6
Properties go here
Methods go here
ApacheV6
Properties go here
Methods go here
NginxPreV6SingleByte
Properties go here
Methods go here
NginxPreV6DoubleByte
Properties go here
Methods go here
ApachePreV6SingleByte
Properties go here
Methods go here
ApacheV6SingleByte
Properties go here
Methods go here
NginxV6SingleByte
Properties go here
Methods go here
NginxV6DoubleByte
Properties go here
Methods go here
ApachePreV6DoubleByte
Properties go here
Methods go here
ApacheV6DoubleByte
Properties go here
Methods go here

Figure 2.2 – The UML equivalent of a dumpster fire next to a train wreck (this is an example of too much inheritance)

This is a poorly constructed inheritance chain represented using the **Unified Modeling Language** (**UML**). If you've never seen a UML class diagram before, you should jump to *Appendix 2* at the end of the book. It has an overview of how to draw and interpret these diagrams.

The inheritance chain represented in the diagram goes deeper than is necessary, The design goal can be accomplished more cleanly by using tools such as interfaces and techniques such as composition. You're also staring at the beginning of class proliferation.

Class proliferation happens when you find yourself needing to add more and more classes to support new requirements. This sets us up for a classic stovepipe design.

Too many layers of inheritance, especially when functionality that belongs in one class slowly bleeds into other classes over time, will yield lasagna.

If only we had yet another pasta metaphor that is delicious and satisfying, but less deleterious to the work-life balance of whoever has to maintain what we've created thus far…

Ravioli – the ultimate in pasta code

While everybody loves spaghetti and lasagna, you don't want them showing up in your code. Ravioli, on the other hand, is something we aspire toward.

When we make ravioli, it is again comprised of the same set of ingredients used to make spaghetti and lasagna. Again, the difference is in the configuration of the materials. The noodles now form a full boundary around the deliciousness inside. The meat or cheese within the noodle is fully encapsulated, with the content only ever being exposed when it is consumed. There's probably also a delicious sauce on the outside that brings the whole dish together into something that's not chaotic and not unnecessarily dense.

It is also worth noting that the level of effort needed to shape and stuff the noodles is a lot more work than simply boiling a handful of straight noodles. Making ravioli, as with good object-oriented code, requires patience and work. The idea of ravioli is the epitome of pasta object-oriented design if you remember the foundational pillars used to previously describe object-oriented design:

- Encapsulation is used to limit access to an object's state.
- Abstraction refers to a class modeling a real-world object with only the level of detail needed by the software. If we were to write soft-

ware that needed a `Person` class to model a human being, the `Person` class would be modeled differently in a medical records application than it would in a phone book application.
- Inheritance allows common structures between classes to be shared in a parent class, negating the need to copy the same code repeatedly between classes.
- Polymorphism allows a class designer to defer implementing abstracted details to concrete classes. For example, an abstract `Vehicle` class might have a `Go()` method. The `Go()` method would work very differently for concrete classes such as a car versus a boat, a skateboard, a submarine, or an airplane. Polymorphism allows us to create the appropriate implementation at the concrete level but define it at the abstract level.

We need to talk a little more about encapsulation with specific regard to C#. In well-designed object-oriented code, objects are encapsulated. This means the object's state is closely guarded. Only the instance should be allowed to alter its own internal state.

By contrast, poorly constructed code allows for objects to directly change the state of other objects. Auto-implemented properties in C# make this very easy by making properties no better than fields. If every property is public, and there are no rules, as is the case with auto-implemented properties, any object can change the state of any other object. We can compound the problem with a long inheritance

chain, where every property at every level is public, internal, or protected. When this is the case, any object at any level can impose state changes up or down the inheritance chain. This makes it difficult to track what is changing each object's state and under what circumstances. State changes bleeding between layers is the very definition of lasagna.

If your code is similar to ravioli, each instance of each object stands alone as a fully encapsulated class with limited inheritance. It is the master of its own state, and it meticulously and entirely controls that state. Since these objects bear all these hallmarks, you can easily leverage composition to build complex objects instead of relying solely on inheritance to define the behavior of an object.

Composition is a technique focusing on building objects out of other objects. Composition works best when we use interfaces to define how these objects fit together. An interface defines the structure of an object. Think of it as a mechanical socket such as a light socket. You can plug any bulb into that socket, provided it fits. What makes it fit is the definition presented by the dimensions of the socket. Interfaces in C# can define a set of properties and methods that an object is expected to have. Any object that fits the specification can be used just as a bulb fitting into a socket would.

Imagine a set of classes designed to model an automobile. There are lots of different kinds of cars. There are sports cars, family sedans, minivans, and my personal favorite, the Jeep. All of these cars are very different. We could try to come up with an inheritance chain to tie them all together in a familial hierarchy. I suspect though, that if you limited your design to strictly inheritance, you'd wind up with hundreds of classes. The resulting code would be a messy, muddy plate of pasta.

A better idea is to use composition. We can keep our abstract car model, but instead of making subclasses, we can add interfaces to define the characteristics of the car. For example, we can make an interface for each component that defines a car. For example, we can make an interface that defines an engine.

We can also create an interface that defines the transmission. There is a huge difference between the transmission of a family sedan, a high-end sports car, and a 4x4 off-road vehicle. However, it is possible to define a common set of properties and methods. Any transmission object I can model will slot into my car object, provided it follows the required interface.

If I were to sum up the job of an object-oriented developer, and I were only allowed one sentence, I would say that our job is to always en-

sure that no object should ever be allowed to fall into an invalid state. I'll expand on this idea in the next section.

The foundational principle – writing clean code

The main point of presenting the topics in this chapter is to set a boundary. You can master all the patterns in this book and more, but if your software is poorly written, overly clever, haphazardly structured, or hard to maintain, then all the patterns in the world can't help you. Let's set some boundaries. I'm going to suggest some guidelines for creating "clean code." You are welcome to argue the minutiae. It doesn't bother me one bit if we differ in opinion over tabs versus spaces as long as you have a method to your particular flavor of madness. Let's paint some broad strokes that hopefully everyone can agree with.

Clean code has the following characteristics:

- Easily readable by human beings with limited cognitive load
- Consistent in style
- Documented with an appropriate level of commentary

You should write code that is readable by humans

I feel as though this isn't as obvious as it should be. We write code to be compiled and executed by machines who don't care for the expressiveness we use when we write our code. Writing code is writing language. Human languages are expressive and so are humans. We can take our written languages and create a business email indicating stock shortages, a medical report outlining potential patient outcomes following a diagnosis of dermatomyositis, a master's thesis on molecular biology, a long-form romance novel, or even a haiku. Your computer languages are also expressive with their limited vocabulary. C# has 79 keywords. With those 79 keywords, you can create everything from a short program to download your email to an entire operating system. It's a powerful and expressive language. As long as you use valid syntax, the compiler will be happy. You need to focus your efforts on making your code readable to other people. I've always believed the golden rule applies here: *do unto others as you would have others do unto you.* You should treat everyone with the same level of respect you'd like to be treated to yourself. You should write code that requires minimal effort by others to scan and understand.

This starts with the way you name classes, variables, methods, and the other elements used to create your software. If the names repre-

sent the intent for how those elements are used, you've taken a step toward creating code that is more readable and more maintainable. Some staples of best practice include the following:

- Name your elements so that their intended use is intuitive. **`this.pn`** is not intuitive. You can likely think of a dozen things it might be. If we'd used **`this.phoneNumber`**, you wouldn't have to guess.
- Create names that are searchable. **`MAX_FILES_PER_UPLOAD`** defined as a constant makes it easy to find in your code, especially if you're using an IDE that indexes your code.
- Leave outdated encodings to the old-timers. You likely don't do things like this unless you are very (how can we say it nicely?) seasoned. A long time ago, in this very galaxy, we learned to create variable names using codes. I don't mean programming code; I mean codes as in Hungarian notation. This was before we had type-checking IDEs. It reminds me of Garrison Keeler's *Lake Wobegon* podcasts where he spun tales of a simpler time. We wrote in simple console editors such as *vi* and *emacs* and we liked it that way. We might as well have been using *Notepad* with our fingers tied together. Back then, we didn't have *Roslyn* looking over our shoulders and pointing out our missteps. We needed a way to tell which data type was used in our variables, so we'd call a variable **`intAge`** or maybe **`iAge`**. As I said, you probably don't do this anymore unless you were taught this way. If you were taught this way, kindly knock it off. Thank you.

- Don’t bother with member prefixes or suffixes. It used to be common to prefix member variables with `m_`, for example. As with encodings, this kind of thing isn’t needed because we have good IDEs, and your classes should be short and discrete in function so that prefixes are not needed. Besides, after a while, people tend to ignore them as noise while scanning your code and they cease to be significant. I’ve seen university professors teach it this way for a long time. I might have been one of them. I’m not proud of it, but that was a long time ago and I’ve moved on since then. The exception that I still observe is the common custom of naming private fields with an initial underscore. I think most people do that so they can name the field something obvious that matches the property name where they intend to expose the field.
- Use parts of speech in your code. Your objects are nouns. They represent people, places, and things. Name them as such. **`Person`** makes for a good abstract class name, which might be extended in **`Student`** and **`Professor`**. Class names that contain verbs as part of the name, such as a class named **`ParseLogLines`**, become confusing as class names. Naming the class **`LogLineParser`** is clearer because it sounds like a thing rather than an action. Within classes, the methods are your verbs, so name them that way. **`ParseLogLines`** totally works as a method name. If you pay attention to these details, your code will wind up reading as a normal sentence would, albeit with odd but understandable punctuation.

- **Don’t Repeat Yourself** (**DRY**). By this, I mean don’t write the same code twice. Also, don’t duplicate anything, and never repeat something you’ve already written. Dang it, sorry about that. I see this most often when people are in a hurry. They’ve got some code in another project or another part of their current project but it’s not written in a way that is conducive to reuse, so they copy and paste it into different programs or different parts of the same program. This is a broken window. Soon, your code will not be DRY; it’ll be wetter than an octopus’ belly button, and that kind of thing will be everywhere.
- Get rid of dead code. This is a huge pet peeve for me. Once I was working as a Java developer. I know, I was young and I needed the money. I was pretty new to the job and we had a database function that wasn’t writing to the database as expected. I’d change the method around and run the tests. Same result. Eventually, I decided to make a silly, ridiculous big change. Nothing. I was working on a method called `WriteInventoryPartToDatabase`. OK, that seems well named. It seemed to be an obvious place to look for the problem. After an hour or so, I realized the method I was working on was *old code* and that the actual method I needed had been moved to another class. The developer who moved the method kept the old method in place *just in case* and never cleaned it up later. OK, people. This is why revision control systems exist. You can always go back. Don’t be lazy. If you delete or

remove something, don't just change it in the calling method. Get rid of the dead parts or someone else will potentially spend a lot of time on a red herring.

- Format your code for human consumption. Something such as this should be an anathema to your very being:

```
var l = 1;
var O = 0;
if (O == l) {
  O++;
} else {
  l = O * (l + 1);
}
```

Our IDEs have gotten pretty good at defaulting to fonts that help us detect the differences between zeros and capital Os, and lowercase Ls and 1s. Options such as turning on font ligatures help even more. Ligatures are improved fonts that show you a more expressive set of characters. For example, look at the way `!=` is rendered in *CaskaydiaCove Nerd* Font from nerfonts.com. It appears as ≠, which is the way your high school math teacher presented it:

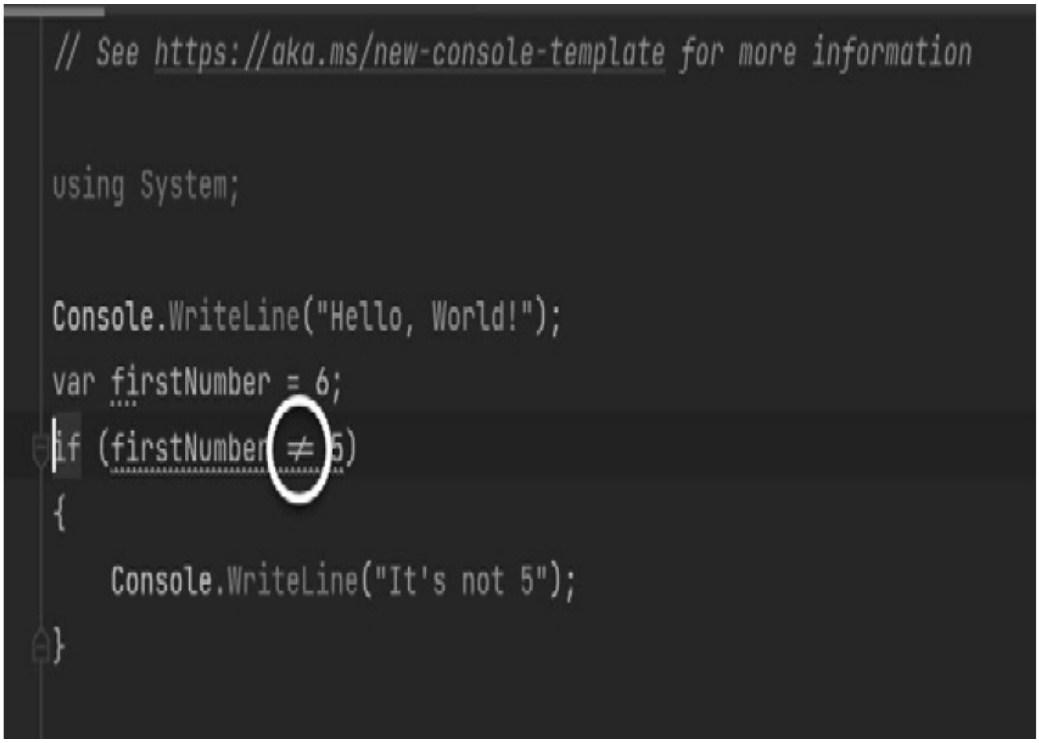


Figure 2.3 – An IDE with font ligatures turned on displaying !=

You can turn these on in your IDE. In Visual Studio, you just set the font to one that supports ligatures:

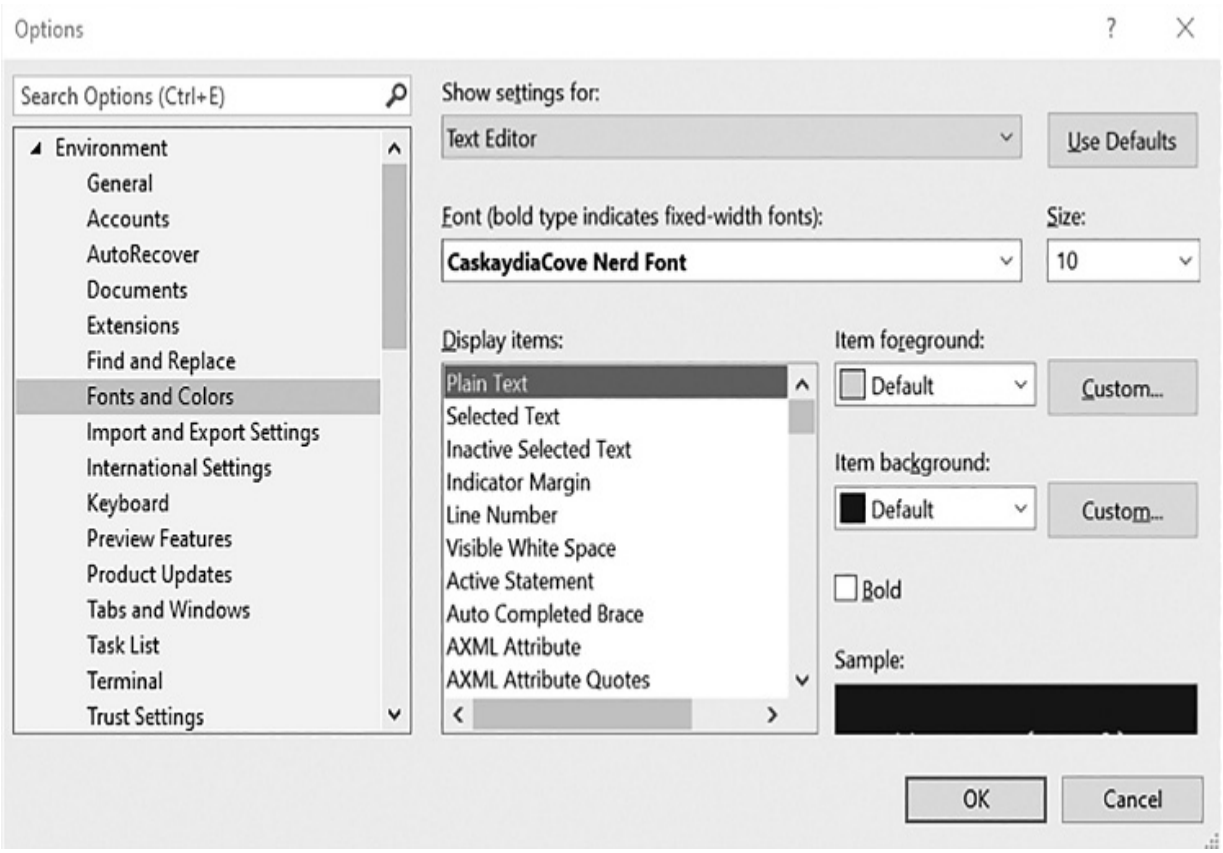


Figure 2.4 – In Visual Studio, simply set the font in the Options window to one that supports ligatures, such as Caskaydia-Cove Nerd Font Mono, which you can find free at nerdfonts.com

In *JetBrains Rider*, you have an actual setting:

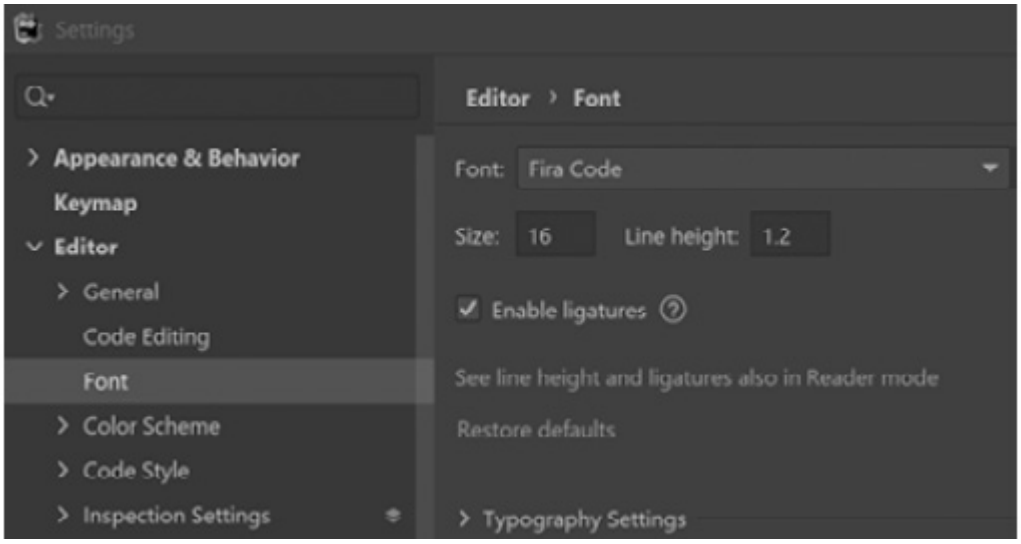


Figure 2.5 – JetBrains Rider IDE settings for fonts allow you to specifically turn ligatures on and off

If you prefer *Visual Studio Code*, you'll need to edit your **`settings.json`** file:

```
"editor.fontFamily": "CaskaydiaCove Nerd Font, mo
"editor.fontLigatures": true,
```

The IDE's normal coloring helps too, except when you're reading a book printed in grayscale 😊.

The previous bullet points are guidelines for writing code for human consumption. These tips will help you as we move into the next section.

Establishing and enforcing style and consistency

You should be using a set of conventions for consistently naming elements of your code, as well as applying a consistent coding style. If you do this well, you won't be able to tell where your code ends and someone else's starts. I'm not going to dedicate time to recommending coding conventions, as these are well established in the industry. Some such conventions are baked into the IDEs, such as curly braces always dropping to a new line, For the rest, you can use an automation tool such as JetBrains' *ReSharper*, *Prettier* for *Visual Studio Code*, or the open source project called *StyleCop*. All of these have tools that allow you to add style enforcement to the IDE, as well as the ability to run a check when you commit your code to a **Continuous Integration** (**CI**) server. Mavericks who aren't using the proper styling will fail their builds, giving everyone else on the team the opportunity to help them see the error of their ways.

Limiting cognitive load

The problem isn't that the poorly written code is unintelligible. The problem is that it's unscannable. One thing I've noticed over the years is that, regrettably, I no longer read entire paragraphs in prose anymore. Granted, it's rare that I read books with zero code in them. Over the years, my brain has adapted to reading code, so I scan. If

you've also been reading code for a while, you probably do it too even if you don't realize it. Remember this and work toward making your code scannable. If you can just look at it and think *OK, got it. Next?*, then it's easy to scan and understand. The opposite is having to stare at some code for seconds or minutes in order to decode it in your brain. It takes a noticeable and uncomfortable amount of time to figure out what it means and what it does. Psychologists refer to this as **cognitive load**.

Try this instead:

```
var lastIndexedValue = 1;
var oldValue = 0;
if(oldValue == lastIndexedValue)
{
  oldValue++;
}
else
{
 lastIndexedValue = oldValue * (lastIndexedValue-
}
```

That's better. No more thinking. Save that for when you really need it.

Terse is worse

Speaking of cognitive load, I know a lot of developers who love terse syntax. Let me show you what I mean. I found a perfect example on Stack Overflow at https://stackoverflow.com/questions/7103979/nested-ternary-operators/7104091#7104091:

```
_viewModel.PhoneDefault = user == null ? "" :
  (string.IsNullOrEmpty(user.PhoneDay) ?
    (string.IsNullOrEmpty(user.PhoneEvening) ?
      (string.IsNullOrEmpty(user.Mobile) ? "" :
        user.Mobile) :
      user.PhoneEvening) :
     user.PhoneDay);
```

It's so terse that it's impossible to scan. You're going to have to hunker down for a good few minutes to even figure out what it does. Some developers think writing code in this way makes them appear smarter than everyone else. It doesn't. This would be similar to an English writer who is overtly sycophantic and obsequious toward their quasi-internalized sesquipedalian. Did you get all that, or did your eyes scan back across the sentence a few times? I could have simply said, "*This would be the same as an English writer who goes out of their way to give in to the obsessive use of long words.*" Good writers can write to be understood by a university-educated audience, which is to say, their peers. Great writers can write the same content to be

understood by a group of talented sixth graders. This takes as much concentrated effort as writing tersely, but you can do the same thing with your code and everybody charged with maintenance will thank you for it.

Let's refactor it:

```
if(user == null)
{
  _viewModel.PhoneDefault = string.Empty;
}
if(!string.IsNullOrEmpty(user.PhoneDay))
{
  _viewModel.PhoneDefault = user.PhoneDay;
}
if(!string.IsNullOrEmpty(user.PhoneEvening))
{
  _viewModel.PhoneDefault = user.PhoneEvening;
}

if(!string.IsNullOrEmpty(user.Mobile))
{
  _viewModel.PhoneDefault = user.Mobile;
}
```

Bam! Welcome to the least sexy code ever written. If you post it to Reddit, the best you can hope for is the occasional *meh* amidst a

litany of trollish comments, not all of which are limited to your code. That said, now you can scan it, and the rules governing setting the default phone number are fairly obvious. This is true because while the terse code was bad, we did at least use obvious object and property names to indicate proper intent.

Comment but don't go overboard

Commenting on your code is good. I've read academic articles that present solid arguments for as much as one-third of your lines of code being devoted to comments. The problem with this is the practice is dangerously close to writing documentation, and the only thing programmers hate more than writing documentation is directly interacting with users. Good news – if you write clean code using the ideas I've stressed so far, you can get away with fewer comments because your code will already be easy to read.

I think a happy medium is represented by a comment containing a brief restatement of the requirements. If your requirements are documented in an online system as they should be, you could even link to the requirement.

I also comment on anything that isn't obvious, such as my motivations for writing something a certain way. This is a useful comment: *I'm doing it this way because our suppliers require data to be in this*

format. This way, my team doesn't come in behind me, see something they think needs refactoring, but wind up breaking code that conforms to customer requirements.

Comments go bad when there are too many of them. Commenting every line is silly unless you're writing a book, or teaching a class for beginners who are learning to read code for the first time.

Creating maintainable systems using SOLID principles

SOLID is a reference to the top five principles of **Object-Oriented Design** (**OOD**):

- The Single Responsibility principle
- The Open-Closed principle
- The Liskov Substitution principle
- The Interface Segregation principle
- The Dependency Inversion principle

Following these principles will allow you to create systems that are robust, extensible, and maintainable. Honoring these principles prepares you well for working with patterns because many patterns are built on or reference these principles.

The Single Responsibility principle

Every method should do one thing. Every class should represent one thing. We call this idea the **Single Responsibility Principle** (**SRP**). If you have a method inside an object that does many things without invoking outside methods, your method is doing too much and runs the risk of becoming an example of the antipattern known as *the god function*. These are big, messy piles of inedible pasta. Once, I got a desperate text from a colleague. Her program was crashing. She couldn't figure out why. I looked. The entire program was in one file that when printed out was over 20 pages long. There were nine functions in the whole program. I closed the project, deferring the review for a time when I could sit down and go through it in earnest. I highly suspected she was ignoring the SRP.

Let's look at an example of a method that does too much:

```
public void doesTooMuch()
    {
      StreamReader sr = new StreamReader("C:\\Sar
      var line = sr.ReadLine();
```

We've started a function that's clearly going to do too much. We begin by opening a text file on our computer's hard drive and reading in

the first line. Next, let's read through each line in the file, and send each line of text to a fictitious online service that translates the text into another language.

To be valid, the sentence needs the text to be trimmed, all uppercase, and there can be no semicolons in there because the service's author got hacked by SQL injection once, and he vowed never again:

```
        while (line != null)
        {
          Console.WriteLine(line);
          var processedLine = line.Trim();
          processedLine = line.ToUpper();
          processedLine.Replace(";", ""); // no sql
```

OK, the text is ready. Let's transmit it:

```
          var url = "https://fake-translation-
              service.com/translate";
          var httpRequest =
              (HttpWebRequest)WebRequest.Create(url
          httpRequest.Method = "POST";
          var data = "{\"input\":\"" + processedLin
          using (var streamWriter = new
          StreamWriter(httpRequest.GetRequestStrea
```

```
            {
              streamWriter.Write(data);
            }
```

We've transmitted the data; let's parse out the response and print it to the console:

```
            var httpResponse =
                (HttpWebResponse)httpRequest.GetRespo
            using (var streamReader = new
            StreamReader(httpResponse.GetResponseStre
            {
              var result = streamReader.ReadToEnd();
              Console.WriteLine("Translates to " + re
            }
```

Read the next line, rinse, and repeat until we reach the end of the file:

```
              line = sr.ReadLine();
          }
```

If you open it, you should close it, so let's clean up after ourselves:

```
        sr.Close();
      }
```

Can you see the problem here? We have one method performing many operations:

1. We open a file.
2. We process each line.
3. We transmit to the service.
4. We process the results.

Each of these should be separated into its own method. When you do this, you can reuse the methods in other contexts to solve other problems. Reading a file is generic. It's the kind of thing you'd do often. So is sanitizing your input string. So is posting to a **`RESTful`** endpoint. You can't reuse the **`doesTooMuch()`** method. It's too specific to a single point of implementation. In *Chapter 3, Getting Creative with Creational Patterns*, we're going to take off the training wheels and start learning patterns. Patterns are utterly incompatible with god methods of this kind.

The Open-Closed Principle

Classes should be open for extension but closed for modification. This is called the **Open-Closed Principle** (**OCP**). This is especially

true of software that's already in production. You've got a set of well-written, fully tested production classes. Messing around with the wiring introduces the risk of breaking something. When new code can be written as an extension to what you already have, the risk is limited to the extension.

Let's look at an example by looking at code that violates the OCP. We're going to make a simple utility designed to add up the areas of a set of geometric shapes. For our initial release, we'll support circles and squares. I'll represent the **`Circle`** library this way:

```
public class Circle
{
  public double Area { get; }
  public Circle(double radius)
  {
    Area = Math.PI * (radius * radius);
  }
}
```

I've set this up with an **`Area`** property that is read-only. There is no reason to allow someone to set the property directly. Instead, I use the constructor to force you to define the radius of the circle on instantiation. This prevents you from setting **`Area`** to whatever the heck you feel like, which might result in the object entering an invalid state.

When you instantiate, you pass in the radius, which is all we need to find the area of the circle. Good old $\pi \bullet r^2$, or the constant pi times the radius squared. The area is set automatically.

We can do something similar with a square. We only need to know the length of one side:

```
public class Square
{
  public double Area { get; }
  public Square(double lengthOfOneSide)
  {
    Area = lengthOfOneSide * lengthOfOneSide;
  }
}
```

As before, we present an **`area`** property as read-only and use the constructor to set the area automatically upon instantiation by multiplying the **`lengthOfOneSide`** argument by itself.

Now, I have two classes to represent my shapes and each has an **`area`** property set automatically on instantiation. All I need is a class to glue them together:

```
public class AreaCalculator
```

```
  {
    private double _area { get; set; }
    public double Area { get { return _area; } }
    public void AddShape(Square square)
    {
      _area += square.Area;
    }
    public void AddShape(Circle circle)
    {
      _area += circle.Area;
    }
}
```

Here, we have a class called **`AreaCalcuator`** with an **`area`** property. On this one, I chose to create a backing variable to make it easy to keep a running total. Each time you add either a square or a circle using the twice-overloaded **`AddShape`** method, it takes the area computed on instantiation and adds it to the total.

I can test its function with code as follows:

```
var areaCalculator1 = new AreaCalculator();
areaCalculator1.AddShape(new Square(5d));
areaCalculator1.AddShape(new Square(25.3452d));
areaCalculator1.AddShape(new Circle(2342.093d));
Console.WriteLine("The total area of the shapes
```

This looks pretty good! Ship it already! All is well until our delighted customer returns with a request to support more shapes. It's not hard, is it? Maybe all they want is a rectangle. All I need to do is make a **`Rectangle`** class, then modify the **`AreaCalculator`** class with another constructor.

If I do this, I am violating the OCP because I have to change the **`AreaCalculator`** class directly every time we get a new shape requirement. I should design it so that you can pass in anything with an **`area`** property on it, then I would never have to alter **`AreaCalculator`** ever again. Let's fix it.

I'll make an interface to define my requirement for an **`Area`** property:

```
public interface IShapeWithArea
{
  public double Area { get; }
}
```

Now, let's modify the shape classes to implement the interface. This is easy since both already have the **`area`** property exposed in a way that satisfies the requirements of the interface.

First, let's do the circle:

```
public class OCPCircle : IShapeWithArea
  {
    public double Area { get; }
    public OCPCircle(double radius)
    {
      Area = Math.PI * (radius * radius);
    }
  }
```

I renamed the class **`OCPCircle`** so that I wouldn't get them mixed up. Now, let's do the square:

```
public class OCPSquare : IShapeWithArea
  {
    public double Area { get; }
    public OCPSquare(double lengthOfSide) { Area
        lengthOfSide * lengthOfSide; }
  }
```

Next, I'll modify the **`AreaCalculator`** class to use the interface as its type on the **`AddShape`** method:

```
public class OCPAreaCalculator
  {
    private double _area { get; set; }
```

```
    private double _area { get; set; }
    public double Area { get { return _area; } }
    public void AddShape(IShapeWithArea shape)
    {
      _area += shape.Area;
    }
  }
```

Awesome! Now, we never need to alter this method in this class again. It is open for extension and closed for modification. To honor any new requirement for any shape, I need only add a new class that implements the **`IShapeWithArea`** interface.

I'll go ahead and add a rectangle class that implements the same **`IShapeWithArea`** interface:

```
public class OCPRectangle : IShapeWithArea
  {
    public double Area { get; }
    public OCPRectangle(double width, double hei
    {
      Area = width * height;
    }
  }
```

It doesn't matter what the clients ask for next. Adding support for a hexagon, rhombus, or dodecagon is a simple matter of Googling the area formula for that shape and making a new class that implements our interface.

The Liskov Substitution principle

In 1988, Barbara Liskov delivered a keynote speech at a conference titled *Data Abstraction and Hierarchy*, wherein she introduced what came to be called the **Liskov Substitution Principle** (**LSP**).

The principle states that any object of a subtype should be a suitable and working substitute for its superclass. This really relies on inheritance more than anything else. Using the strategy I just showed you, which is to rely more on interfaces than inheritance, you'll not likely run afoul of the LSP. By now, you've no doubt surmised that I enjoy breaking rules (ask anyone), so let's break this one.

The requirements I'm working toward are similar but not identical to the last example. Here, I have a requirement to compute the area of rectangles – just rectangles. To make this work, pretend that I didn't show you the best solution already in the last example:

```
public class Rectangle
{
```

```
    public double Width { get; set; }
    public double Height { get; set; }


    public double Area { get { return Width * Heigh
  }
```

I was a little less cautious this time. I used normal auto-implemented properties for the width and height rather than a constructor. If I'm breaking rules, I may as well let my hair down entirely. The **`Area`** property is suitably encapsulated, so you can't get up to anything too nefarious here.

Now, let's make a class to handle adding up areas of rectangles in a very similar fashion to our last example for the OCP:

```
public class RectangleAreaCalculator
{
  public double Area { get; set; }
  public void AddShape(Rectangle rectangle)
  {
    Area += rectangle.Area;
  }
}
```

That's pretty straightforward. Not even an hour after you create this, the ubiquitous pointy-haired boss tells you about a requirements

change. You need to support a square too.

No problem, you think. A square IS A rectangle. Obviously, we can do this with inheritance. Perhaps we don't need any change at all, but the pointy-haired boss must be appeased. I highlighted the IS A term earlier because this phrase characterizes inheritance, and it happens to be true. A square IS A rectangle. We make the class. Since we're using inheritance, we need to make a quick change to the **`Rectangle`** class so that it can support inheritance. We'll make the properties virtual so that we can override them in the subclass if we need to:

```
public class Rectangle
  {
    public virtual double Width { get; set; }
    public virtual double Height { get; set; }
    public virtual double Area { get { return Wid
  }
```

The thing about the square is that it is rectangular, but we only need the length of one side to compute the area. The parent class, **`Rectangle`**, requires the length of two sides. Suddenly, things are starting to look a little ugly. Let's think while we type:

```
public class Square : Rectangle
```

```
{
  private double _lengthOfSide;
  public override double Width {
    get { return _lengthOfSide; }
    set { _lengthOfSide = value; }
  }
  public override double Height {
    get { return _lengthOfSide; }
    set { _lengthOfSide = value; }
  }
  public override double Area {
    get { return Width * Height; }
  }
}
```

I had this great idea! We could fix it so that we have a backing variable. Any time you modify the width or height, the **`accessor`** method simply changes the backing variable, which is referenced by the getter on the **`Area`** property. Since both the width and height are set to the same thing, multiplying width times height will give us the area of the square. It doesn't mess up the whole rectangle class needing both values. I would now like you to visit YouTube and search for *Guinness beer commercial brilliant*. Pick any video with the cartoonish-looking guys. I'll wait. Brilliant! This calls for celebration, perhaps even a libation. Will it work? Of course it will. I'll ship it and feel extra special today, as though I got away with something naughty because

I was clever. In fact, right now might be a good time to go do my taxes.

Meanwhile, in a totally different department, the newbie developer recently hired by the pointy-haired boss wrestles with a similar requirement. Naturally, we want to reuse as much as we can. The newbie takes your code but does something totally unexpected. The requirements make it necessary to change the numbers for the width and height. Never mind why. Do you know what happened to the last newbie that questioned the veracity of the pointy-haired boss' technical direction? They're in a basement cubicle without air conditioning somewhere hot and humid, working on ways to speed up bubble sort algorithms using Z80 assembly language on punched cards. Their breakroom is devoid of a latte machine. They don't even get Herman Miller chairs. It's downright barbaric. You don't want that to be you, right?

Here's the newbie's code:

```
public class LiskovAreaCalculator
{
  public double Area { get; set; }
  public void AddShape(Rectangle rectangle)
  {
    rectangle.Width = 10;
```

```
      rectangle.Height = 20;
      if (rectangle.Area != 200)
      {
        throw new Exception("Bad area!");
      }
      else
      {

        Area = rectangle.Width * rectangle.Height;
      }
    }
  }
```

Brilliant? Maybe, but probably not. Some requirement is forcing the newbie to change the values of width and height independently within the superclass. This is not a problem for the **`Rectangle`** superclass, which is designed for this kind of thing. However, this will absolutely break the square implementation when you try to substitute the square for the rectangle, despite the obvious IS A relationship between a square and a rectangle.

If you're familiar with the Darwin Awards, you'll know that every winner utters the same phrase before embarking upon whatever it is that won them the award. In Texas, or pretty much any southern state, it happens as follows: *Hey, y'all? Hold my beer and watch this.*

I'm going to show you an example of Liskov substitution.

```
var test1 = new LiskovAreaCalculator();
var testRectangle = new Rectangle();
testRectangle.Width = 5d;
testRectangle.Height = 6d;
test1.AddShape(testRectangle);
// Don't forget the answer won't be 30 on purpose
//It prints 200.
// That's not the problem.
Console.WriteLine("Area of test rectangle is " +
```

So far, so good. Everything works. This next part is where it breaks:

```
var testSquare = new Square();
testSquare.Width = 5d;
test1.AddShape(testSquare);
Console.WriteLine("Area of test square is " + tes
```

If we pass a square to **`AddShape`**, we get an answer that isn't 200. It comes out as 400, which causes the error to be thrown.

Because our intrepid newbie needed to change the **`base`** class to modify the values independently, we broke the LSP, which says we must be able to substitute the subclass, **`Square`**, for the superclass,

`Rectangle`. While our sneaky overrides in the `Square` class seem clever, it doesn't work for every case. You might say we're trying to fit a square peg in a round hole. Then again, you might think better of it, ask for that beer back, and stare admiringly at the rugged simplicity of the earlier OCP solution with the interface, which is likely the best solution to this problem.

You've now seen an example of broken Liskov substitution. No doubt you want to see it done well. I will be happy to oblige you in later chapters, which are chock-full of references to the LSP.

The Interface Segregation principle

No class should be forced to implement an interface that it doesn't use, nor should it depend on a method it doesn't use. Don't you hate it when this happens? You're forced to use something you don't need or want? This is undoubtedly how the newbie who blew it with the Liskov substitution problem felt. However, we're coders, not psychologists, so let's press on and expand upon the idea of railing against being forced to use things we don't need via the **Interface Segregation Principle** (**ISP**).

Let's say we get a new requirement that has us working with two- and three-dimensional shapes, where we need the area for two-dimensional shapes and the volume for three-dimensional shapes. Natural-

ly, we want to stick with something that works, and the interface idea in the section on the OCP seemed to work so well that it proved useful in the Liskov scenario too.

This time, the pointy-haired boss doesn't give us a newbie – he gives us a seasoned veteran. Unfortunately, this seasoned veteran has a case of psychogenic blindness as a result of the *not invented by me* syndrome. This is to say that when he looks at any solution he didn't invent personally, he instantly assumes said solution is flawed, and he expresses his disdain for the solution using colorful language commonly heard emanating from maritime professionals. In short, he cusses as though he were a sailor when he describes code that he didn't write, even though the person who did write that code is probably a few cubes away. Since awkwardness means nothing to this guy, given he's the smartest guy in the room, he decides he doesn't need anything that we've written so far and he strikes out on his own.

He's in a hurry and this work is beneath him anyway. He wants to get back to solving the traveling salesman problem and the n-body problem simultaneously with a runtime of O(log n). Your silly project is in the way. He half-heartedly types the following:

```
public interface IPollutedShape
{
```

```
    public double Width { get; set; }
    public double Height { get; set; }
    public double Depth { get; set; }
  }
```

OK, he wouldn't call it **`IPollutedShape`**. I did that for you. I spent 3 years working in technical pre-sales and I learned something that stuck with me. Never let the truth or any sense of realism get in the way of a good story.

You can see the problem without the implementation, right? If I made a **`cube`** or **`cuboid`** class that extends this interface, it's fine. They're both three-dimensional shapes and we need three dimensions to calculate volume.

However, if we use the same interface with a two-dimensional shape, we'd be forced to have a **`Depth`** property, which has no place in such a class:

```
public class PollutedSquare : IPollutedShape
{
  public double Width { get; set; }
  public double Height { get; set; }
  public double Depth { get; set; } //this is use
  public double getArea()
  {
    return Width * Height;
```

```
    return Width * Height;
  }
}
```

Yuck! We've got this extra property just sitting there doing nothing! It reminds me of that time my aunt Linda had spinach in her teeth. Whatever, we saved time by only making one interface. Our cocky super-coder can get back to picking up Sir Isaac Newton's slack.

We have now violated the ISP by forcing the use of a property or method that we don't need. That other guy isn't paying attention. Someone asked about how it is to be a graduate student at Carnegie Mellon, and there's nothing he loves more than talking about his time in graduate school back when he was dumbing down his best work just so that his professors could understand his assignments. This is perfect. Let's fix it while he's not looking.

Obviously, we need two interfaces. Unlike inheritance, where you can only have one superclass, you can implement multiple interfaces on one class. With C#, there's an even cleaner way. Let's make the interface for the two-dimensional shapes:

```
public interface ITwoDeeShape
{
  public double Width { get; set; }
```

```
    public double Height { get; set; }
  }
```

Then, we do this:

```
public class SquareISP : ITwoDeeShape
{
  public double Width { get; set; }
  public double Height { get; set; }
  public double getArea() { return Width * Height
}
```

That was easy. A three-dimensional shape uses all the same parts as a two-dimensional shape but adds depth. Let's do this:

```
public interface IThreeDeeShape : ITwoDeeShape
{
  public double Depth { get; set; }
}
```

C# allows inheritance in interfaces. I used the properties defined in the interface for two-dimensional shapes and extended it to require a property representing depth. Now, let's type out the **cube** class for the win:

```
public class CubeISP : IThreeDeeShape
{
  public double Width { get; set; }
  public double Height { get; set; }
  public double Depth { get; set; }
  public double getVolume()
  {
    return Width * Height * Depth;
  }
}
```

Ta-da! We honor the ISP by only enforcing exactly what is needed and no more. There's nothing wrong with stacking multiple interfaces if that's your thing. You can also use inheritance in your interfaces, as I did.

The Dependency Inversion principle

When you first learn how to design an inheritance hierarchy, you learn that high-level objects establish a base, while low-level objects form dependencies on that base. The **Dependency Inversion Principle** (**DIP**) flips this upside down and forms the basis of a very important practice called **decoupling**. We saw the power of decoupling in the OCP example earlier. We learned how to use an interface to prevent a strong coupling between two concrete classes.

I am offering a note of caution. Dependency inversion on the surface sounds similar to **Dependency Injection** (**DI**). It also sounds similar to **Inversion of Control** (**IoC**). These are not identical concepts. Given that we just talked about Liskov substitution, you should understand that they are not interchangeable. The reasons are subtle but they are there. I don't want to stray too far off-topic, so I'll leave you a link in the *Further reading* section at the end of this chapter where Martin Fowler explains the subtle differences between the three seemingly synonymous words. Let's move our focus back to the DIP. This principle states two points:

A. High-level classes or modules should not depend on low-level classes or modules and both should depend on abstractions.

B. Abstractions should not depend on implementation details but instead details should depend upon abstractions.

To me, this sounds comparable to something a Shaolin monk might say in a kung fu movie right before ripping off his shirt and serenely engaging in mortal combat. Let's see whether we can bring it to the street. Consider how you might design a lamp using OOP principles. One way might look as follows:

```
public class CoupledLamp
```

```
{
  public bool IsLit { get; set; } = false;
}
```

There's a lamp. We need a button to turn it on and off:

```
public class CoupledButton
{
  public CoupledLamp Lamp { get; set; }
  public void ToggleLamp()
  {
    if (Lamp.IsLit)
    {
      Console.WriteLine("The lamp is off");
    }
    else
    {
      Console.WriteLine("The lamp is on");
    }
    Lamp.IsLit = !Lamp.IsLit;
  }
}
```

This lamp will work. When you trigger the **`ToggleLamp`** method inside the button, the lamp will turn on or off as expected. However, it violates the DIP. The notion that a lamp is a high-level object while the button is a low-level object feels intuitive. There's just something

wrong with putting a lamp inside of a button. You'd think it should be the other way around, just as it is in real life. Your gut feeling here results in violating both tenets of the DIP. The high-level object is inside the low-level object, and everything is a concrete class with nary an abstraction in sight. How do we fix it?

We can use composition. Remember that composition entails building, or composing, an object out of component objects. This could be something similar to the following:

```
public class ComposedButton
{
  private bool _isOn = false;
  public bool Toggle()
  {
    _isOn = !_isOn;
    return _isOn;
  }
}
```

Here's our button with no lamp in it. Instead, the button should be in the lamp. To put it another way, we should compose a lamp using a button:

```
public class ComposedLamp
```

```
{
  private bool _isLit;
  public ComposedButton Button { get; }
  public ComposedLamp(ComposedButton button)
  {
    Button = button;
  }
  public void ToggleLamp()
  {
    _isLit = Button.Toggle();
    if (_isLit)
    {
      Console.WriteLine("The lamp is on");
    }
    else
    {
      Console.WriteLine("The lamp is off");
    }
  }
}
```

We inverted the dependency. The button is inside the lamp and the low-level object directly depends on the high-level object. We've satisfied the first tenet of the DIP. We still have more work to do before we reach Shaolin-level mastery. The button and the lamp are still tightly coupled via concrete object implementations. What if we want to use something other than the button to turn the lamp on? Maybe a motion detector. Maybe one of those fancy **Internet of Things** (**IoT**)

microcontrollers that everybody's talking about. Effectively, this is still a button, but it's much fancier and its object representation would be more complex than our button. We need an abstraction for the button that allows us to plug in any object that works as a button would. Let's make an interface that describes what the lamp needs:

```
public interface IToggleServer
{
  bool ToggleOnOff();
}
```

Then, we make the button extend the interface:

```
public class DIPButton : IToggleServer
{
  private bool _enabled = false;
  public bool Enabled { get { return _enabled; }
  public bool ToggleOnOff()
  {
    _enabled = !_enabled;
    return _enabled;
  }
}
```

Now, we rewire the lamp:

```
public class DIPLamp
{
  public IToggleServer DipoleSwitch { get; set; }
  public DIPLamp(IToggleServer dipoleSwitch)
    { DipoleSwitch = dipoleSwitch; }
  public void ToggleLamp()
  {
    if (DipoleSwitch.ToggleOnOff())
    {
      Console.WriteLine("The lamp is on");
    }
    else
    {
      Console.WriteLine("The lamp is off");
    }
  }
}
```

As usual, I changed the names a little bit so that we can keep our examples straight. I changed the name of the member variable that formerly held our concrete button to something more generic, namely **`DipoleSwitch`**, which I figured could describe just about anything that fulfills a binary on-and-off function.

We now have a lamp with high-level and low-level objects in the proper orientation. We could argue that they're correct now and that we

started with them inverted the wrong way. Kind of like in the southern United States when someone offers you sweet tea or unsweetened tea. There's no such thing as unsweetened tea. It isn't as if you put the sweetness in and then took it out. It was unsweetened when it was made. The wording is weird, but we say it that way anyway. That's how our lamp started when it was inside the button class. It works, and everybody knows what you meant, but it's still wrong.

We also removed the concrete button object and swapped it for an interface. We could have used an abstract class to satisfy the second tenet, but I still believe we should tend toward interfaces, as they are more flexible.

We have, in the process of learning about the DIP, learned what it means to decouple. Decoupling is a strategy rather than a pattern and it has many uses at many different levels of your software projects.

Measuring quality beyond the development organization

We've spent a lot of time in this chapter talking about quality code from the perspective of software engineering. Software is never developed in a bubble. It is always developed in concert with business

professionals working to solve business problems. That means a great many people on your project might not be developers or engineers. All of our discussions so far have revolved around specific engineering practices. However, you need to realize that these views won't be shared by your business-oriented colleagues.

For them, *quality* is often defined as *conformance to requirements*. That means quality code is code that meets its requirements. I would argue that this is a far too basic definition. The corporate quality movements of the 80s and 90s, such as the lean manufacturing and zero defects ideals espoused by Deming, have been coopted into our field by well-meaning managers. They want to focus 100 percent on the customer. They have trouble relating to what we do because they don't understand anything about software beyond the superficial layer the users see. Your non-engineer colleagues will have trouble talking to you about quality because you are speaking different languages.

To get everybody on the same page, let's consider some questions:

- What does the term *quality code* mean to you?
- What does it mean to your boss?
- What does it mean to your boss's boss?
- What does it mean to your end users?

- What does it mean to your suppliers, that is, the business professionals who supply the raw process inputs that drive your work?

After you've done this exercise, you've likely explored a wide gamut of perspectives.

Code reviews

In an age where everything is expected to be available in an instant, it becomes increasingly difficult to stress refactoring as a regular exercise. It's very easy to solve a bug report as quickly as possible, do something sloppily, then move on to the next bug. Runners have the *runner's high*, referring to a release of endorphins at some point during the run. Personally, I wouldn't know. I only run when chased. I think coders have a similar dependence on the dopamine released when we move an issue from *In Progress* to *Done*. Equally, if you have a project manager who reports up the chain of command, it's tempting to always present progress in terms of the number of issues worked through, rather than using actual quality metrics and tending to the health of your code base.

The best practice is to conduct code reviews.

Code reviews are perhaps the most important practice for maintaining a healthy code base, which leads to happy developers. You'll experience fewer shipped bugs, fewer outages, and higher retention

rates at a time when organizations are engaged in stiff competition to retain their development talent. Code reviews are an easy and inexpensive way to invest in your individual contributors. The developers under review, often by a senior teammate, will learn better techniques and more patterns. The senior developers can help their junior counterparts identify areas where they might benefit from training, at the beginning of the course, by assigning them a good book on patterns. Hopefully, you presently find yourself in a position to recommend one.

A lot of this book assumes very little about your level of experience in the field of software development. Following that principle, you may currently be wondering what a code review entails. A bad one is where someone glances at your code and after a minute or two says, "*Meh. Looks good to me*." They sign off on it and go back to what they were working on. Code reviews need to be scheduled as a meeting. If the code review is an interruption to someone else's work, it will not yield results, and it will not be effective.

When conducting a code review, there are two key areas to consider: overall design and functionality.

Overall design

This is ultimately a book designed to help you improve your understanding and practice of software design. It stands to reason that I'd

bring this up in a review. Some questions I invariably ask consist of the following:

- How does the new code fit into the overall architecture of the production code?
- Does the code follow SOLID principles, domain-driven design, behavior-driven design, or other paradigms followed by the team?
- Are formatting and style conventions being honored? Do the names of things follow the guidelines we established earlier? Do they make sense? Are they easy to read? Are they self-describing?
- Is the code readable per the first half of this chapter?
- What patterns are used and are they appropriate?
- Is the code in the right place? We don't want lasagna!
- Can the new code be reused in other places? This might necessitate moving it.
- Could the new code replace something we already have? If your addition has 100 lines of code that can replace $1{,}000$, that's a huge win!
- What can I take out of the current code now that we have this new addition? I consider it a big deal when I get to remove code. If only this could apply to the US tax code as well!
- Is the code too clever? Clever code requires effort through the cognitive load required to understand it. Look for opportunities to make your code scannable.

- Aside from being readable, is it rife with features that were not asked for or specified? Is it loaded with code for use cases that are very unlikely to ever be needed? This is called **You Ain't Gonna Need It** (**YAGNI**).

Use of the word "ain't" will likely anger my editors, my wife, my boss, my pastor, and my sixth-grade English teacher, but there it is. Deal with it, y'all. Having reviewed the overall design, the review can shift to functionality.

Functionality

The review's focus should entail making sure the code does what it is supposed to do. It should also ask questions about how we know the code is done beyond the warm feeling we got when we changed the issue status from *In Progress*:

- Can someone from another group who doesn't know the code look at this code and understand it?
- Are the automated unit and regression tests passing consistently? What? You don't have tests? You need tests!
- Is there documentation for the new features? How will the end users know how to use these new features, or even know that they're there?

- Are exceptions communicated well enough to give a full account of what went wrong to the technical team without blaming or unnecessarily confusing the user?
- Does this new code introduce holes in your security?
- Does anything violate regulatory constraints for your industry?
- Did you benchmark performance? Can we make anything faster?
- Is there any risk involved with rolling this to production right now?

What have you not thought of?

This one's tough. You can't do it by yourself. You need code reviews with another developer. While this is not an exhaustive checklist, these questions should get you moving in the right direction.

Summary

This chapter was all about preparing you to transform the way that you work and the way you that think about creating software. Learning patterns is a big deal. They usually encourage big changes to the quality and maintainability of your projects.

We started with some common metaphors that describe the typical devolution of software projects, namely spaghetti code and lasagna code. Spaghetti code is represented by a chaotic structure. Lasagna code is represented as a tiered structure with a leaky state and func-

tionality between layers. Ravioli code represents the best code, as it uses the same ingredients as spaghetti and lasagna, but the contents of ravioli are fully encapsulated.

We introduced a few topics that usually fill entire books on their own. SOLID principles are the guiding star for most serious coding organizations, but I rarely see them taught. Beginner- and intermediate-level coders are usually so focused on languages and syntax that they don't develop the muscles needed to plan and execute an architecture.

If this is you, the five principles of SOLID design are an excellent starting point and should be understood before diving into patterns.

We ended with a few pointers on performing code reviews.

You are now ready for an adventure. You've got all your prepper gear and you're ready for anything I might throw at you. In the next chapter, we'll begin learning about patterns, starting with creation patterns.

Further reading

- Martin, R. C., & Martin, M. (2006.) *Agile principles, patterns, and practices in C#* (Robert C. Martin.) Prentice Hall PTR.

- Martin, R. C. (2009.) *Clean code: a handbook of agile software craftsmanship.* Pearson Education.
- Schubert, B (2013.) *DIP in the Wild [Blog post.]* Retrieved from https://martinfowler.com/articles/dipInTheWild.html#YouMeanDependencyInversionRight.

Part 2: Patterns You Need in the Real World

These chapters explain the most useful and popular patterns created by the Gang of Four. We're strictly focused on the patterns that appear all the time in real-world projects – patterns you should definitely know to advance your craft, and your career, to the next level. In each case, we will provide a generic UML diagram, followed by a real-world application. The generic pattern diagram will be re-drawn to fit the problem and we will then write the coded solution as instructed by our diagram.

This part covers the following chapters:

- *Chapter 3, Getting Creative with Creational Patterns*
- *Chapter 4, Fortify Your Code with Structural Patterns*
- *Chapter 5, Wrangling Problem Code by Applying Behavioral Patterns*

3

Getting Creative with Creational Patterns

Creational patterns deal with creating objects, a process we call *instantiation*. Remember, an object is a class that has been instantiated. Objects only exist in running programs. They are built from blueprints called *classes*. Since C# is a static language, you can't generally change the structure of an object once it has been instantiated, which means you should use the best strategy to create your objects. That's what we'll be discussing in this chapter.

Even if you're **new** to software development with C# (that might be the first pun in the book that relies only on formatting to be funny), you already know the simplest way to instantiate an object from a class. You simply use the **new** keyword and invoke the class's constructor:

```
var myConcreteClass = new ConcreteObjectThingy();
```

That's instantiation. You're creating an instance of a class that acts as the point where the class turns into an object. In C#, like many lan-

guages, we use the `new` keyword in conjunction with a constructor. A constructor is a method, whose name. matches the class's name Its only job is to create and return an instance of the class as a new object. That is why you have a parenthesis at the end of `CreateObjectThingy()`.

Why do we need anything else? Remember the foundational principles we covered earlier in this book. We don't want a stovepipe system consisting of a bunch of object classes we can't build from because they're too specific, too concrete, or tightly coupled to one another. By definition, the `new` keyword creates a concrete object. Our classes should be open for extension but closed for modification. and However in stovepipe software they aren't stovepipe software, they aren't. Following patterns will help us avoid those situations and provide us with a clear way to create complex objects that are easy to understand and extend.

In this chapter, we will cover the following topics:

- The initial design
- No pattern implementation
- The Simple Factory pattern
- The Factory Method pattern
- The Builder pattern
- The Object Pool pattern

- The Singleton pattern

These patterns will be shown in simple C# command-line projects to keep the proverbial signal-to-noise ratio low. There are a great many books out there that teach patterns independently of any coding language. I think this is a mistake. Most people learn by doing, and to "do" patterns, you need an implementation language. Without that, you are just learning academic concepts. We are focused on the real world and we're using C#, but what you will learn in this book is 100% portable. As you learn about these patterns, remember that they are not language-specific. You can use the knowledge you gain here and apply it to Java, Python, or any other object-oriented language.

Technical requirements

Throughout this book, I assume you know how to create new C# projects in your favorite **integrated development environment** (**IDE**), so I won't spend any time on the mechanics of setting up and running projects. Should you decide to try any of this out, you'll need the following:

- A computer running the Windows operating system. I'm using Windows 10. Since the projects are simple command-line projects, I'm pretty sure everything here would also work on a Mac or Linux, but I haven't tested the projects on those operating systems.

- A supported IDE such as Visual Studio, JetBrains Rider, or Visual Studio Code with C# extensions. I'm using Rider 2021.3.3.
- Some version of the .NET SDK. Again, the projects are simple enough that our code shouldn't be reliant on any particular version. I happen to be using the .NET Core 6 SDK.
- You can find the completed project files for this chapter on GitHub at https://github.com/Kpackt/Real-World-Implementation-of-C-Design-Patterns/tree/main/chapter-3.

The following story is fictitious

Any resemblance to actual persons, alive or dead, is a mere coincidence.

Meet Kitty and Phoebe – Texas-born sisters who share a love for riding bicycles. Phoebe is studying engineering at Southern Methodist University, a private university in the sisters' hometown of Dallas, Texas. Texas is one of the southernmost states in the United States. Kitty is studying industrial design at Sul Ross University in West Texas. Purely through kismet, they both won a summer internship at MegaBikeCorp, which is a large multi-national bicycle manufacturing company:

Figure 3.1: Kitty (left) and Phoebe (right) posing with their innovative bicycle prototype built entirely in their robotic factory.

While working at MegaBikeCorp, the sisters had the opportunity to try out several different kinds of bicycles. Phoebe, who lives in a big city, is partial to the road bike. Road bikes use a diamond-shaped frame with two large thin wheels, slick tires, and plenty of gears to help a rider deal with the steep hills they often encounter in many cities:

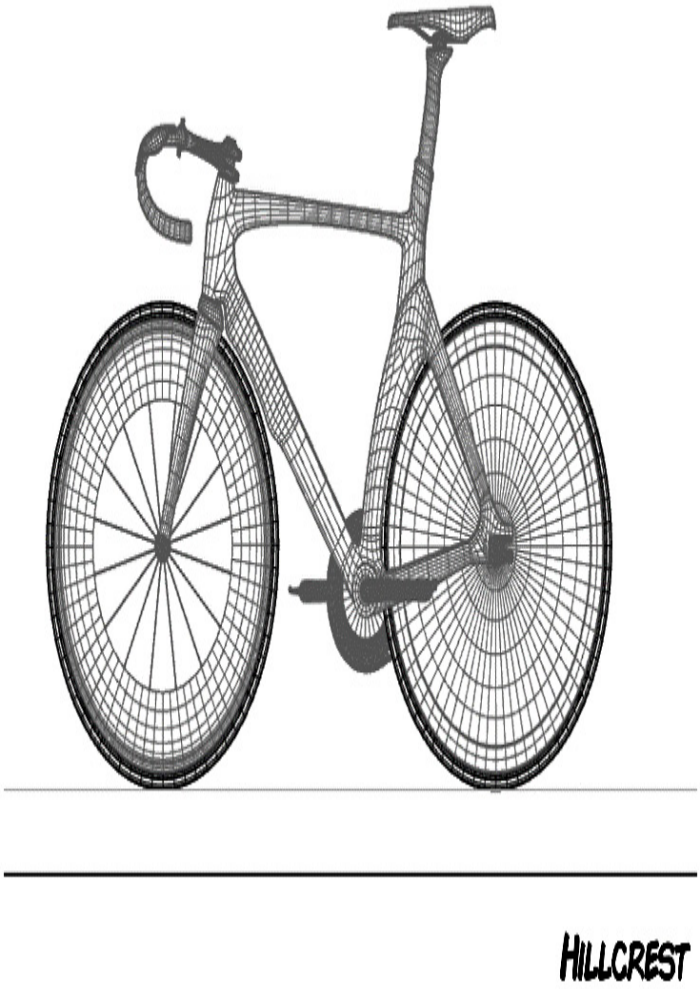


Figure 3.2 – Kitty's CAD design for a road bike she calls Hillcrest.

The rider on a road bike leans far forward on a set of dropped handlebars. This makes these bicycles fast, and every year, hundreds of riders from all over the world compete in a famous race called the *Tour de France*. In the tour, racers compete to see who is the fastest in the world. Kitty and Phoebe recalled that their mother, a huge cycling fan, would not allow anyone in the house to speak while the race was on television. Phoebe also enjoyed the tour and as she cycled through the streets of Dallas on her road bike, she would often have a desire to ride faster. This requires a different bike.

Phoebe rides a recumbent bike when she needs to satisfy her need for speed. A recumbent bicycle features the rider sitting in a comfortable seat that looks more like a lawn chair but with wheels. You can see Kitty's design in the following diagram:

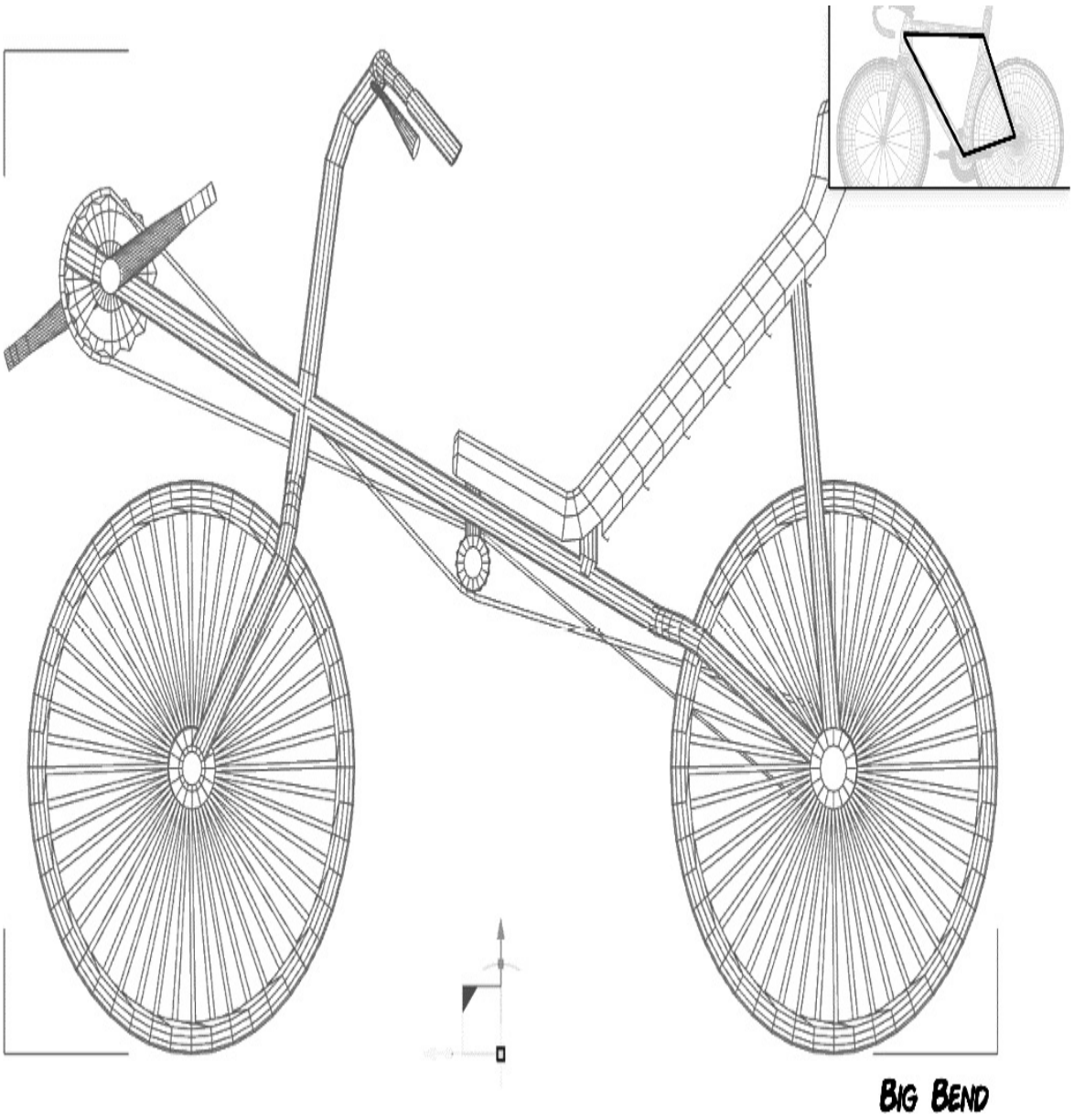


Figure 3.3 – Kitty's CAD design for a recumbent bicycle. Note that the frame is shaped differently than the diamond shape used on most other bicycles such as the Hillcrest shown in the inset. These are among the fastest bicycles ever made.

The rider leans way back and sits very low on the bicycle. The seat is stretched out beneath Phoebe. The pedals are elevated above the

front wheel to capture the power of the rider's full leg extension. These bikes can go faster than a traditional racing bicycle and they're very comfortable to ride over long distances. However, because of racing rules recognized throughout the world, they are never used for racing in events such as the *Tour de France*.

Kitty, on the other hand, enjoys riding a mountain bike, which is designed to be ridden off-road like a Jeep. Mountain bikes, like road bikes, use a diamond-shaped frame, but they have big, knobby tires and disc brakes that work well in the mud. The rider sits more upright so that they can see and react to any off-road obstacles. The standard mountain bicycles usually have fancy shock absorbers in the front, and the more expensive bikes have a separate shock absorber on the back wheel, along with the front shocks:

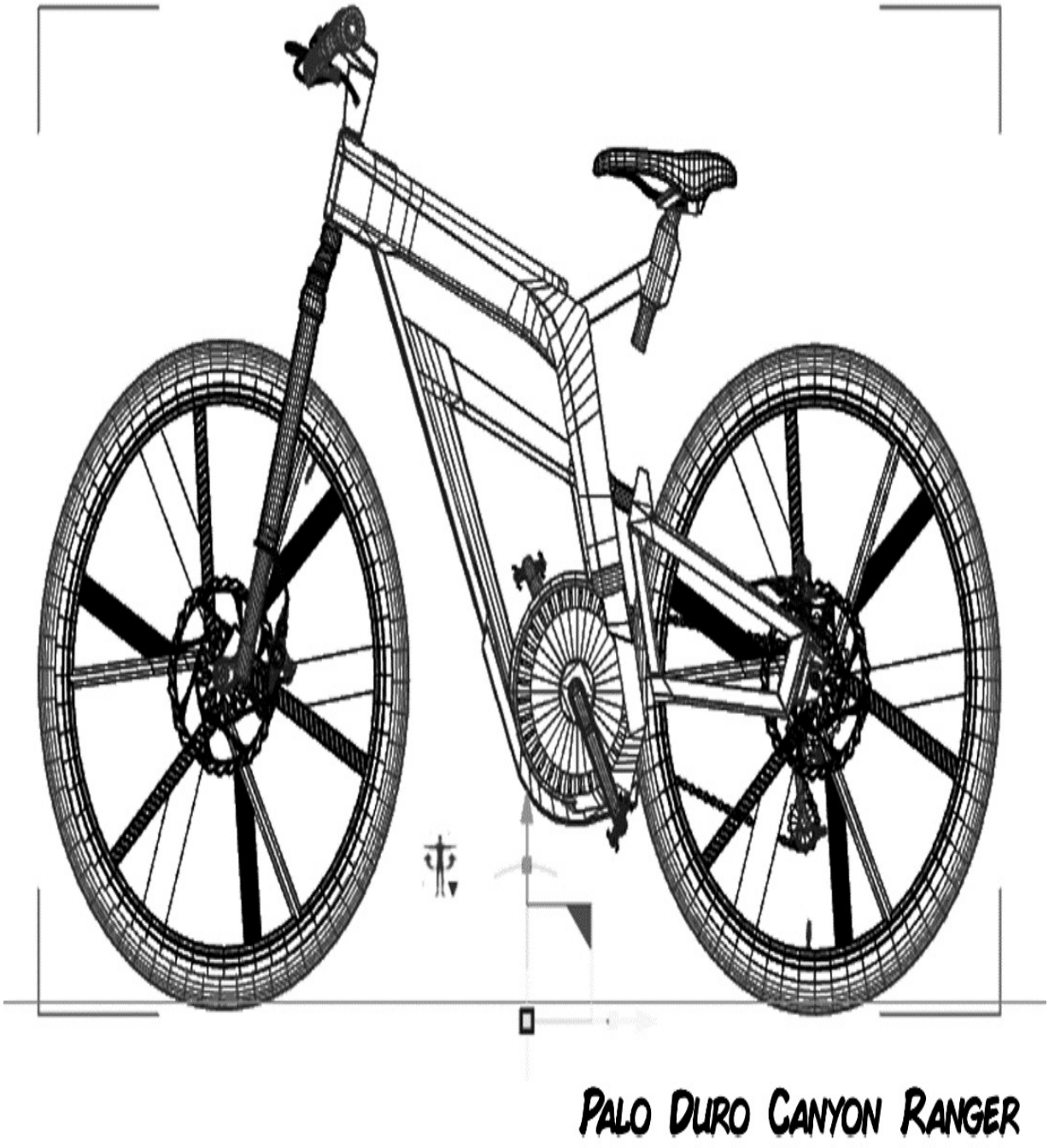


Figure 3.4 – Kitty's CAD drawing for an innovative design for a mountain bike that she calls the "Palo Duro Canyon Ranger".

Kitty also likes riding in town, but she doesn't like the uncomfortable riding position of a road bike. A recumbent bike is too dangerous to ride because it is so low to the ground that cowboys driving the big trucks that are so popular in Texas won't be able to see her. Instead, Kitty likes to ride comfort cruisers, sometimes called *granny bikes*. They are perfect for in-town riding. They let her sit fully upright in a comfortable position owing to a curved handlebar design:

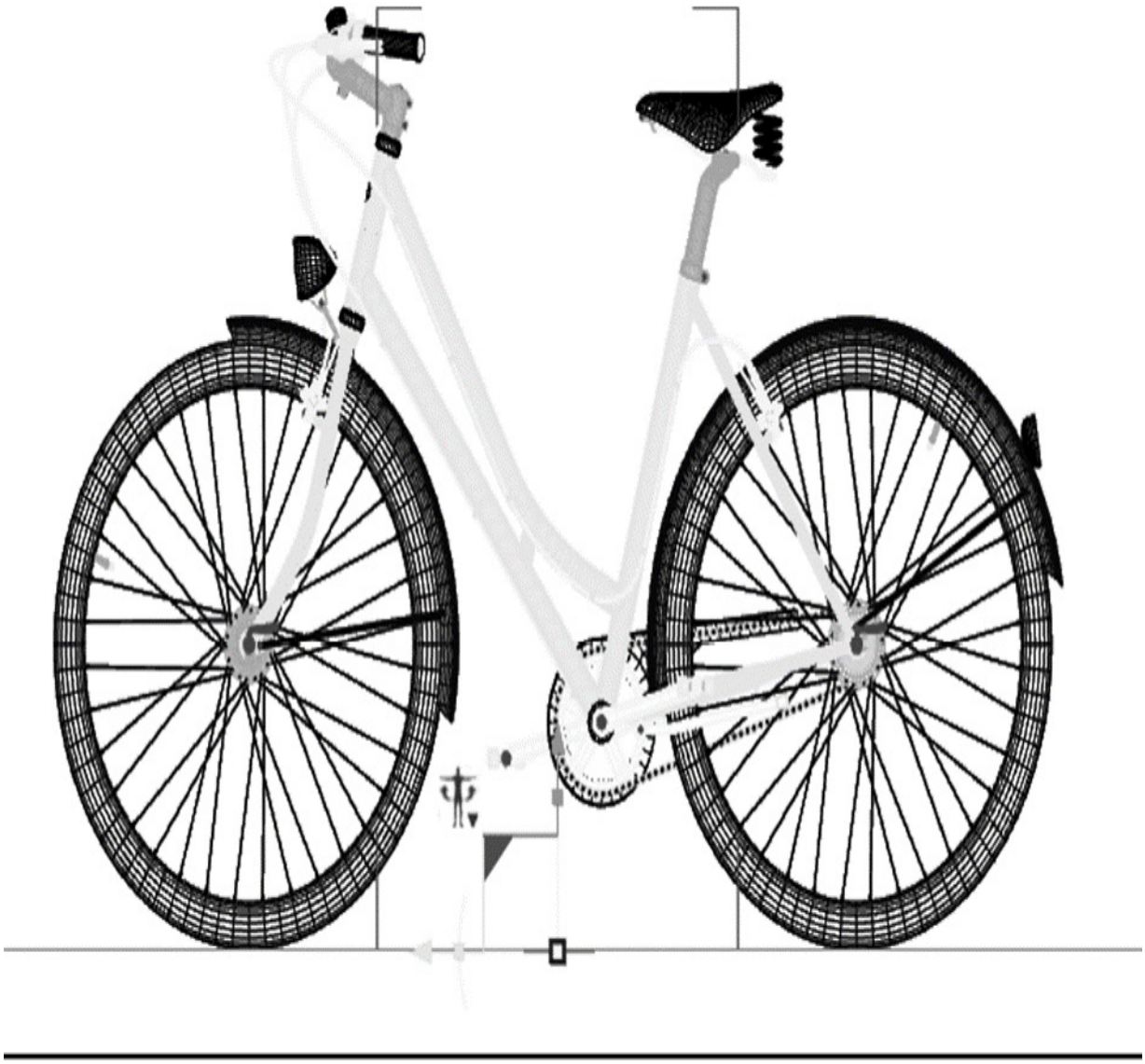


Figure 3.5 – Kitty's design for a comfort cruiser bicycle, sometimes called a granny bike. These are perfect for in-town commutes and tourists who need to get around comfortably.

The cruisers have thick tires for stability and they have a chain guard and enclosed gear system to prevent ruining the left leg of your pants, which tends to rub against the bicycle chain. These bicycles are built for comfort rather than speed or off-road prowess.

Between them, the sisters rode every model of bicycle produced by MegaBikeCorp. Phoebe found the level of engineering innovation at MegaBikeCorp uninspiring. Kitty felt MegaBike's designs were stale. "*They've been making the same four bicycles for 30 years,*" Kitty said. The sisters decided they could do better and decided to make a plan of action. After many long nights, lots of tacos, online collaboration sessions, bicycle rides, and a few skinned knees, they came up with four bicycle designs. But the sisters did not stop with bike designs. They also came up with plans for a robotic manufacturing system that could fully automate the bicycle production process. Together, they formed a start-up company called *Bumble Bikes Incorporated*, named after Phoebe's favorite childhood toy: a stuffed bumblebee she named *Bumbles*. They plan to open two manufacturing locations: one in the sisters' hometown of Dallas, Texas, and the other in Kitty's college town of Alpine, Texas. The two areas are very different geographically.

North Texas, Dallas in particular, is urban and heavily developed. Cycling is a popular sport and hobby for many North Texans. The riders in the Dallas area primarily buy traditional road bicycles. Surprisingly,

there is also a lot of interest in recumbent bicycles, especially around the large engineering colleges. You can even see recumbents ridden around the 7 million-square-foot (650 million square-meter) campus of Texas Instruments – a large semi-conductor manufacturer in Dallas. The girls decided to build their road bike and recumbent bike designs in the factory in Dallas.

In West Texas, the city of Alpine is the last bastion of civilization north of Big Bend National Park. Extreme mountain biking is popular in the Chisos mountain basin and the old mining and cattle roads that criss-cross the desert. Alpine is a small town and doesn't have or need any mass transit services. Kitty's research indicates there is a market for comfortable "cruiser" bicycles that are ideal for getting around town. Kitty decides to manufacture the mountain bike and the cruiser models in Alpine. Phoebe and Kitty, with their initial plans in hand, set to work.

Phoebe tasked herself to work on the robotics to be used in manufacturing the bicycles. Kitty settled down with her favorite C# IDE and got to work on the control software that would ultimately control Phoebe's robotics. She knows that her designs, along with Phoebe's engineering, will produce the best bicycles the industry has ever seen. Kitty is planning on their company and the software that runs it being wildly successful, which, as we learned in *Chapter 1*, *There's a Big Ball of Mud on Your Plate of Spaghetti*, is the worst-case scenario.

Kitty's starting point in writing the code for her robotics control software is going to be modeling out the basic class structure for very generic bicycle objects. This modeling will form the basis of her entire company's future. Modeling the classes well the first time will put Kitty and Phoebe riding their bicycles down the road to success!

Let's follow the exploits of the sisters, Kitty and Phoebe, as they embark on building the automated plant's robotic control systems. We'll focus on creational patterns, which are just what they sound like: patterns for controlling the instantiation of objects.

The initial design

Kitty knows she wants to model a bicycle for her software, and she wants to design her models in a way that maximizes each class's flexibility. The plan for the start-up is to develop four bicycles with a future expansion to include exotic bicycles and custom builds.

Kitty opens her IDE and creates a class library project to hold her classes since she knows she's probably going to use these in several different programs. She calls the class library **`BumbleBikesLibrary`** which you'll find in the sample code for this chapter.

She decides to start with an abstract class and to use inheritance to define her bicycle models for each of the four types of bicycles she

intends to initially design and manufacture. *Figure 3.6* shows the result of her effort. This set of properties can be used to define nearly any type of bicycle:

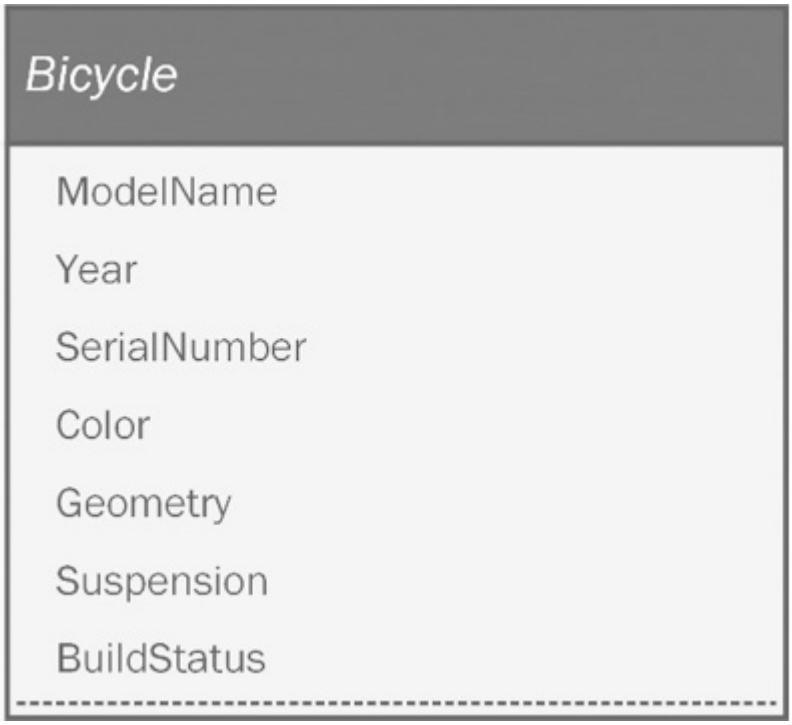


Figure 3.6 – The abstract bicycle model.

Let's take a look at these properties in more detail:

- **`ModelName`**: The name of the model as it will appear on the company website.
- **`Year`**: The model year for the bicycle. All bicycle designs should be updated every 2 years to prevent the designs from becoming stale like MegaBikeCorp's.
- **`SerialNumber`**: A unique identifier for each bicycle that rolls off the assembly line.

- **`Color`**: The color of the bicycle. To keep initial costs low, Kitty will define an enumeration with a limited set of colors for each model.
- **`Geometry`**: This refers to the frame configuration for the bicycle. All the bikes for the initial manufacturing run are either upright or recumbent geometries. This can be moved into an enumeration as well.
- **`Suspension`**: This refers to the type of shock absorbers used on the bicycle. Shocks are most important on mountain bikes, but you also find them on road bicycles, recumbents, and some cruisers. It could be argued that this property is only appropriate for a mountain bike subclass, but Kitty knows about "analysis paralysis." Rather than trying to get the model perfect on the first go-around, she decides to put it in the superclass for now. She can always refactor it later when things become more concrete, or if she gets a whiff of smelly code structure.
- **`BuildStatus`**: Kitty knows her robotics control system needs to understand the current state of the bicycle build process, so she decides to include an enumerated property to hold this information.

Several properties can be expressed as an enumeration, which is a fixed collection. Back in the old days, we used "magic numbers" to represent finite lists. The programmers would assign an integer to each value. As an example, to represent a finite list of suspension types, we might be tempted to number them 0 through 3, since there are four possibilities. This made maintenance difficult because every-

one had to remember what each number meant. Was it 0 for a full suspension? What was the code for a hardtail? Later, we wised up and defined constants for everything:

```
const int FULLSUSPENSION = 0;
const int FRONTSUSPENSION = 1;
const int SEATPOSTSUSPENSION = 2;
const int HARDTAIL = 3;
```

This is better, but it isn't very object-oriented. You wind up with dozens of lines of ugly constant definitions in all caps. You might be tempted to use an integer type to model them since you are presented with the opportunity to assign an arbitrary integer as a value. What will stop someone from setting it to 100 when the potential values are supposed to be from 0 to 3? We could write encapsulation logic to enforce our range of 0 to 3. That's all well and good, but we'd have to constantly alter the accessor logic every time we add a new suspension to the line-up. Thankfully, in C#, we have **enumerations**. Enumerations allow you to define integer values like these as a set with easy names tied to the enumeration type name itself. This is used instead of declaring an integer type. In UML, they look like a class diagram except for a giant **<<Enumeration>>** tag at the top of the box. Our bicycle class uses four enumerations, as shown here:

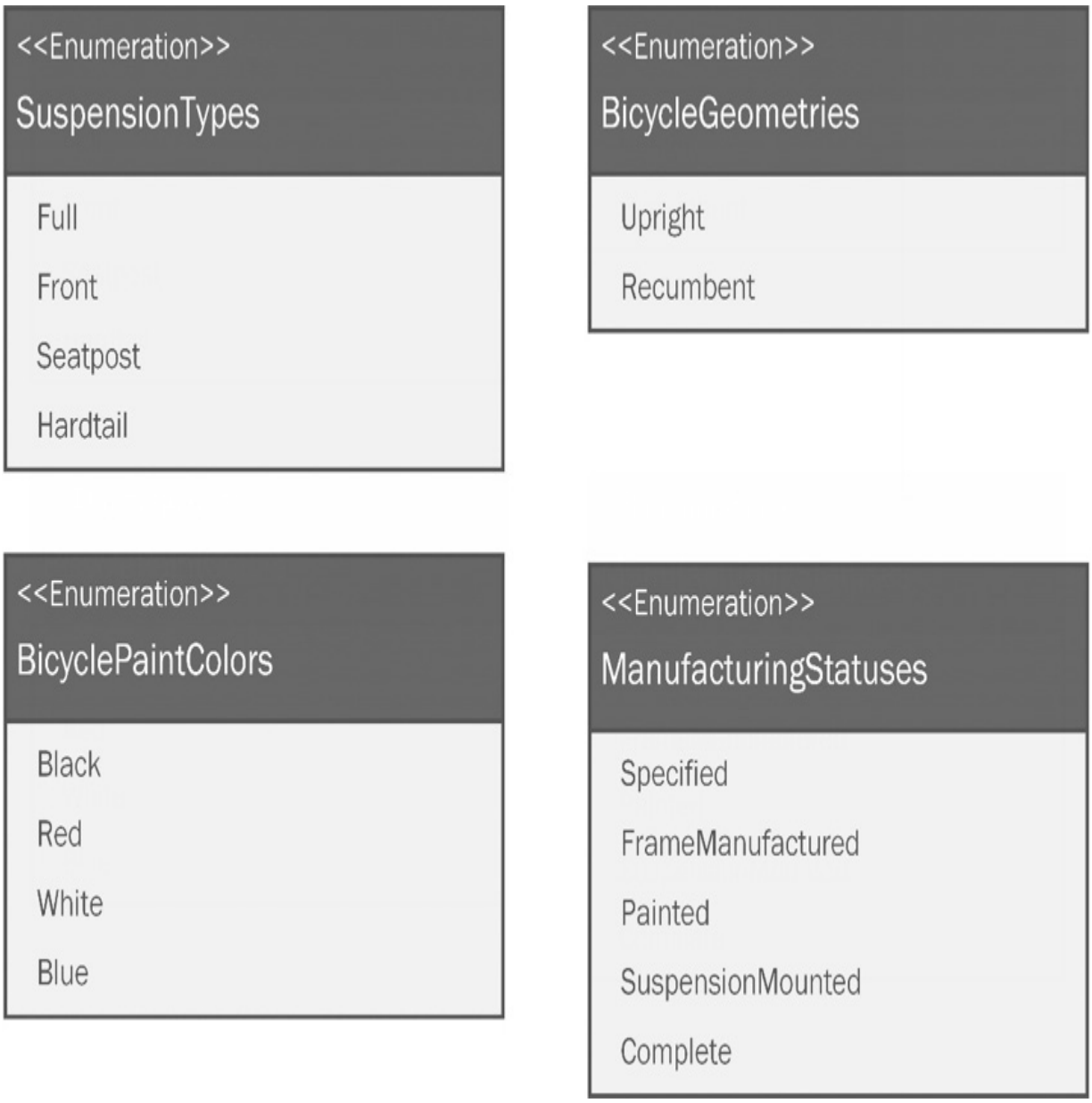


Figure 3.7 – Four enumerations are defined to limit the options available for our abstract bicycle's geometry, suspension, color, and current manufacturing status. Enumerations help us keep our object's state clean, valid, and easy to read.

Before getting into patterns, let's begin by creating what we have diagrammed so far. First, Kitty will make the four enumerations:

- **SuspensionTypes**
- **BicycleGeometries**
- **BicyclePaintColors**
- **ManufacturingStatus**

Here is the **BicycleGeometries** enumeration:

```
public enum BicycleGeometries
{
  Upright, Recumbent
}
```

Next, she makes an **enum** for suspension types:

```
public enum SuspensionTypes
{
  Full, Front, Hardtail
}
```

Next comes the paint colors:

```
public enum BicyclePaintColors
{
```

```
    Black, Red, White, Blue
  }
```

Finally, Kitty makes an **`enum`** for manufacturing status codes:

```
public enum ManufacturingStatus
{
  Specified, FrameManufactured, Painted, Suspens
   Complete
}
```

With those out of the way, let's see what the **`Bicycle`** base class looks like:

```
public abstract class Bicycle
{
  protected string ModelName { get; init; }
  private int Year { get; set; }
  private string SerialNumber { get; }
  protected BicyclePaintColors Color { get; init
  protected BicycleGeometries Geometry { get; in
  protected SuspensionTypes Suspension { get; in
  private ManufacturingStatus BuildStatus { get;
```

Kitty creates the **`Bicycle`** class as an abstract class, just as she did in the UML model. Note that the name of the class in *Figure 3.6* is *italicized*, indicating it is abstract. Next, she adds the properties. The UML model purposefully left out the access modifiers and types, leaving those as implementation details for the programmer to determine. In this case, as it is with many start-up projects, the architect and developer are the same person.

Kitty defines several of the properties as being protected because she intends to manipulate them from the subclasses. A few of the properties are marked **`private`** since it is appropriate to manipulate those at a higher level.

Next, she moves on to the constructor, which is the function that is run with instantiation using the **`new`** keyword:

```
public Bicycle()
{
  ModelName = string.Empty;
  SerialNumber = Guid.NewGuid().ToString();
  Year = DateTime.Now.Year;
  BuildStatus = ManufacturingStatus.Specified;
}
```

Kitty sets the default values for each property within the constructor. The model name, for now, is empty. She'll change that in the sub-class. The serial number property is a generated GUID, which is a string that is guaranteed to always be unique. The **`Year`** property is set to the current year, and **`BuildStatus`** is set to the first status in the enumeration.

The last step is to add the **`Build`** method. For now, the **`Build`** method will just print to the console to show the logic is working correctly. Eventually, this can be substituted for the more complex control logic for Phoebe's robotic systems:

```
public void Build()
{
  Console.WriteLine($"Manufacturing a {Geometry
  BuildStatus = ManufacturingStatus.FrameManufa
  PrintBuildStatus();

  Console.WriteLine($"Painting the frame {Color
  BuildStatus = ManufacturingStatus.Painted;
  PrintBuildStatus();

  if (Suspension != SuspensionTypes.Hardtail)
```

```
      {
        Console.WriteLine($"Mounting the {Suspensio
        BuildStatus = ManufacturingStatus.Suspensio
        PrintBuildStatus();
      }
      Console.WriteLine("{0} {1} Bicycle serial nur
      BuildStatus = ManufacturingStatus.Complete;
      PrintBuildStatus();
    }
```

The abstract class is done! Next, Kitty needs to create the concrete subclasses for the bicycles they intend to build:

- **`RoadBike`**
- **`MountainBike`**
- **`Recumbent`**
- **`Cruiser`**

First, she makes the **`RoadBike`** class:

```
public class RoadBike : Bicycle
{
  public RoadBike()
  {
    ModelName = "Hillcrest";
    Suspension = SuspensionTypes.Hardtail;
```

```
      Color = BicyclePaintColors.Blue;
      Geometry = BicycleGeometries.Upright;

    }
  }
```

The **`RoadBike`** class inherits from **`Bicycle`** and the constructor sets the defaults for the class. The model name is **`Hillcrest`**, after the name of the street that runs West of Phoebe's university campus. Road bikes typically lack shock absorbers, so she defines the suspension type as **`Hardtail`**. The initial models only come in one color, and this one comes in blue. Since this isn't a recumbent, the geometry is set to **`Upright`**.

With Phoebe's favorite road bike out of the way, Kitty sets to modeling her favorite – the mountain bike:

```
public class MountainBike : Bicycle
{
  public MountainBike()
  {
    ModelName = "Palo Duro Canyon Ranger";
    Suspension = SuspensionTypes.Full;
    Color = BicyclePaintColors.Black;
    Geometry = BicycleGeometries.Upright;
  }
```

```
}
```

Kitty names the mountain bike after the Palo Duro Canyon located in the Texas panhandle. Few people know that the Palo Duro Canyon is the second-largest canyon in the United States, after the Grand Canyon. Palo Duro Canyon is home to some of the best mountain biking in Texas. She wants an aggressive look for her design, so she goes with **`Black`** for the color. Naturally, not being one to skimp, she creates the design with full suspension – shocks on the front and rear of the bicycle to handle any obstacle the trail might present. As mentioned previously, given it isn't recumbent, the geometry is defined as **`Upright`**.

Now, the **`Recumbent`** bicycle class needs to be created:

```
public class Recumbent : Bicycle
{
  public Recumbent()
  {
    ModelName = "Big Bend";
    Suspension = SuspensionTypes.Front;
    Color = BicyclePaintColors.White;
    Geometry = BicycleGeometries.Recumbent;
  }
}
```

Kitty decides to call this bike *Big Bend*. The Big Bend area is a desert with a mountain range, which has a host of mountain biking opportunities. However, there are great stretches of paved and unpaved roads that are more or less straight. Recumbents do well in those environments as their design allows riders to go faster and farther without getting tired. The recumbents are also pretty good at pulling trailers. A person who desires to camp can carry adequate water and supplies. Since the bicycle's namesake has paved and unpaved roads, Kitty opts for a front suspension. Recumbents already have a nice seat, so rear suspension doesn't add many benefits, and it would make the bicycle more expensive. Naturally, the geometry is set to **`Recumbent`**.

The final bicycle is the cruiser:

```
public class Cruiser : Bicycle
{
    public Cruiser()
    {
        ModelName = "Galveston Cruiser";
        Suspension = SuspensionTypes.Hardtail;
        Color = BicyclePaintColors.Red;
        Geometry = BicycleGeometries.Upright;
    }
}
```

Kitty names this bicycle the *Galveston Cruiser* after fondly remembering her seaside vacations with her family. Galveston is a medium-sized city on the Gulf of Mexico. Galveston has beaches, a pier for commercial and vacation cruise ships, and a lively historical district called *The Strand*. The Strand is loaded with one-of-a-kind shops, cafes, bars, museums, and fun activities for tourists. Parking at the Strand is expensive, and often difficult to find. Galveston's Strand is the perfect place to ride a cruiser, aside from the less known streets of Alpine.

No pattern implementation

Kitty is on a roll! You know how it gets. She knocked out the enumerations, base class, and subclasses in nary an hour. She's seriously *cruising* along and she doesn't want to lose velocity. Kitty gives in to the temptation to write this code for the final implementation of the main entry point for the program:

```
using BumbleBikesLibrary;
const string errorText = "You must pass in mounta
```

We take in an argument from the command-line program and use that to determine what to make. If a string was passed in, the length

of **`args`** will be greater than zero and we can do our thing. Otherwise, we can admonish our foolish users for thinking our software can read their minds:

```
if(args.Length > 0)
{
```

It's a good idea to trim and normalize your command-line input. This means we ignore extra spaces in front of and after the argument. We ignore the case by forcing everything to either upper or lowercase so that we can compare the input with our expected values. The comparison works regardless of whether the user passed in *mountainbike*, *MOUNTAINBIKE*, or even *mOuNtAiNbIkE*:

```
    var bicycleType = args[0].Trim().ToLower();
    Bicycle bikeToBuild;
```

Next comes a **`switch`** statement based on the input. The input determines what to build and returns the corresponding class instance:

```
    switch (bicycleType)
    {
      case "mountainbike":
```

```
        bikeToBuild = new MountainBike();
        break;
      case "cruiser":
        bikeToBuild = new Cruiser();
        break;
      case "recumbent":
        bikeToBuild = new Recumbent();
        break;
      case "roadbike":
        bikeToBuild = new RoadBike();
        break;
```

If the user passes an argument that we haven't accounted for in the switch, such as **`MotorCycle`** or **`PeanutButter`**, we write and throw an exception:

```
      default:
        Console.WriteLine(errorText);
        throw new Exception("Unknown bicycle type:
    }

    bikeToBuild.Build();
  }
```

If no arguments were passed into the command-line program, we show an error message instructing the user to supply the required argument:

```
else
{
  Console.WriteLine(errorText);
}
```

Effectively, Kitty is taking a command-line argument and using the software's main entry point, **`Program.cs`**, as the class that will instantiate the correct bicycle type. She tries it out and it works! Kitty has cleverly leveraged Liskov substitution by using the abstract base class as her type for the **`bikeToBuild`** variable, which allows her to instantiate the appropriate subclass based on what kind of bike she wants to build. Kitty would not be able to do that if she was working at MegaBikeCorp. She would probably have a pointy-haired boss who would tell her to clean up the code and ship it. Thankfully, she's in business for herself as a nagging voice in the back of her mind decries, "*You can do better.*"

Kitty's first implementation can be found in the **`NoPattern`** project within this chapter's sample source code.

The Simple Factory pattern

Kitty decides to do a little research on patterns. She wasn't a computer science major in college and had only heard of patterns in her cod-

ing class. Kitty looks around and finds some blog articles on something called **the Simple Factory pattern**. Perfect, she thinks. This being her first coding project since college, and since she has a lot *riding* on her code (see what I did there?), she decides that something with *simple* right there in the name is a good place to start.

According to the blogs, all she has to do is move her instantiation logic into its own class, called a **factory class**. This is done, say the articles, to decouple the instantiation logic from the main program. This should get her closer to honoring the open-closed principle and should make her code more flexible.

She returns to her IDE and adds a class called `SimpleBicycleFactory` and moves her instantiation logic there. The logic is the same as that shown previously:

```
public class SimpleBicycleFactory
{
  public Bicycle CreateBicycle(string bicycleType
  {
    Bicycle bikeToBuild;
    switch (bicycleType)
    {
      case "mountainbike":
        bikeToBuild = new MountainBike();
        break;
```

```
      case "cruiser":
        bikeToBuild = new Cruiser();
        break;
      case "recumbent":
        bikeToBuild = new Recumbent();
        break;
      case "roadbike":
        bikeToBuild = new RoadBike();
        break;
      default:

        throw new Exception("Unknown bicycle type
    }
    return bikeToBuild;
  }
}
```

Then, Kitty refactors her **`Program.cs`** file to use the simple factory:

```
using SimpleFactoryExample;

const string errorText = "You must pass in mounta

if (args.Length > 0)
{
  var bicycleType = args[0].Trim().ToLower();
```

Here's the different part – Kitty uses the **`SimpleBicycleFactory`** class instead of directly running a **`switch`** statement:

```
    var bicycleFactory = new SimpleBicycleFactory()
    var bikeToBuild = bicycleFactory.CreateBicycle(
    bikeToBuild.Build();
}
else
{
    Console.WriteLine(errorText);
}
```

A refactor, by definition, means you are improving the structure or performance of your code without introducing any new features. Kitty has accomplished this by making her code a little more elegant.

When the logic is free-range in the **`Program.cs`** file, it is locked into being used in only that program. Kitty wisely realizes that she and, undoubtedly, Phoebe will want to create other programs that can use the creation logic. Encapsulating the logic into a class is an obvious improvement.

Kitty, before closing her laptop for the evening, notices a post on social media from her programming professor. She asks how he's been

and while catching up, she mentions her code project. The prof says he'd love to see it, so she sends him a GitHub link.

The code Kitty's professor is going to review can be found in the `SimpleFactoryPattern` project within this chapter's sample source code.

The Factory Method pattern

Kitty's old professor looks at the code and tells her this newer code is an improvement, but she isn't using a pattern. The simple factory is classified as a **programming idiom**. Idioms are like patterns, in that they occur frequently. You recognize them when you see one, but they haven't fully realized any solutions to common problems. Perhaps the most famous programming idiom ever devised was created in Kernighan and Ritchie's book titled *The C Programming Language*, also known as *The K&R book*. It was in this book we saw our very first *Hello, World* program. *Hello, World* is an idiom. It usually serves as the first few lines of code you try when you are learning a new language. It doesn't solve any industrial-grade problems by encouraging flexibility and code reuse.

Kitty realized she'd seen this idiom in her IDE of choice for C#, JetBrains *Rider*. When you create a console application, like the one

she's been working on, the first thing you see is this code , which is generated by the IDE as a starting point for the project:

```
Console.WriteLine("Hello, World!");
```

I have included an example of a **`Hello, World`** program. You'll find it in the **`HelloWorld`** project, located within the chapter's source code. Yes, I did include a **`Hello, World`** program because I'm all about attention to detail. Now, back to the story.

The next day, Kitty sets herself to fully researching the creational patterns. She wanted to do this the right way. After some reading, Kitty found several patterns called factory patterns:

- The simple factory, which we've established isn't a pattern, but it is often mistaken for one.
- The Factory pattern, which is a mild improvement over the simple factory.
- The Factory Method pattern, which truly abstracts the creation process when you have a variety of object types to instantiate.
- The Abstract Factory pattern, which, for now, seems more complex than necessary since it deals with creating groups of objects.

Kitty settled on what she thought would be the perfect pattern for her problem: **the Factory Method pattern**. She also thought it ironic that

the starting point for her software design was working with a factory pattern, given she was modeling what would one day be a physical factory. The factory patterns are so-called because they take a set of inputs and produce a concrete object as output. The simple factory accomplished this at a superficial level, but there are some problems with relying on the idiom in place of the real pattern. The idiom isn't as flexible as it could be. Bumble Bikes Inc. is going to have two factory locations making different types of bicycles. The simple factory can create any bicycle, but at the same time, it's locked into making all four. Relate that to a real factory and you can see it might be wasteful to require the factory to make every kind of bicycle, instead of the two that it will be making.

In our software design, the factory shouldn't have any knowledge of what it is going to create. It should be flexible enough to make any kind of bicycle. We can structure this so that we can create a factory capable of producing a subset of all the subclasses. The subclasses decide what can and should be made concrete. We can diagram the generic factory method pattern as so:

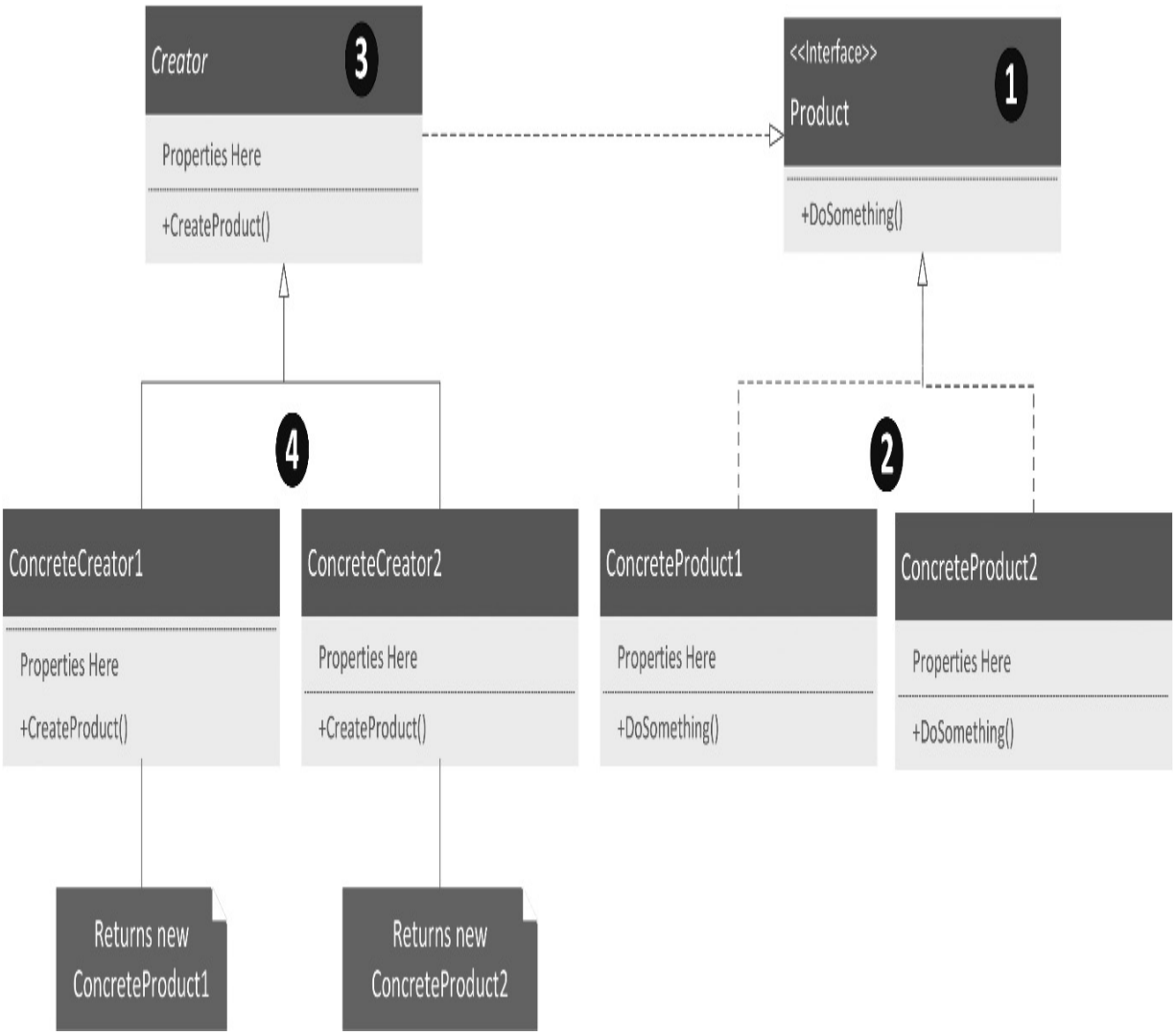


Figure 3.8 – A generic diagram of the Factory Method pattern.

Let's review each part of the diagram based on the numbers:

1. The Factory Method pattern starts with an interface that defines a common behavior or set of behaviors. In general, using interfaces is more flexible than using a base class because you aren't limited by the inheritance rules in C. This is to say that any child class in C# may have only one parent class. Multiple class inheritance is

not supported. In the case of interfaces, any class may implement as many different interfaces as are needed.

2. When we discuss the Factory Method pattern, we call the objects the factory creates **`products`**. These are the concrete products the factory will produce. They will all implement the common product interface. In practice, you don't need to stick with the same names, as shown in the preceding diagram.
3. A factory method has a **`Creator`** class that houses the factory method itself. The Factory Method is coded to return the **`Product`** interface so that it can return any product that implements that interface. It isn't tied to a particular abstract base class, as was the case with Kitty's original refactor. These creators are abstract and are meant to be overridden in concrete creator subclasses. This is what provides the flexibility we need with our bicycle factory.
4. Concrete creators provide the actual concrete classes. All your creation logic will be here.

Let's remember the specifics of Kitty and Phoebe's plan. It calls for two factories – one in Dallas to make road and recumbent bicycles and another in Alpine to make mountain and cruiser bicycles. Kitty heads to her whiteboard and makes her version of the preceding diagram:

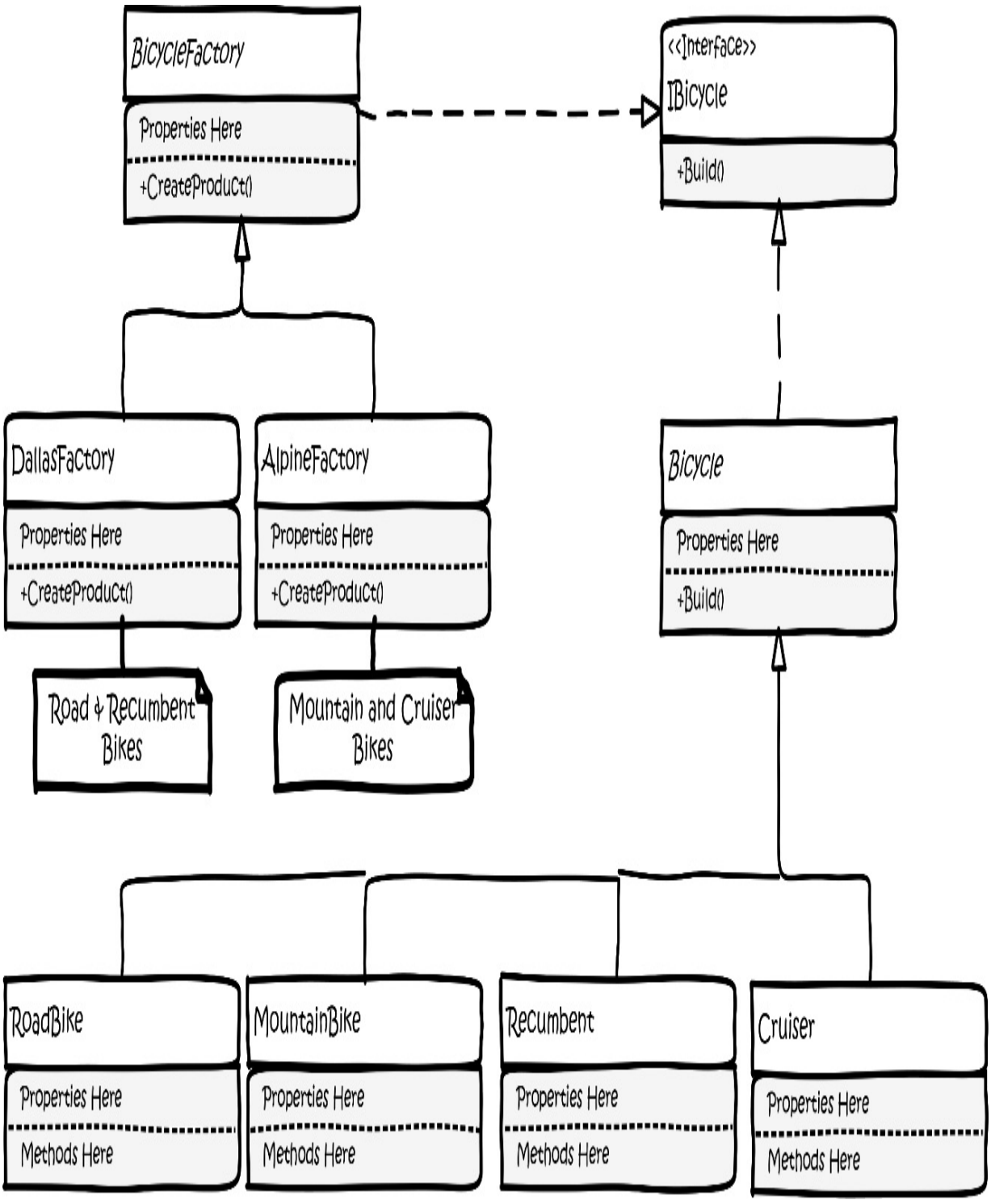


Figure 3.9 – Kitty's whiteboard drawing of her Factory Method pattern design idea.

This looks pretty good! Kitty decides to move a lot of what is in her abstract bicycle class into an interface she calls **`IBicycle`**. This doesn't mean she should throw out the abstract class, but it's easy to have the abstract class implement the interface. Once she does that, she can pass the interface around, which is more flexible than using the base class.

The abstract bicycle base class won't change, other than implementing the **`IBicycle`** interface. None of the bicycle subclasses change at all.

She does need to add some creator classes. She'll need to create an abstract **`BicycleCreator`** class, which will be subclassed by as many concrete creation classes as she might need.

This fits the design problem because we need to model two actual factories. One is called **`DallasCreator`**, which will make road bikes and recumbent bikes, and the other is called **`AlpineCreator`**, which will produce mountain bikes and cruisers.

Our design is closed for modification. We never need to mess with the base classes and interface again. However, the design is also open for extension. As the line of products expands in the future, we can continue to add factories and each factory can specialize in any

set of products. New bikes can be added by simply creating new sub-classes of **`Bicycle`**.

There's nothing left but the typing.

Kitty adds the **`IBicycle`** interface to her **`class`** library:

```
public interface IBicycle
{
  public string ModelName { get; set; }
  public int Year { get; }
  public string SerialNumber { get; }
  public BicycleGeometries Geometry { get; set; }
  public BicyclePaintColors Color { get; set; }
  public SuspensionTypes Suspension { get; set; }
  public ManufacturingStatus BuildStatus { get; s
  public void Build();
}
```

Then, she modifies the **`Bicycle`** base class. Defining non-public members is not possible in an interface. C# generally requires properties and methods defined in an interface to be **`public`**. Non-public members in an interface don't make sense. They would be housing implementation details, not something for public consumption, which is the point of an interface. The bottom line is that if we want to require anything on the **`Bicycle`** base class, we will need to change the

access modifiers to **`public`**. We usually want to avoid changing classes once they are published in production. At this stage, we haven't done that yet. We can either change the access modifiers or avoid defining those elements in the interface and just have the **`build`** function. Kitty decides on the more complete version of the interface she's already typed:

```
public abstract class Bicycle : IBicycle
{
  protected Bicycle()
  {
    ModelName = string.Empty; // will be filled i
    SerialNumber = new Guid().ToString();
    Year = DateTime.Now.Year;
    BuildStatus = ManufacturingStatus.Specified;
  }

  public string ModelName { get; set; }

  public int Year { get; }
  public string SerialNumber { get; }
  public BicyclePaintColors Color { get; set; }
  public BicycleGeometries Geometry { get; set; }
  public SuspensionTypes Suspension { get; set; }
  public ManufacturingStatus BuildStatus { get; s
```

Next, we need our creator classes. Kitty starts with the abstract class, which she'll call **`BicycleCreator`**:

```
using BumbleBikesLibrary;
namespace FactoryMethodExample;
public abstract class BicycleCreator
{
    public abstract IBicycle CreateProduct(string
}
```

Next comes the two concrete creator classes, beginning with **`DallasCreator`**:

```
using BumbleBikesLibrary;
namespace FactoryMethodExample;
public class DallasCreator : BicycleCreator
{
    public override IBicycle CreateProduct(string
    {
        return modelName.ToLower() switch
        {
            "hillcrest" => new RoadBike(),
            "big bend" => new Recumbent(),
            _ => throw new Exception("Invalid bic
        };
    }
```

```
    }
}
```

This is followed by the **`AlpineCreator`** class:

```
using BumbleBikesLibrary;
namespace FactoryMethodExample;
public class AlpineCreator : BicycleCreator
{
    public override IBicycle CreateProduct(string
    {
        return modelName.ToLower() switch
        {
            "palo duro canyon ranger" => new Mour
            "galveston cruiser" => new Cruiser(),
            _ => throw new Exception("Invalid bic
        };
    }
}
```

Kitty needs a quick test, so she adds this code to **`Program.cs`**:

```
using FactoryMethodExample;
Console.WriteLine("Let's make some bicycles");

var dallasBicycleFactory = new DallasCreator();
```

```
var phoebesBike = dallasBicycleFactory.CreatePro
phoebesBike.Build();

var alpineBicycleFactory = new AlpineCreator();
var kittysBike = alpineBicycleFactory.CreateProdu
kittysBike.Build();
```

Now is the moment of truth. Kitty hits the run button in the IDE and moves to the edge of her seat as her code compiles:

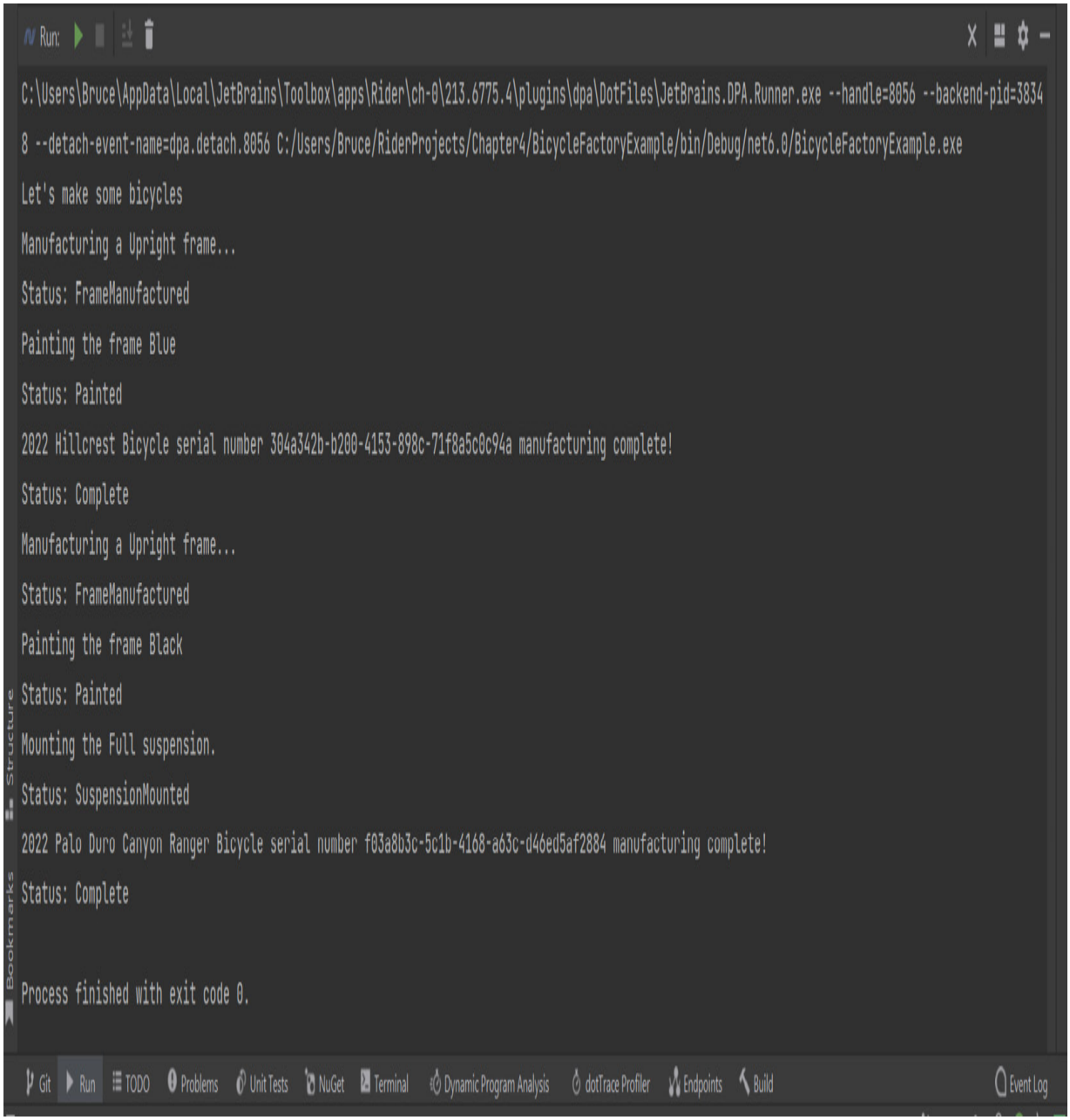


Figure 3.10 – Booyah! It works! The Factory Method pattern is running in Kitty's code!

Kitty commits and pushes her code, which you can review in the **`FactoryMethodExample`** project within the source code for this chapter.

Don't forget that the **`IBicycle`** interface was added to the **`BumbleBikesLibrary`** project.

The Abstract Factory pattern

After Kitty finishes her initial design using the factory method, Phoebe checks Kitty's work on GitHub. Phoebe has managed to finish the tooling that creates the frames and she's hard at work on some of the other parts that go into making a bicycle.

"*Kitty!,*" Phoebe says, "*This code will allow us to make a bicycle object but that's a little bit too abstract. A bicycle is made of lots of different parts.*" After a long discussion, the two decided to concentrate on manufacturing the bicycle frame and the handlebars for each bicycle type. The other parts, such as the wheels, tires, brakes, and gears, can be outsourced for the initial production of the bicycles.

It occurs to Phoebe that these parts can be made in families. The road bike uses dropped handlebars, while mountain bikes use a flat handlebar design. You shouldn't interchange these parts. Flat handlebars on a road bike create a new class of bicycle called a *gravel bike* or *hybrid*. We're not interested in changing our product line-up yet. Drop handlebars on a mountain bike make no sense at all and are dangerous. It makes sense for the physical bicycle factory to mirror

the software pattern. The sisters conclude a better fit might be to use the **Abstract Factory pattern**.

This is a pattern a lot of people get wrong. It is a common misconception that the Abstract Factory pattern simply involves making your factory class abstract. Not so! The Abstract Factory pattern is designed to create objects that are related and to decouple those objects from the client's dependency on a concrete type.

Our bicycle design consists of four families of bicycles:

- Road bicycles (the Hillcrest)
- Mountain bicycles (The Palo Duro Canyon Ranger)
- Recumbent bicycles (The Big Bend)
- Cruiser bicycles (The Galveston Cruiser)

Each type has a particular type of frame, as well as a different design for handlebars. We can say we will be making four *families* of bicycle components. When you encounter a problem that involves families of related objects, you should automatically think about the abstract factory pattern.

A second benefit of the abstract factory pattern is that it decouples the client's dependency on any particular concrete object. Let's look at the following diagram:

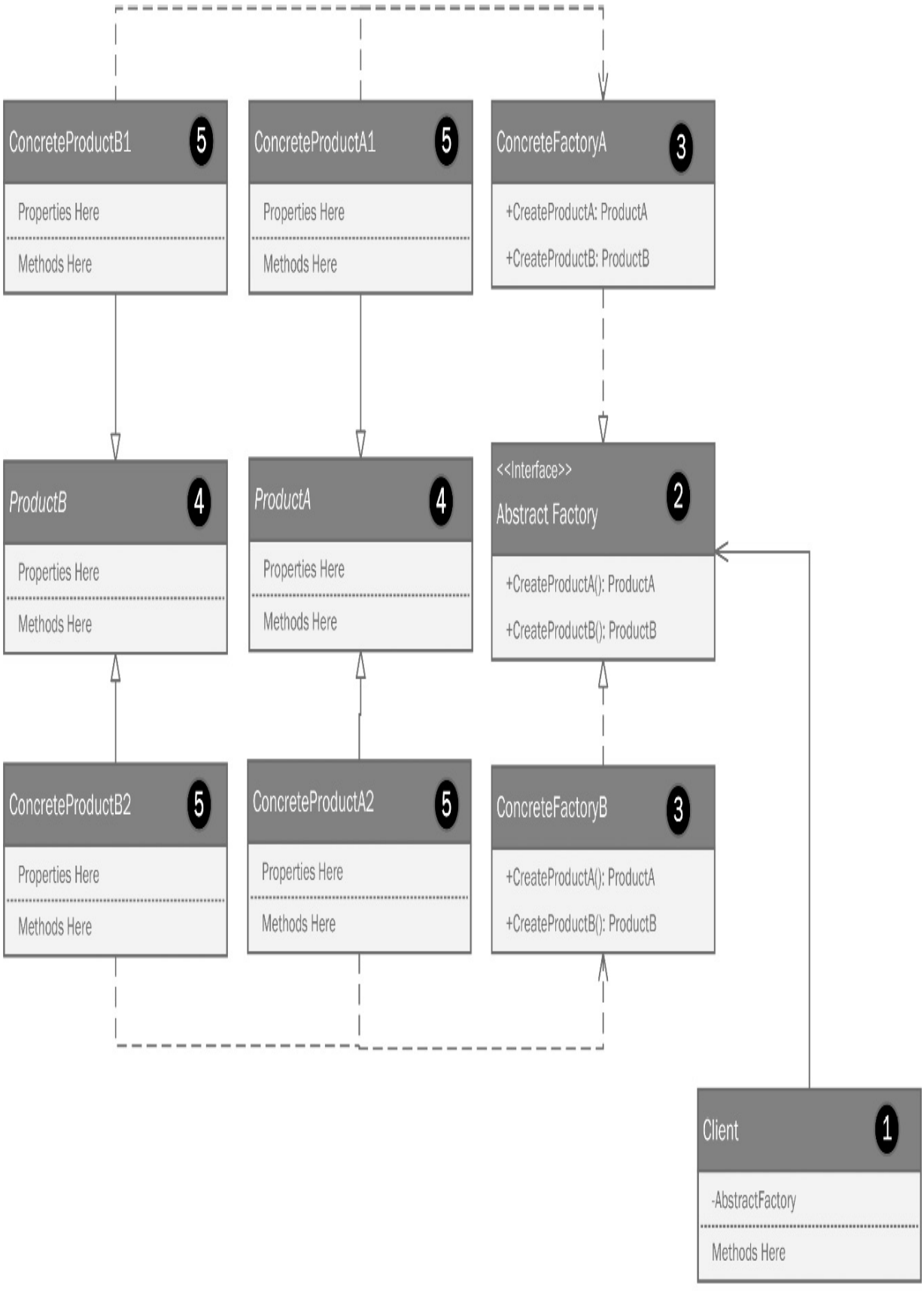

ConcreteProductB1
5
Properties Here
Methods Here
ConcreteProductA1
5
Properties Here
Methods Here
ConcreteFactoryA
3
+CreateProductA: ProductA
+CreateProductB: ProductB
ProductB
4
Properties Here
Methods Here
ProductA
4
Properties Here
Methods Here
<<Interface>>
Abstract Factory
2
+CreateProductA(): ProductA
+CreateProductB(): ProductB
ConcreteProductB2
5
Properties Here
Methods Here
ConcreteProductA2
5
Properties Here
Methods Here
ConcreteFactoryB
3
+CreateProductA(): ProductA
+CreateProductB(): ProductB
Client
1
-AbstractFactory
Methods Here

Figure 3.12 – The Abstract Factory Pattern.

I have drawn the parts going from right to left, beginning with the client:

1. The client is whatever code consumes the objects created by the Abstract Factory. Here, I'm showing a private reference to an Abstract Factory as a property on the client object.
2. The client is dependent on the **`AbstractFactory`** interface. This interface defines two methods. Normally, I don't put the return types in the UML, but in this case, it is truly important. The interface is going to refer to a pair of abstract classes. Maybe now you're starting to see where this is going. The final product created by the Abstract Factory will be a concrete class that inherits from one of these abstract classes.
3. Two concrete factories are presented for each family of products. Our requirements have four families of products, but this diagram only shows two to keep things simple. You can add as many concrete factories as you need.
4. Two abstract classes are used to define two types of objects, independent of the family for the concrete object.
5. The final concrete products inherit from the abstract products. Phoebe and Kitty can model more specifics into their process because they are not just making bicycles – they are making the han-

dlebars too. Each physical factory should make the frames and handlebars that are needed.

The sisters head to the whiteboard and draw out the Abstract Factory pattern design. They will limit their diagramming to road bikes and mountain bikes to keep things simple:

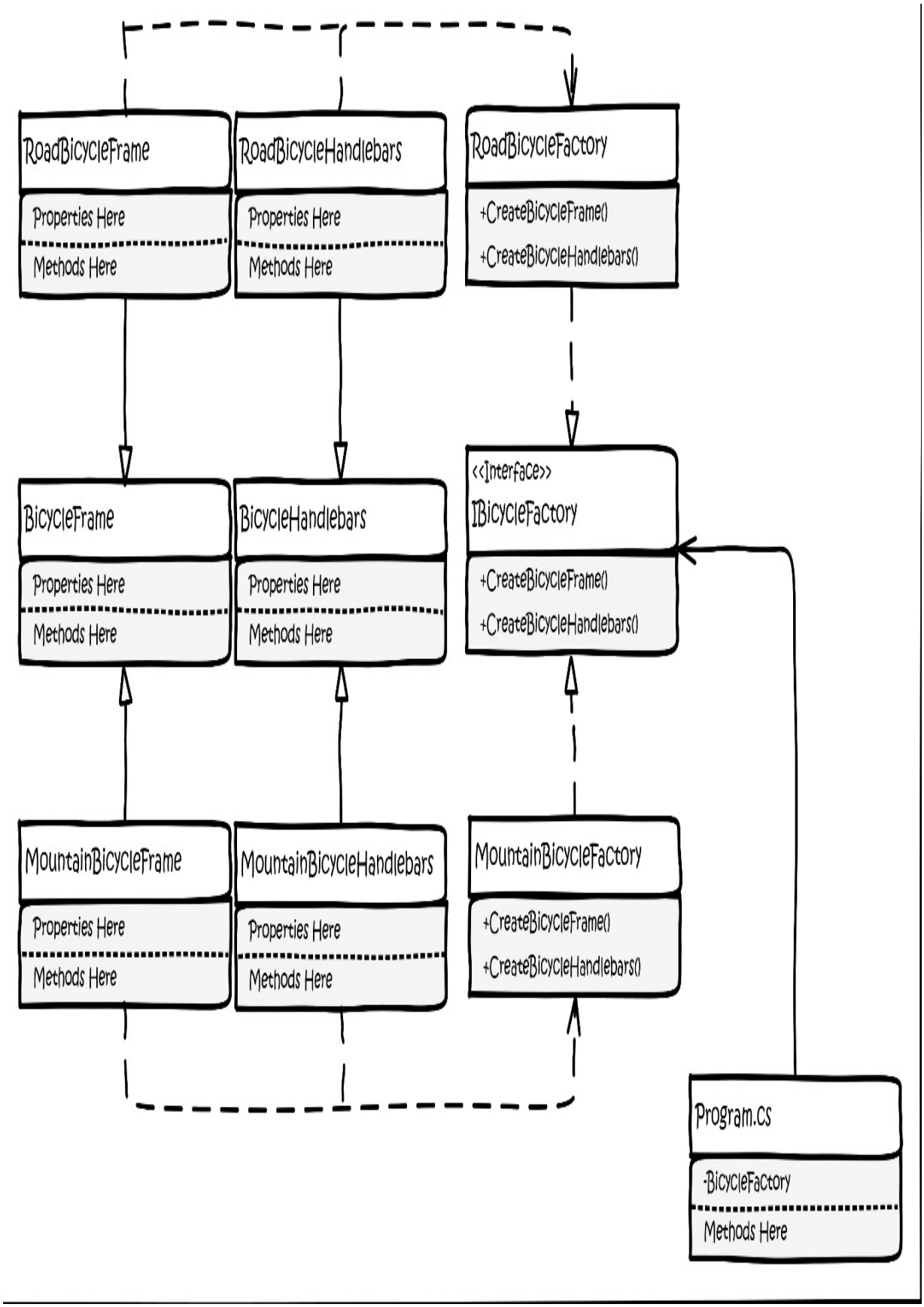

RoadBicycleFrame
Properties Here
Methods Here
RoadBicycleHandlebars
Properties Here
Methods Here
RoadBicycleFactory
+CreateBicycleFrame()
+CreateBicycleHandlebars()
BicycleFrame
Properties Here
Methods Here
BicycleHandlebars
Properties Here
Methods Here
<<Interface>>
IBicycleFactory
+CreateBicycleFrame()
+CreateBicycleHandlebars()
MountainBicycleFrame
Properties Here
Methods Here
MountainBicycleHandlebars
Properties Here
Methods Here
MountainBicycleFactory
+CreateBicycleFrame()
+CreateBicycleHandlebars()
Program.cs
-BicycleFactory
Methods Here

Figure 3.13 – Phoebe and Kitty's whiteboard design using the Abstract Factory pattern.

The client in this program is just the **`Program.cs`** file created by your IDE when you created a command-line project.

The diagram says **`Program.cs`** and depends on an object created from **`IBicycleFactory`**. Note that the arrowheads are different. These are significant in UML. The closed arrowhead on a solid line indicates inheritance. A closed arrowhead on a dashed line indicates a realization of an interface. The open arrowhead on a closed line indicates an association. The client depends on something that follows **`IBicycleFactory`**. The **`Program.cs`** class has a private field to hold an instance of an object that realizes this interface, which can be either **`RoadBicycleFactory`** or **`MountainBicycleFactory`**, as per the preceding diagram.

It is worth noting here that Phoebe drew **`IBicyleFactory`** as a literal interface. The code-level understanding of an interface may be either a literal interface or an abstract class because either can serve to define the structure an object must take.

We have concrete classes that realize the **`IBicycleFactory`** interface called **`RoadBicycleFactory`** and **`MountainBicycleFactory`**, respectively. Each concrete factory is responsible for creating a family of ob-

jects. In our case, the families are **`Road`** and **`Mountain`**. **`RoadBicycleFactory`** can create **`RoadBicycleFrame`** and **`RoadBicycleHandlebars`** based on the dependency line, but you can see that **`RoadBicycleFrame`** and **`RoadBicycleHandlebars`** inherit from the **`BicycleFrame`** and **`BicycleHandlebars`** abstract classes respectively.

When the client requests a bicycle frame and a set of handlebars, it can reference the abstract classes. Owing to Liskov substitution, the client doesn't need to be strictly coupled to any concrete class. This makes our client very flexible. As our bicycle family line-up changes, we won't need to alter the client because the client does not know what the factory is sending back. The client only knows it has methods called **`CreateBicycleFrame`** and **`CreateBicycleHandlebars`**.

Phoebe makes a branch on the repository. She knows you should always work in a branch and goes to work creating the code. She begins with the **`IBicycleFactory`** interface.

```
using BumbleBikesLibrary.BicycleComponents.Bicycl
using BumbleBikesLibrary.BicycleComponents.Handle

namespace BicycleAbstractFactoryExample;

public interface IBicycleFactory
```

```
{
  public IFrame CreateBicycleFrame();
  public IHandlebars CreateBicycleHandleBars();
}
```

Note the first two **`using`** statements. Since we are starting to break the bicycle into components, Kitty and Phoebe decided to refactor those components into a **`BicycleComponents`** namespace within **`BumbleBikesLibrary`**. The refactor isn't particularly relevant to patterns – it's just a refactor to get a little bit more organized. You can find the components in BumbleBikesLibrary in the GitHub repository for this chapter.

The sisters are coding by the book. Everything is typed into an interface for flexibility. We're adding methods to handle creating frames and handlebars, two related classes specified by the interfaces. Naturally, we could make more. There are more component classes in the **`BicycleComponents`** namespace. We're keeping it to two here to keep things simple. If you'd like to practice, see whether you can add the other components, such as seats, drivetrain, and brakes, to the pattern code.

Next, Kitty and Phoebe need to work in the concrete factories. These classes will only be used in this project, so you'll find them in the **`BicycleAbstractFactoryExample`** project in the book's sample source on GitHub. Remember, an *abstraction* can refer to an interface or an

abstract class. In this case, the abstract part of the *abstract factory* is this interface. Kitty takes on writing **`MountainBicycleFactory`** based on the **`IBicycleFactory`** interface:

```
using BumbleBikesLibrary.BicycleComponents.Bicycl
using BumbleBikesLibrary.BicycleComponents.Handle

namespace BicycleAbstractFactoryExample;

public class MountainBicycleFactory : IBicycleFac
{
  public IFrame CreateBicycleFrame()
  {
    return new MountainBikeFrame();
  }

  public IHandlebars CreateBicycleHandleBars()
  {
    return new MountainBikeHandlebars();
  }
}
```

The concrete factory is responsible for creating the specific objects we need relative to the object family we're creating. In this case, Kitty is making the parts for a mountain bike, so all the parts being re-

turned are specific to that family. More product families can be added without modifying the abstract factory.

Here is what Phoebe wrote for the **`RoadBicycleFactory`** class, which also extends **`IBicycleFactory`**:

```
using BumbleBikesLibrary.BicycleComponents.Bicycl
using BumbleBikesLibrary.BicycleComponents.Handle
namespace BicycleAbstractFactoryExample;
public class RoadBicycleFactory : IBicycleFactory
{
  public IFrame CreateBicycleFrame()
  {
    return new RoadBikeFrame();
  }

  public IHandlebars CreateBicycleHandleBars()
  {
    return new RoadBikeHandlebars();


  }
}
```

We now have an Abstract Factory pattern in place making two possible components, in two possible product families. Let's look at the

client. Phoebe writes that part:

```
using BicycleAbstractFactoryExample;

Console.WriteLine("Let's make some bicycles!");

IBicycleFactory roadBikeFactory = new RoadBicycle

var frame = roadBikeFactory.CreateBicycleFrame();
var handlebars = roadBikeFactory.CreateBicycleHan

Console.WriteLine("We just made a road bike!");
Console.WriteLine(frame.ToString());
Console.WriteLine(handlebars.ToString());
```

We made a road bike using **`RoadBikeFactory`**! The key here is the use of the interface as the factory type. Coding this way makes it possible to change the factory without relying on concrete factory classes directly. You can see the console output showing the results of the operation. Phoebe continues and writes some code to generate a mountain bike using **`MountainBikeFactory`**:

```
IBicycleFactory mountainBikeFactory = new Mountai
frame = mountainBikeFactory.CreateBicycleFrame();
handlebars = mountainBikeFactory.CreateBicycleHan
```

```
  Console.WriteLine("We just made a mountain bike!"
  Console.WriteLine(frame.ToString());
  Console.WriteLine(handlebars.ToString());
```

The same process could be repeated for any new bicycle type that Phoebe and Kitty could ever dream of making.

The Builder pattern

Phoebe finishes her implementation of the Abstract Factory and goes back to designing more on the robotics for the bicycle factory. How many times have you thought some job or project was simple, only to find out that once you got into the thick of working on it, things involved more than you realized?

Phoebe and Kitty are new to engineering, design, and software development. Phoebe's unique understanding of the design problems surrounding building an automated factory solidifies over time. She realizes that building a bicycle is more complicated than she had first realized. The sisters built their prototypes by hand. They were able to use wood for the frames and off-the-shelf parts for everything else. They are now committed to making their own frames and handlebars using a lightweight aluminum alloy.

Phoebe realizes the frame is the hard part. The other components, such as the wheels, brakes, and drivetrain, as well as the handlebars, could easily be built in-house as part of the automated process.

Naturally, this increases the complexity of the machinery and the software that runs it. Phoebe calls Kitty.

"*Hey sis, how is the Abstract Factory pattern treating you?*" Kitty asks.

"*It's fine, but I've been thinking,*" Phoebe replies.

"*Uh oh. Every time you do that Daddy's credit card gets a workout. What is your idea?*" Kitty asks.

Phoebe told Kitty about her idea to make all the parts from the same aluminum alloy. "*Some of the parts will need to be reinforced, but the result will be a lighter bicycle that costs less to build than we initially thought,*" said Phoebe. Kitty loved the idea. The sisters were excited before, but now they are fired up. Phoebe says, "*I'll get Dad's credit card and source the aluminum I need for the first few bicycles.*" Kitty replies "*Great. While you're doing that, I'll incorporate your ideas into the control software. You merged your branch into main, right?*"

Kitty pulls the latest code from *GitHub* and starts a new branch. She starts thinking hard about how to accomplish coding a process that will build an entire bicycle per any possible specification a customer

might have. The sisters have a set of four bicycles with constrained options for the first release. Kitty doesn't want to lock her thinking and her software design into building just those four bicycles. That would make it a stovepipe system. Kitty heads to the whiteboard. She realizes that she might have to disregard the progress she's made so far in the Abstract Factory pattern. A different pattern might not simply be a direct evolution of what they've done so far. She knows the elemental structures will remain the same. A bicycle will always need a frame, seat, handlebars, wheels, brakes, and a drivetrain.

The new problem entails creating a complex object. The construction will require multiple steps instead of a simple **`BuildBicycle`** method that creates the frame and handlebars, as she had in the earlier designs.

Deep in thought, her reverie is broken by a loud buzzing from her phone. It's sitting on her metal worktable and the vibration, indicating she's received a text, nearly bounced a small pile of screws and some tools right off the edge of the table:

Figure 3.15 – She checks the screen. There's a text from her father.

Uh - oh. Kitty decides to rip off the proverbial band-aid and calls her father. Maybe her excitement over the latest developments will cushion the blow. "*Daddy will understand,*" she says quietly to herself as if she is quietly trying to convince herself out loud.

Kitty and Phoebe's father was a software engineer. He'd lost his job a few years ago because he was always talking about crazy conspiracy theories on social media. He decided to just retire, and after a falling out with his extended family, Kitty and Phoebe's parents moved to a small town in Southern Oklahoma, just north of the Texas border where the roads are unpaved and the internet access is non-existent. This is just the way he likes it. Kitty dials the number.

When he answered, her father's language was, shall we say, colorful. This wasn't the first time Phoebe had run up a bill. She had once charged over a thousand dollars in 1 day. Most teenage girls run up a bill traveling or shopping. Phoebe's bill was from several reputable industrial suppliers, a tool and die shop, and a peanut butter factory. She never came clean on what she was up to, but shortly after that, she got into engineering school and the incident was all but forgotten.

Kitty talks him down from his anger and tells him about what's going on. "*You should look at* **the Builder pattern**," he says. "*Yeah, I once wrote a book on patterns and I remember this one. It is used to make complex objects using a flexible set of steps. It sounds like that's what you need. Oh, and y'all are gonna pay me back! I want stock in your company and the first bicycle you two build.*"

"*Of course, Daddy,*" Kitty said in her sweetest voice, normally reserved for when she wanted ice cream but had already been told, "*No.*" She decided not to admonish him for not telling her he had written a book on patterns. Her father had taught the sisters to code when Phoebe was 11 and Kitty was 12. He had published a lot of books and videos over the years. Of course, he had done one on patterns. At that moment, Kitty's call went to static. It was hard to hear anything. Her father was still speaking to her, but she only caught a few words. Something about alternate timelines and temporal recursion. "*He says the silliest things sometimes. He's probably been watching old episodes of Doctor Who again,*" she thought. The call dropped.

Kitty shrugged and set to researching the Builder pattern. Her father's book had been out of print for years, but it was a runaway international bestseller, so she was able to find a few of the diagrams among the many more recent books that cited his:

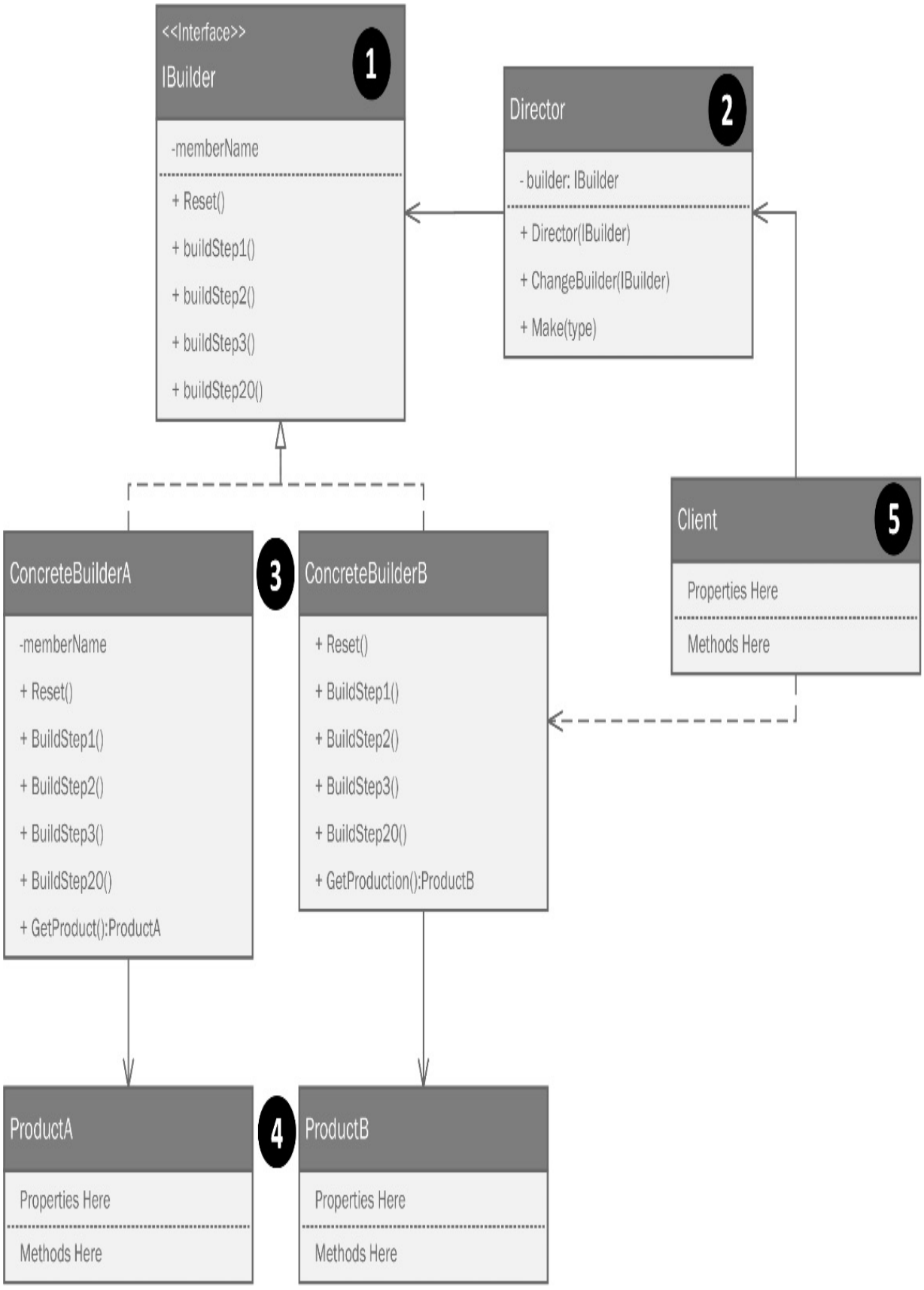

<<Interface>>
IBuilder
1
-memberName
+ Reset()
+ buildStep1()
+ buildStep2()
+ buildStep3()
+ buildStep20()
Director
2
- builder: IBuilder
+ Director(IBuilder)
+ ChangeBuilder(IBuilder)
+ Make(type)
Client
5
Properties Here
Methods Here
ConcreteBuilderA
3
-memberName
+ Reset()
+ BuildStep1()
+ BuildStep2()
+ BuildStep3()
+ BuildStep20()
+ GetProduct():ProductA
ConcreteBuilderB
+ Reset()
+ BuildStep1()
+ BuildStep2()
+ BuildStep3()
+ BuildStep20()
+ GetProduction():ProductB
ProductA
4
Properties Here
Methods Here
ProductB
Properties Here
Methods Here

Figure 3.16 – The builder pattern consists of a builder interface, a director to control the creation process, and concrete builders based on the builder interface to produce specific products.

Let's look at this diagram in more detail:

1. There are two important parts to the Builder pattern. The first is the IB. Remember, this can be a literal interface or an abstract class. I'll stick to using a true interface for the sake of flexibility. The **`Builder`** interface defines all the methods that will be in a set of concrete builder classes. The second part that you'll always find in the Builder pattern is the **`Director`** class.
2. A **`Director class`** is created, which contains logic that defines the creation process in a step-by-step fashion.
3. A set of concrete builder classes defines each type of object that can be created.
4. Different products come out of the concrete builder classes, depending on the logic contained within the director.

Armed with this new knowledge, Kitty goes back to her whiteboard:

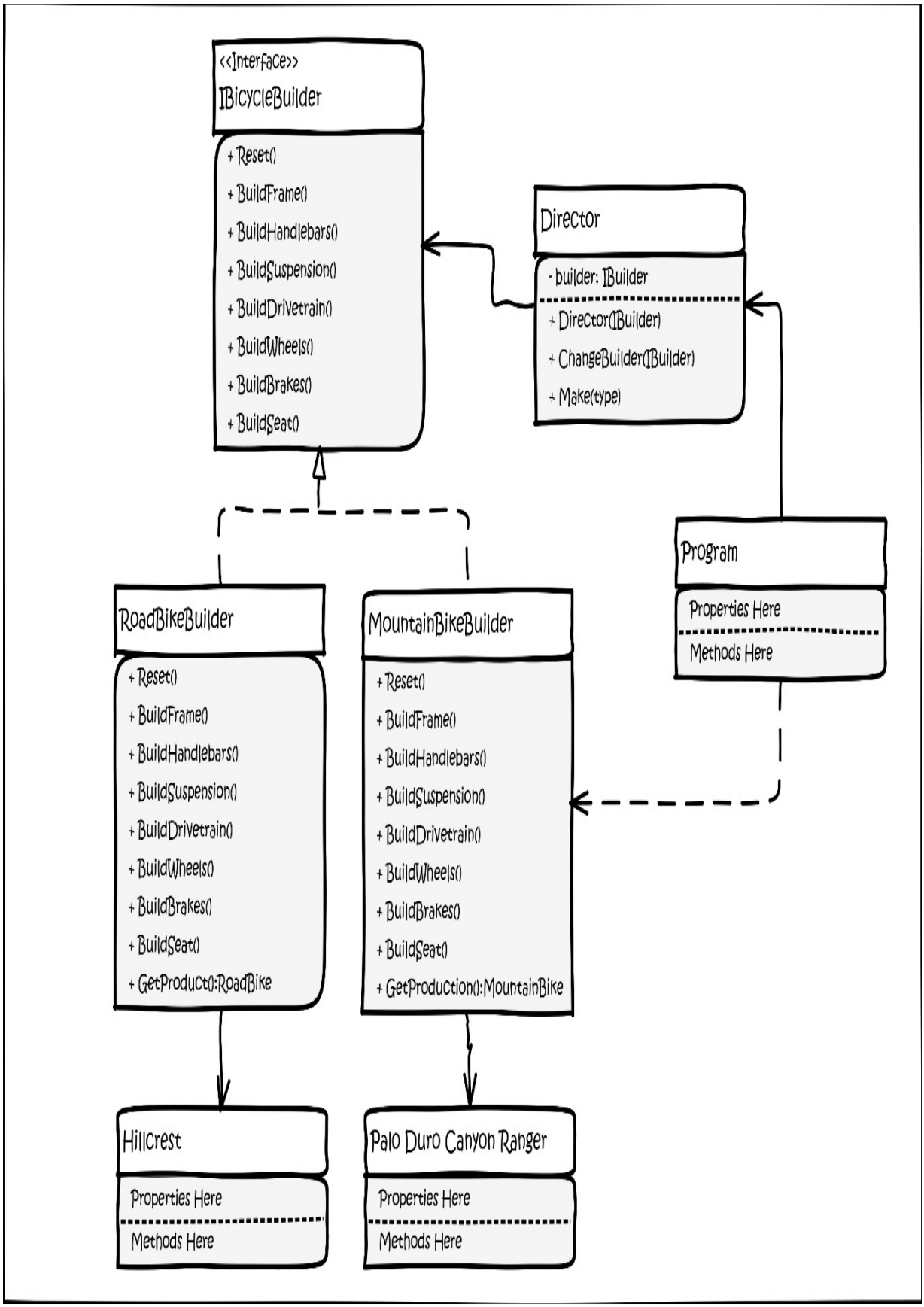

<<Interface>>
IBicycleBuilder
+ Reset()
+ BuildFrame()
+ BuildHandlebars()
+ BuildSuspension()
+ BuildDrivetrain()
+ BuildWheels()
+ BuildBrakes()
+ BuildSeat()
Director
- builder: IBuilder
+ Director(IBuilder)
+ ChangeBuilder(IBuilder)
+ Make(type)
Program
Properties Here
Methods Here
RoadBikeBuilder
+ Reset()
+ BuildFrame()
+ BuildHandlebars()
+ BuildSuspension()
+ BuildDrivetrain()
+ BuildWheels()
+ BuildBrakes()
+ BuildSeat()
+ GetProduct():RoadBike
MountainBikeBuilder
+ Reset()
+ BuildFrame()
+ BuildHandlebars()
+ BuildSuspension()
+ BuildDrivetrain()
+ BuildWheels()
+ BuildBrakes()
+ BuildSeat()
+ GetProduction():MountainBike
Hillcrest
Properties Here
Methods Here
Palo Duro Canyon Ranger
Properties Here
Methods Here

Figure 3.17 – Kitty's whiteboard implementation of the builder pattern.

Kitty was excited about this one. Having gone through a few different patterns, this one felt like it was the best representation of what the sisters are trying to accomplish. You can find this code in the **`BicycleBuilderExample`** project within this chapter's GitHub repository.

Kitty starts by creating an abstraction for what the builders will be producing – that is, the product. She creates an interface called **`IBicycleProduct`**:

```
public interface IBicycleProduct
{
  public IFrame Frame { get; set; }
  public ISuspension Suspension { get; set; }
  public IHandlebars Handlebars { get; set; }
  public IDrivetrain Drivetrain { get; set; }
  public ISeat Seat { get; set; }
  public IBrakes Brakes { get; set; }
}
```

The interface contains everything needed to make a complete bicycle. We have something of a design luxury at play here. All the bicy-

cles follow the same interface. We no longer need to think about a road bike or a mountain bike. A road bike is simply a collection of parts:

- A road bike frame
- A hardtail suspension (that is, no suspension at all)
- Road bike handlebars (that is, dropped curved handlebars)
- A road bike drive train (that is, a normal length chain and a group set with 3 front and 8 rear cogs for a total of 24 speeds)
- Caliper brakes
- A standard, cheap, very uncomfortable seat

Phoebe and Kitty learned about seats during their internship. Bicycle manufacturers know that seats are a very personal choice and no two riders will pick the same seat if they are given a choice. They all sell bicycles with cheap, uncomfortable seats and offer separate products to upgrade the seat to the rider's preference. This lowers the build cost for the bicycle and gives their dealers something to upcharge, as well as offering seat installation as a service.

Kitty codes a generic bicycle object for the builder called **`BicycleProduct`**:

```
public class BicycleProduct : IBicycleProduct
{
```

```
    public IFrame Frame { get; set; }
    public ISuspension Suspension { get; set; }
    public IHandlebars Handlebars { get; set; }

    public IDrivetrain Drivetrain { get; set; }
    public ISeat Seat { get; set; }
    public IBrakes Brakes { get; set; }

    public override string ToString()
    {
      var fullDescription = new StringBuilder("Here
      fullDescription.AppendLine(Frame.ToString());
      fullDescription.AppendLine(Suspension.ToStrin
      fullDescription.AppendLine(Handlebars.ToStrin
      fullDescription.AppendLine(Drivetrain.ToStrin
      fullDescription.AppendLine(Seat.ToString());
      fullDescription.AppendLine(Brakes.ToString())

      return fullDescription.ToString();
    }
  }
```

This class is just an implementation of the interface, plus a large **`ToString()`** override that we're using as the meat of our sample code.

Next, Kitty creates the **`IBicycleBuilder`** interface, which will define the various builders for the bicycle lines:

```
namespace BicycleBuilderExample;

public interface IBicycleBuilder
{
  public void Reset();
  public void BuildFrame();
  public void BuildHandleBars();

  public void BuildSeat();
  public void BuildSuspension();
  public void BuildDriveTrain();
  public void BuildBrakes();

  public IBicycleProduct GetProduct();
}
```

The interface defines what must be in a **`builder`** class. The next piece of the puzzle is the **`Director`** class. The classical implementation of the **`Builder`** pattern will always have one of these. In our case, we could skip this part because all of our bicycles fit the same interface. A director can create anything, given a builder. The director's job is to call the methods in the builder according to any business logic required and to return the product created by the builder. In our case, every product has the same set of properties, and the builders all have the same methods. Just realize that if your builders don't all follow the same interface, it's okay to have a lot more logic in your di-

rector to figure out what to do with it. The point of the director is to run the build methods in the builder in the right order. Sometimes, that entails a lot more logic than what we have here:

```
public class Director
{
  public Director(IBicycleBuilder builder)
  {
    Builder = builder;
  }
  private IBicycleBuilder Builder { get; set; }
```

Kitty starts with a private field to hold a reference to the builder she'll be working with. You can pass in anything that extends the **`IBicycleBuilder`** interface. The constructor sets the actual builder object that is passed in. Next, she makes a method that allows us to change the builder without exposing it directly:

```
  public void ChangeBuilder(IBicycleBuilder build
  {
    Builder = builder;
  }
```

Lastly, she creates a **`Make`** method per her UML diagram. It's the **`Make`** method's job to run the build steps in the right order. Kitty has this method follow the same process the robots will use. They'd start with a frame and then add parts to it. Nobody would think to start building a bicycle around the seat or the gears. Here, we can see a logical build process for a bicycle that starts with the bigger and more important parts, followed by smaller parts that attach to the larger ones:

```
    public IBicycleProduct Make()
    {
      Builder.BuildFrame();
      Builder.BuildHandleBars();
      Builder.BuildSeat();
      Builder.BuildSuspension();
      Builder.BuildDriveTrain();
      Builder.BuildBrakes();

      return Builder.GetProduct();
    }
  }
```

Next, Kitty creates the concrete builders for mountain and road bikes. Naturally, she created the code for the other bicycle types, but here, we'll keep things short so that you don't have to sort through so much code to see the pattern. Here's the **`RoadBikeBuilder`** class based on the **`IBicycleBuilder`** class:

```
public class RoadBikeBuilder : IBicycleBuilder
{
  private BicycleProduct _bicycle;

  public RoadBikeBuilder()
  {
    Reset();
  }

  public void Reset()
  {
    _bicycle = new BicycleProduct();
  }
```

Kitty creates a private field called **`_bicycle`** to hold the product to be built by the **`Director`** class. We have a public constructor and a **`Reset()`** method that sets the **`_bicycle`** field to a new bicycle based on the **`BicycleProduct`** class:

```
  public void BuildFrame()
  {
    _bicycle.Frame = new RoadBikeFrame();
  }
  public void BuildHandleBars()
  {
```

```
      _bicycle.Handlebars = new RoadBikeHandlebars(
    }
    public void BuildSeat()
    {
      _bicycle.Seat = new GenericSeat();
    }
    public void BuildSuspension()
    {
      _bicycle.Suspension = new HardTailSuspension(
    }
    public void BuildDriveTrain()
    {
      _bicycle.Drivetrain = new RoadDrivetrain();
    }
    public void BuildBrakes()
    {
      _bicycle.Brakes = new CaliperBrakes();
    }
    public IBicycleProduct GetProduct()
    {
      return _bicycle;
    }
  }
```

The same builders can be made for mountain, recumbent, and cruiser bikes. They look the same, but naturally, they use different parts as appropriate. The builder steps (methods) understand the parts need-

ed to build, while the director understands the order needed for the steps (methods) to be called.

The client code that's used to call these builders would look like this:

```
using BicycleBuilderExample;

Console.WriteLine("Let's make some bikes with the
```

Here's where all the hard work pays off. You can build any bicycle with three lines of code. First, make the builder. Then, make a director, if you don't have one yet, and pass in the builder you just created. Then, call the **`Make()`** method. The result is a perfectly built product:

```
var roadBikeBuilder = new RoadBikeBuilder();
var director = new Director(roadBikeBuilder);

var roadBike = director.Make();
Console.WriteLine(roadBike.ToString());
```

Would you rather have a mountain bike? That's easy!

```
var mountainBikeBuilder = new MountainBikeBuilder
director.ChangeBuilder(mountainBikeBuilder);
```

```
var mountainBike = director.Make();
Console.WriteLine(mountainBike.ToString());
```

We can make recumbents and cruisers just as easily, but in the interest of saving trees, I'll leave that to your imagination. If you're like me and have a weak imagination, the sample code for this chapter contains the builder for creating all four types.

Kitty is delighted with the way things turned out. She has a software architecture that is closed for modification but open for extension. She has leveraged interfaces and abstract classes wisely, so Kitty is ready for the next challenge the world of robotic bicycle manufacturing might have for her.

The Object Pool pattern

Back at the factory, Phoebe has made progress on the robotics. She has developed a mobile robotic arm to handle the welding during the manufacturing process. She was hoping to build 30 of these arms to allow for maximum factory output, but her father's credit card mysteriously stopped working. This puzzled Phoebe. Could her sister have told on her, leading her father to block the card from further purchases? "*She wouldn't do that!*" Phoebe thought, but as soon as she said

it, she knew that's exactly what had happened. She considered calling her sister out on it but decided to focus on her father's words:

"*A good engineer is someone who makes the best product possible given constraints of time, materials, people, and budget with a minimum of complaining.*"

The last part is the hardest. Most engineers complain, "*If I'd gotten the budget, I'd asked for…*" or "*If only I had another year…*" It's an ego thing. Phoebe swallows hers and decides she must figure out how to make do with just the 10 robot arms she was able to build. Her arm design looks a little like this:

Figure 3.18 – Phoebe's design for a robotic arm to handle the welding on the bicycle frames.

Given that the arms are mobile, Phoebe decides it is possible to have the arms move between welding projects as needed. As the bicycles come through the line, only a few will need welding at any given point in time. Phoebe thought, "*Running with 30 arms would have been nice, but honestly, they'd be dormant most of the time. This will be better. I'm forced to be more efficient.*"

The pool of 10 arms sit on a track where they move to any of the assembly lines at a moment's notice. As a weld is needed, Kitty's software will move the arm to where it's required. After the weld is finished, the arm can return to the pool and wait until it's needed again. If several welders are needed on different projects, one arm will come out of the pool and do its work, then return. So long as we don't need more than 10 arms at a time, we're fine. If we do, the eleventh job will have to wait until one of the arms is available. Phoebe is happy with this because creating the arms is expensive, and as she's discovered, she only needs a few. She can add more arms to the pool as the business expands.

Phoebe has unwittingly stumbled onto the **Object Pool pattern**. This pattern is used to create computationally expensive objects, and as such, they tend to slow software performance to a crawl. The most obvious case is working with relational databases. This is something every software developer does at some point, and for many of us, this is a skill we rely on daily.

Instantiating and connecting to a database is computationally costly and time-consuming. Sure, it takes maybe 100 milliseconds, but most relational databases are designed to handle millions of transactions every hour. When your software is under load at scale, 100 milliseconds is an eternity. Every commercially viable database has a driver that handles connection pooling for you. The driver creates a pool of open database connections. Just like with the robot arms, your software takes connections from the pool, runs queries, and then when the connection is closed (which should always be done as quickly as is feasible) the open connection, instead of being truly closed, is returned to the pool and is available for another process in your running program. Check out the following generic diagram:

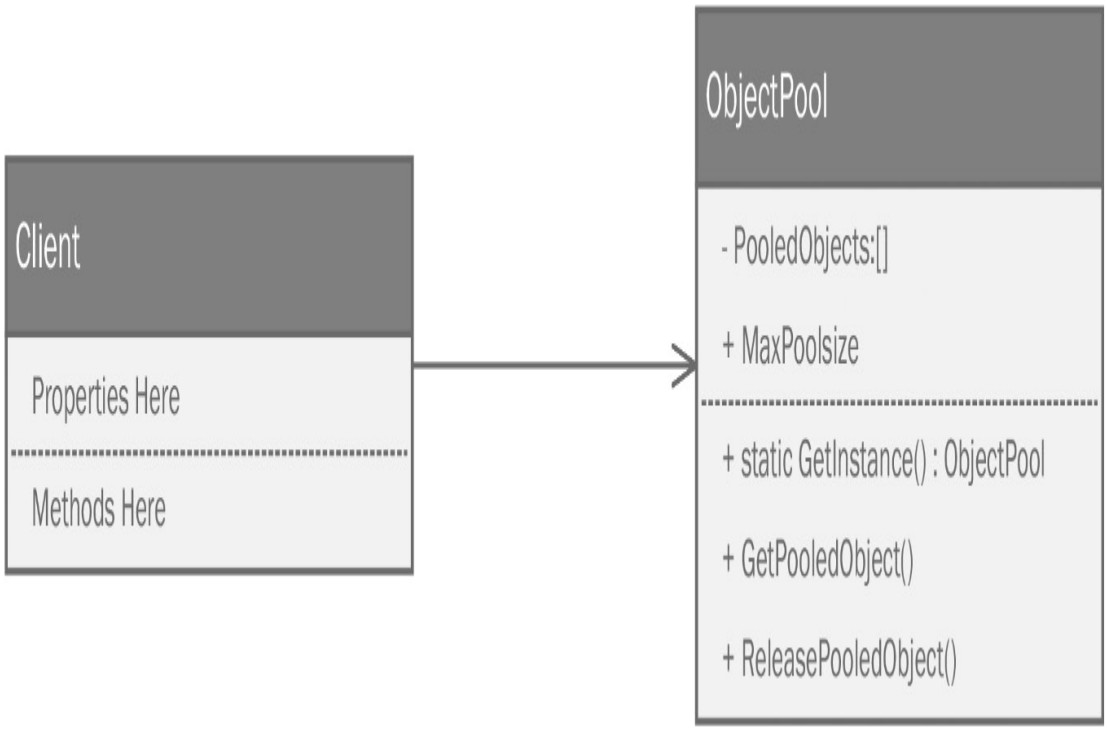


Figure 3.19 – The object pool pattern creates an object that manages a collection of other objects. The client requests objects from the pool to use them and then releases them when it's finished.

There is a private collection called **`PooledObjects`** that holds the objects in the pool. The client, after gaining access to the object pool, can request objects from the pool using the **`GetPooledObject`** method. Once it is finished, the borrowed object is returned to the pool. If the pool is ever empty, a subsequent request to **`GetInstance`** will create a new object if it can. If it can't, it is common for an implementation to wait until an object is returned to the pool.

Phoebe decides to add a pool to control access to her limited number of robot arms:

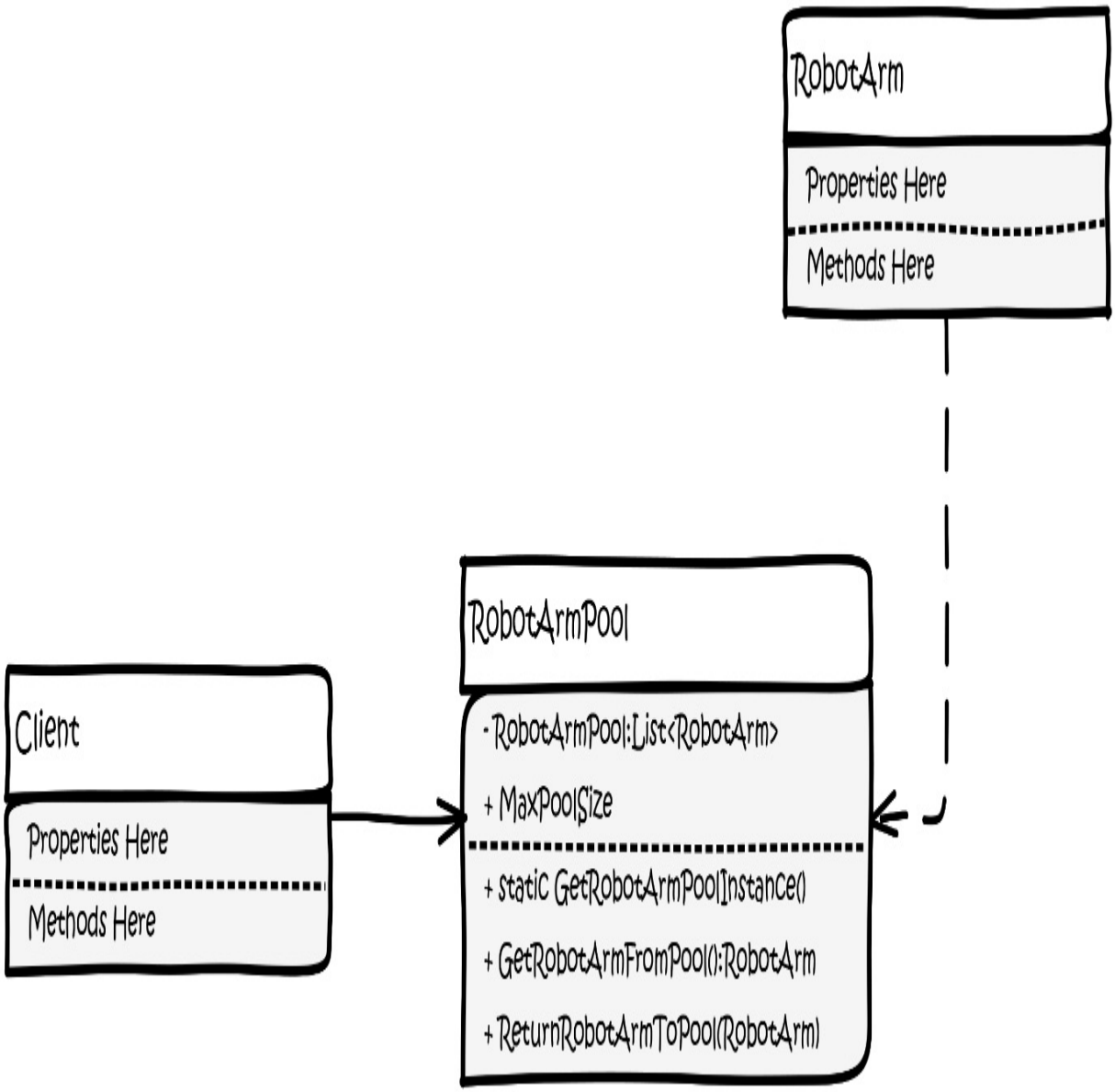


Figure 3.20 – Phoebe's design for an object pool to control a limited number of available robot arms on her assembly line.

Phoebe writes the code for the diagram:

```
public class WeldingArmPool
```

```
{
  private int _maxSize = 10;
```

Phoebe only has 10 robot arms available, but she knows she'll have more in the future. She makes a private variable here and initializes it to **`10`**. Next, she makes a constructor, but normally, a constructor would just handle creating the initialized object. The UML diagram contains an entry for a **`Reset()`** method, which does the same thing. She'll write that in a moment but puts it in place within the constructor. The IDE protests, but she knows it'll all work out when she has finished:

```
  public WeldingArmPool()
  {
    Reset();
  }
```

The most important part of the object pool is some sort of collection to hold the pooled objects. Phoebe opted for a **`List`** of **`WeldingArm`** objects. This will be initialized in the **`Reset()`** method we mentioned a moment ago, though Phoebe hasn't created it yet:

```
  private List<WeldingArm> Pool { get; set; }
```

Phoebe wants to be able to change the max size of the pool so that she's able to build more robot arms. She can increase the pool size or even decrease it if the arms are out of the pool for servicing:

```
public int MaxSize
{
  get => _maxSize;
  set
  {
    _maxSize = value;
    Reset();
  }
}
```

Phoebe decides it would be nice to have a way to see how many arms are in the pool:

```
public int ArmsAvailable => Pool.Count;
```

Finally, the fabled **`Reset()`** method we've been promised. It looks like code you'd put in a constructor. Phoebe wants a way to reset the pool if she needs to. Since being DRY is a good thing and calling a constructor with anything besides the **`new`** keyword seems weird and unnatural, Phoebe flips things around and puts the logic here and then calls it from wherever it's needed, including the constructor. This dri-

ves my IDE crazy. It thinks I have a non-nullable list that is never initialized. That's not true, but the automation in the IDE isn't savvy enough to see that. The code itself initializes the list and then fills it with as many arms as indicated by the **`MaxSize`** property:

```
    public void Reset()
    {
      Pool = new List<WeldingArm>();
      for (var i = 0; i < MaxSize; i++) Pool.Add(ne
    }
```

We need a way to get arms from the pool. The following method checks whether there are any arms available, and if there are, it retrieves the first one in the pool. If there aren't any arms, we throw an error:

```
public WeldingArm GetArmFromPool()
  {
    if (ArmsAvailable > 0)
    {
      var returnArm = Pool[0];
      Pool.RemoveAt(0);
      return returnArm;
    }
```

```
      throw new Exception("You are out of arms. Ret
    }
```

In a real program, you would probably just return a new object and pay the performance penalty. In this case, we are physically constrained. We could implement some concurrent code that watches for an available arm as it becomes free, which directs to a new job, but that's getting outside the scope of a pattern demonstration. Phoebe needs a way to return her arms to the pool. The arm stores the location of its last weld. Phoebe, to avoid confusion, decides to reset it to zero. This will indicate the arm isn't working on anything and that it's in the pool, ready for assignment:

```
    public void ReturnArmToPool(WeldingArm arm)
    {
      arm.CurrentPosition = 0; //not at any station
      Pool.Add(arm);
    }
  }
```

Wrapping things up, Phoebe writes a small test program in **`Program.cs`**:

```
  Console.WriteLine("Here's a program that controls
```

```
var armPool = new WeldingArmPool

{
  MaxSize = 10
};

var arm01 = armPool.GetArmFromPool();
arm01.MoveToStation(1);
if (arm01.DoWeld()) armPool.ReturnArmToPool(arm0
```

As a tool, the Object Pool can seriously speed up most software because usually, you only think to use it when an object takes a lot of time or resources to create. The pool creates the objects in advance, and hopefully, never again.

"*BUT!*" Phoebe says aloud to herself, "*this seems like it would work best if you could guarantee only one pool was in use at a time.*" She's right. Concerning the robot arms, it wouldn't do to have the software simply create more instances of the arm. It stands to reason that eventually, the control software should be multithreaded. You can't have multiple threads creating their own pool. Instantiating additional pools can't magically generate more resources in the real world. "*If only there were a way to make sure the object pool is only ever instantiated once per program run,*" Phoebe thinks. She'd keep working, but it's taco night, so she shuts down her laptop and prepares

herself for dinner. You can find Phoebe's source for the object pool pattern in this chapter's source code in the **`ObjectPoolExample`** project.

The Singleton pattern

That night, perhaps inspired by her Object Pool singularity problem, perhaps by the tacos, or perhaps both, Phoebe had a strange dream. She was a judge dressed in long flowing black robes sitting high on her bench in a courtroom. A trial was in progress. The defendant was one Sing Elton. He was a well-dressed middle-aged gentleman who sat quietly behind a large elaborately carved oaken table next to his counsel.

The courtroom clerk cleared her throat and spoke without inflection into a microphone. "*The defendant, Sing Elton, stands accused of impersonating a beneficial design pattern and is, in fact, an antipattern.*"

There was a collective gasp from half of the gallery. It came from the back of the courtroom, which Phoebe only now notices. The room is filled with software developers, all of whom are clad in cargo shorts, Birkenstock sandals, and $300 replicas of vintage *Metallica* T-shirts. Phoebe bangs her gavel and shouts, "*ORDER IN THE COURT!*" She had seen that in a movie once and had always wanted to do that. With the gallery quietened, the clerk continued without looking up

from her computer screen. "*These are very serious charges, Mr. Elton; how do you plead*?"

Elton's counsel, a wiry nervous man in an inexpensive blue suit stood and in a squeaky voice said, "*Not guilty!*"

The clerk entered the plea and the trial began. The prosecutor rose for his opening remarks, delivered in a slow Southern drawl reminiscent of a cowboy Western movie. "*Your honor, we intend to show beyond a reasonable doubt that Sing Elton is not a pattern as he represents himself but is an antipattern. He is openly a member of an outlaw coding organization called The Golden Hammers.*" The prosecutor sat down, flashing a wry smile with far too many teeth to be human. Do you know why sharks never attack attorneys? Professional courtesy.

The defense called Mr. Elton to the stand and as he questioned him, the defense was able to establish several salient premises aiming to prove Sing Elton was a pattern.

First, it's a widely used, popular pattern that was published in the *Gang of Four* book. This book is considered by many to be unassailable, given it is a seminal work in software design patterns.

"*OBJECTION!,*" the prosecutor yelled as he pounded his fist on the table. "*Your honor, neither popularity nor inclusion in a single book is*

sufficient evidence to prove this is a pattern."

Phoebe tapped her gavel lightly on the bench and said, "*Sustained. The defense will continue, but you're on thin ice, counselor. To be a pattern, Sing Elton must solve a common problem faced by many software developers. This isn't a popularity contest. You'll have to do better.*"

"*Yes, your honor.*" The counselor blushed and was flustered, as though he were hoping the case would be dismissed based on this opening argument. He continued to question Mr. Elton.

"*Can you tell the court exactly what problems you solve?*"

"*Sometimes,*" began Mr. Elton, "*you need to ensure there is only one copy of a class instantiated at any given time while your program is running. For example, you need access to a database, perhaps via an object pool, or maybe you need access to a file or a configuration service. In these cases, you should only have one instance to handle that. The instance is available to all parts of your program, giving you tight control over the global state. These are problems I can solve, and I do it in one place within a single class rather than them being scattered throughout your code.*"

At one point during the trial, the bailiffs brought in a big whiteboard, and Mr. Elton was asked to draw a diagram of himself. It looks as

follows:

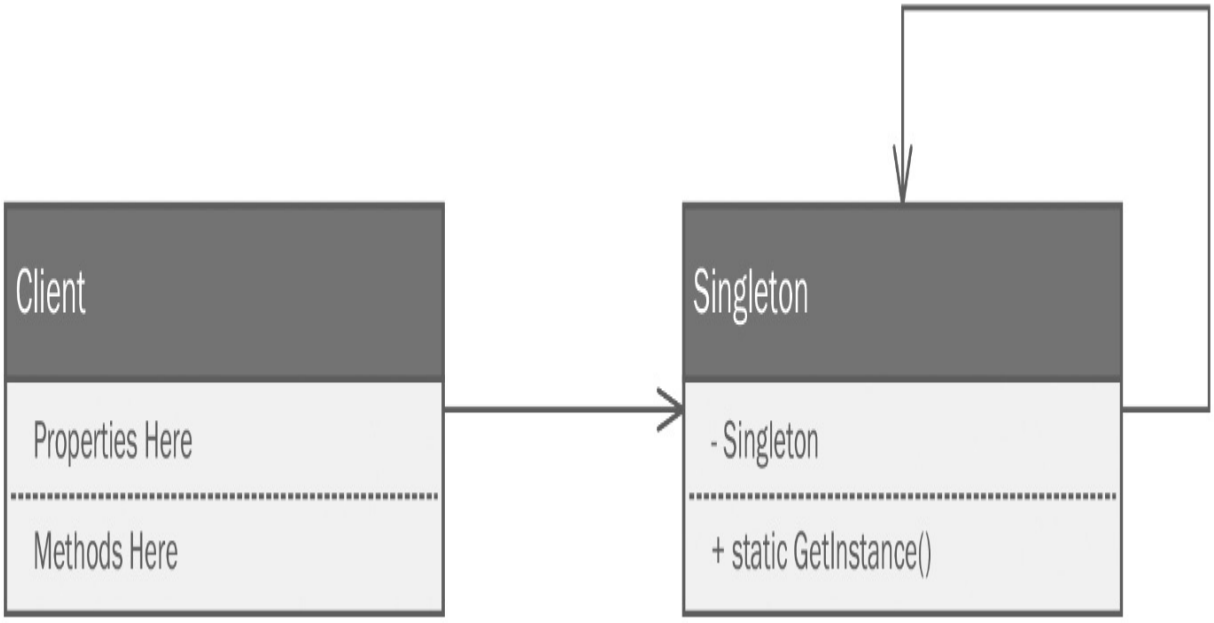


Figure 3.21 – The Singleton pattern uses a private constructor (not shown) to create an instance of the object (itself) held in a private field. Any calls to new for that class will check to see whether an instance already exists. If it does, the existing object is returned.

The defense continued and summarized its arguments this way:

- Shared resources used by your application, whether they be databases, files, remote services, or robotic arms, are granted and guaranteed a single point of access.
- The global state of the program is protected because there is only one point of access.

- Singletons are only initialized once, so any performance hit is taken only when the singleton is initialized.

The defense attorney concluded with "*The defense rests.*"

The nearly silent courtroom suddenly boomed with the echoing sound of a loud slow golf clap. The prosecutor stood up and straightened his tie while hungrily eyeing the jury. You could feel the confidence emanating from him. It was electric. As he spoke, it almost sounded as if his Southern drawl was intentionally more pronounced.

"*Your honor, members of the jury, let me tell you why everything the defense just said is a gigantic load of BAM!*" The *BAM* came from Judge Phoebe, who saw where this was going and slammed her gavel on the bench. She'd only been a judge for a few minutes, and might never be again, so there was no way she was going to let that kind of language into her courtroom. A girl's gotta have standards. Having admonished the prosecutor, to the bemusement of the clerk and defense attorney, the prosecutor called several witnesses who offered anecdotal evidence of Sing Elton being an antipattern and a Golden Hammer.

One software engineering manager testified that Sing Elton must be a Golden Hammer because in every job interview he had ever conducted, the applicants always claimed to have studied patterns.

When the manager asked for an example of a pattern, the only pattern anyone could name was the singleton pattern. There are quite a few classes out there (you'd think there would be only one example) where the implementation doesn't need to be a singleton. Everyone uses it because it's the one pattern they understand and remember.

One software developer employed by a major software company specializing in defenestration pointed out the most damning arguments:

- According to the widely accepted tenet, every class should have one responsibility; it should only solve one problem. Ironically, a singleton solves two. It ensures only one single instance of the class exists and it provides a single point of access to some shared or constrained resource.
- The Singleton pattern smells a lot like a global variable. Instead of just a variable, it's a whole class. Globals are universally vilified throughout coder-land because they are considered unsafe. Any method on any given object within your running program can modify the global state, which typically leads to unstable software. This has compounded ramifications for software designed for threading or concurrency.
- Singletons are cited as promoting tight coupling between classes. You really can't get around this since there is no such thing as an abstract singleton.

- Implementations of singletons are impossible to unit - test effectively. Besides our earlier stated problem of tight coupling, which is itself anathema to testing, you can't mock the singleton class. It's sealed and has no parent class, so using Liskov substitution is impossible as a tool for testing. Unit tests should be isolable and the effects of one test shouldn't affect other tests. With a singleton in the mix, they probably will.

The prosecutor concluded with, "*So you see, your honor, ladies and gentlemen of the jury, Sing Elton is nothing but a low-down dirty fraud. He's no pattern at all and should be stripped of the title.*"

A chorus of harumphs erupted from the gallery. Phoebe smiled because that was her cue to bang the gavel again. She could get used to this. She hit the bench so hard that the resulting sound shook her from her dream. It was like that dream everybody has had at least once – the one where you dream you're falling and you wake up just as you hit the ground.

"*I'm never eating tacos again!"* Phoebe exclaimed sleepily. She knew she was lying. She had a not-taco breakfast and went back to work. The details of her dream had some valid points, but she felt this was a case where she truly needed a singleton, given the very real constraints at play.

Phoebe models an object pool using a singleton to represent her constrained shared collection of robot arms in the control software:

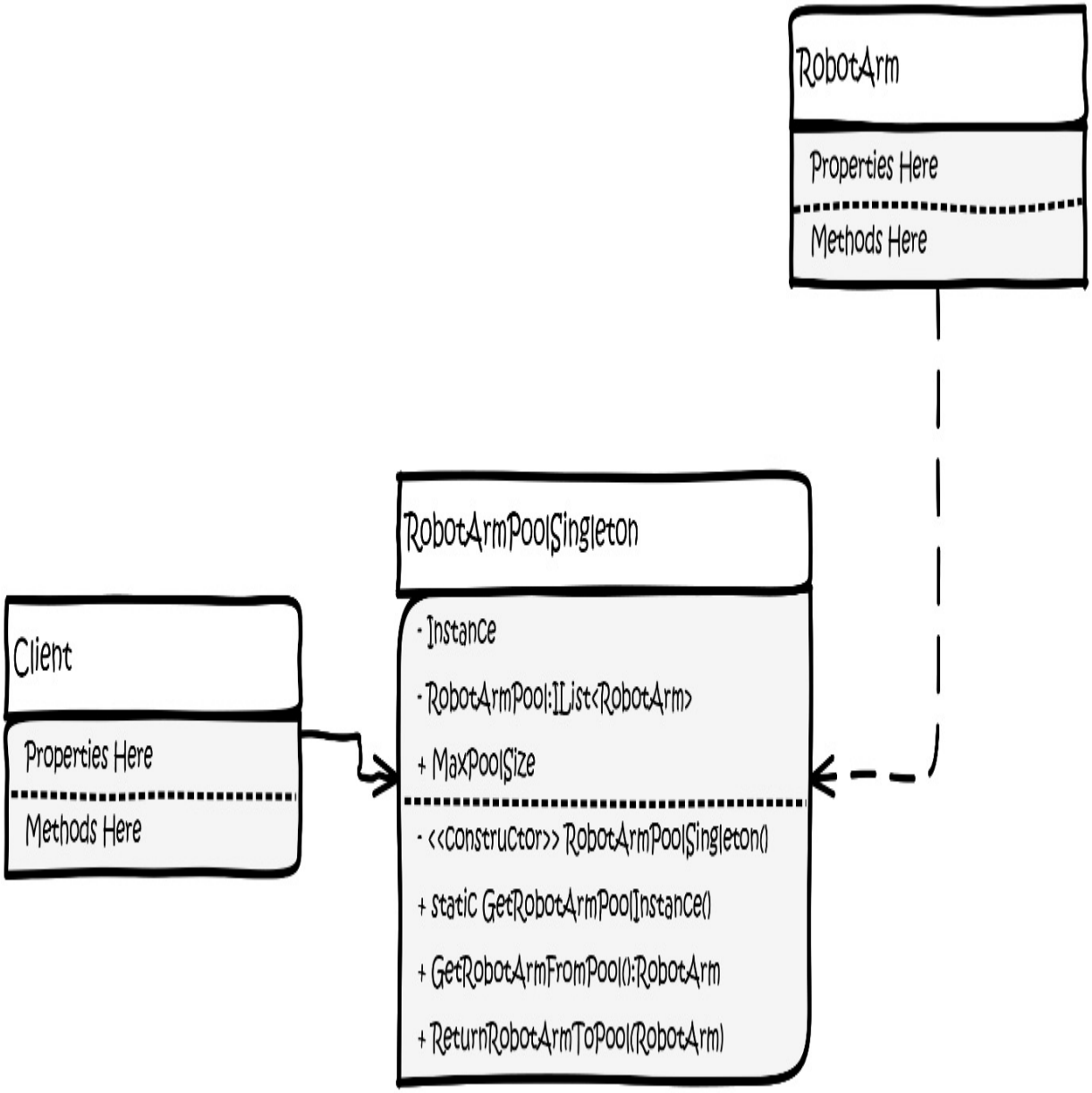


Figure 3.22 – Phoebe has refactored her object pool into a singleton. Now, only one robot arm pool can exist within her running program at any given time.

Phoebe moves to her IDE and starts refactoring the object pool for her robotic arms into a singleton. It's pretty short work. First, she renames the class and seals it so that it can't be extended. If the class were able to be extended, it would lose the protection afforded by the pattern. You can start to see why people hate the Singleton pattern. It's breaking a lot of the rules we've come to admire about keeping things open for extension:

```
public sealed class WeldingArmPoolSingleton
{
```

For a Singleton pattern to work, it needs a private static method to hold the singleton instance. In effect, this is a class that has a reference to itself, which is kinda odd. The key is that there can only be one of them, which is why we need the field to be static:

```
    private static WeldingArmPoolSingleton _instanc
    private int _maxSize = 10;
```

The next oddity you don't see often is a private constructor. There isn't a public one. Right about now is when your IDE, if it is equipped with static analysis tools, will start to complain. It will tell you there is no way to instantiate the object. That's good. That's what we're after. Phoebe keeps her `Reset()` logic in place. All she did was rename the

constructor so that it matches the class name and changed the access modifier to private:

```
private WeldingArmPoolSingleton()
{
  Reset();
}
```

The last piece to the Singleton is a static method to gain access to the instance property we started with. Phoebe uses C#'s property syntax to expose it. The first time the client program references the **`Instance`** property, the getter checks to see whether an instance already exists. If not, it creates one and sets the backing field's **`_instance`**. If **`_instance`** is not **`null`**, that means it's been called once already, so it just returns the instance that is already there. Since it's a static field, all references point to the same location in memory. Voila – you have a class that is impossible to instantiate twice:

```
public static WeldingArmPoolSingleton Instance
{
  get
  {
    if (_instance == null) _instance = new
    return _instance;
  }
```

```
    }
```

The rest of Phoebe's code remains unchanged. You can find the complete refactor in the **`SingletonExample`** project of this chapter's source code.

Summary

In this chapter, we presented one idiom, the simple factory, and four patterns – the Factory Method pattern, the Abstract Factory pattern, the Object Pool pattern, and the Singleton pattern.

All of these patterns are classified as Creational patterns. This means they govern the creation of objects by encapsulating the creation logic in a structure that is more flexible than using strictly concrete objects with the **`new`** keyword.

The Factory Method pattern is what most people think of when they hear "factory pattern." Using it entails abstracting creation logic into a factory class called a creator. The creator object is defined by an interface to maximize flexibility. We also create an interface for the objects the factory is producing. We call this the product. Each factory creator class is responsible for a subset of all the products in your program.

The Abstract Factory pattern involves creating families of objects that organically go together. Using it entails creating an abstract definition for multiple creator classes. Each creator is responsible for a concrete object.

The Builder pattern is used when you need to make objects using a complex set of steps. Using it is similar to the Abstract Factory pattern, but your builder classes are defined by an interface. Each method in the builder represents a step of the build process. It might be tempting to put a single method in each builder class to call the steps in order. However, this is usually delegated to a `Director` class. The builder houses the methods to build the object, but the `Director` class contains the logic behind the order in which those methods are called.

The Object pool Pattern is designed to help you manage objects that are either limited through a real constraint, such as our robot arms, or objects that are computationally expensive to create, such as a database, network service, or file connections. The idea is to pay the creation penalty once by creating a list of those objects that are kept available during your program's run. When one of these objects is needed, the object is taken from the pool and it's returned when it's no longer needed. This allows other processes to use it later, without having to go through normal instantiation.

The Object Pool pattern can be effectively combined with the Singleton pattern. The Singleton pattern is controversial and often considered an antipattern because it cannot be extended, and it promotes tight coupling. Phoebe was able to use it in combination with her robot arm pool to ensure she didn't accidentally create multiple robot arm pools. She only has 10 physical arms to work with, so duplicating pools could prove to be problematic.

These patterns were presented as progression. The sisters started with the simple factory and worked iteratively to the builder pattern. This wasn't intentional – that's just the way it worked out. You shouldn't take this progression as an indication that the Builder pattern is better than the Factory Method or Abstract Factory. Each of these patterns has its place and often, patterns can be paired together like a fine wine and a good steak.

In the next chapter, we'll look at structural patterns, which are patterns designed to refine the way you structure your class hierarchies for maximum flexibility and to fully achieve the open-closed principle.

Questions

Answer the following questions to test your knowledge of this chapter:

1. What is a programming idiom and how are they different from patterns?
2. What are some popular programming idioms besides `Hello, World`?
3. What is the downside to relying on the simple factory idiom?
4. Is the Singleton a pattern or an antipattern? Why?
5. What pattern should you use when you are dealing with creating a family of related objects?
6. What pattern should you use for objects that have a complex build process?
7. Which class in the Builder pattern is responsible for controlling the execution of the build steps in the proper order?

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Ritchie, D. M., Kernighan, B. W., & Lesk, M. E. (1988). *The C programming language*. Englewood Cliffs: Prentice Hall.
- https://sites.google.com/site/steveyegge2/singleton-considered-stupid

4

Fortify Your Code With Structural Patterns

Recently, my wife asked a question that should have been simple to answer: “What did you do for fun when you were a little boy, say 9 or 10 years old?” I had to think about it. When I was 9 or 10, computers in the home were not possible, unless you lived in a military bunker that had a steady high voltage continuous sine wave power feed. The bunker would also have needed several thousand square feet of raised flooring, industrial-grade air conditioning, and a steady supply of clean water to use for CPU cooling. This wasn’t a normal living environment for most of my friends when we were 9 or 10. The question was difficult because as soon as I turned 12, I got my first computer. It was the Radio Shack TRS-80, complete with a level 1 8-bit Z-80 processor, 4 K (as in 4,000 bytes – just bytes - not Kilo, Mega, nor Giga) of memory, a monochrome monitor with a resolution of 128 by 48 very blocky pixels, and a cassette tape deck for loading and storing programs and data. The new computer occupied my every waking moment and since then, I’ve devoted probably far more of my life to screen time than I would care to admit. However, the question was:

what did I do for fun before my computer? After a minute of thinking, I remembered that I built model rockets.

Just up the street from the house where I grew up was a hobby shop that sold model rocket kits along with engines and launchers. On Saturday mornings, I'd paw through mom's purse, scrape together 5 USD, and walk to the shop to grab a kit. In the beginning, they were simple "level 1" kits that you could assemble and launch in a few hours. As I grew more adept, the models became more complicated with elements such as parachutes that deployed when the rocket had spent its fuel and reached its apogee. One rocket even had a single-shot camera that would take an aerial picture on its way back to Earth. The fancy kits had rockets that resembled spaceships from the Star Wars movie (there was only one back then) and Battlestar Galactica.

The rockets generally had a similar structure. However, as they became more sophisticated, more instructions were required to assemble them correctly and launch them safely. This reminds me of our next collection of patterns.

Structural patterns are designed to help you assemble objects into larger, more complex structures, thus avoiding the stovepipe monolithic structure that we've come to abhor. Structural patterns are to larger systems what creation patterns are to individual object in-

stances. Structural patterns help us maintain flexibility and efficiency. There are quite a few documented structural patterns, but this chapter focuses on four of the most important patterns:

- **The Decorator pattern**
- **The Façade pattern**
- **The Composite pattern**
- **The Bridge pattern**

As in earlier chapters, these structural patterns will be demonstrated within the context of simple command-line programs. This will limit the amount of noise you would encounter with more complex, though probably more interesting, desktop, web, or gaming projects.

This chapter assumes you understand the basics of the **Unified Modeling Language** (**UML**). All patterns are diagrammed using UML class diagrams throughout the book. If UML is a new concept for you, check out *Appendix 2* of this book. You don't need to understand all of UML's 14 diagram types. I only use class diagrams because that is all we need for our pattern work.

Technical requirements

Throughout the book, I assume you know how to create new C# projects in your favorite **integrated development environment**

(**IDE**). I do not spend any time on the mechanics of setting up and running projects in this chapter. However, if you need a tutorial on IDEs or how to set up a project, check out *Appendix 1* of this book. Should you decide to code along with me, you'll need the following:

- A computer running the Windows operating system. I'm using Windows 10. Since the projects are simple command-line projects, I'm pretty sure everything here would also work on a Mac or Linux, but I haven't tested the projects on those operating systems.
- A supported IDE such as Visual Studio, JetBrains Rider, or Visual Studio Code with C# extensions. I'm using Rider 2021.3.3.
- Some version of the .NET SDK. Again, the projects are simple enough that our code shouldn't be reliant on any particular version. I am using the .NET Core 6 SDK.

If you'd like the code, you can find the completed project files for this chapter on GitHub at https://github.com/Kpackt/Real-World-Implementation-of-C-Design-Patterns/tree/main/chapter-4.

B2B (back to bicycles)

In our last episode, sisters Kitty and Phoebe had decided to open their own bicycle factory: Bumble Bikes. They intend to leverage Kitty's expertise and design the most innovative bicycles on the road. Phoebe is capitalizing on her own engineering skills by designing and

building robotics. Although neither sister is a professionally trained software developer, their father, a retired time-traveling software engineer, had taught the girls to code at a young age. The sisters know their way around an IDE, but they are only just learning about patterns. Therefore, the coding task at hand is to write the robotic control software that will run an automated factory.

The robotic manufacturing systems take instances of **`Bicycle`** classes and translate those into physical bicycles. The girls have mastered Creational patterns and they have settled on using the Builder pattern. The Builder pattern will be used to create any type of bicycle component needed and assemble those components into a finished bicycle.

The Decorator pattern

It was a hectic Monday morning for Kitty and Phoebe. Over the weekend, Kitty had ridden a prototype mountain bike around some of the trails near her home in the rocky desert of West Texas. She wanted a challenging test, so she chose Black Gap Road in Big Bend, a US national park. Big Bend derives its name from a large bend in the Rio Grande river, which forms the park's border, as well as the United States' southern border with Mexico. Black Gap Road is well known as a challenging trail. It has washouts, shallow creek crossings, and

an actual gap (after which the road is named). The gap consists of a narrow passage between two large hills formed from volcanic rock. In the middle of the gap is a ledge that drops about 3 feet (about 1 meter) into the next section. Kitty had driven over the ledge in her Jeep many times, but never on a bike. She misjudged the drop and wound up flat on her back. After catching her breath, she got back on her bicycle and completed the trail. Phoebe was waiting at the trailhead for Kitty to pick her up with the Jeep.

On Monday morning, their phones started ringing. Phoebe spent almost an hour on the phone speaking with a raw material supplier. Based on the amount of material the girls were projecting, the supplier informed them they would need to set up an account on the supplier's extranet, which is a private network available to larger customers. Bumble Bikes could get favorable pricing on raw materials. However, to take advantage of the pricing, Bumble Bikes had to commit to creating an interface between their robotic manufacturing system and the supplier's inventory control system. Phoebe saw this as a positive because it meant that in the short term, she could leverage the inventory control system of her supplier, rather than having to create this system herself. The only downside was that Phoebe would need to modify her software, in particular the `Bicycle` class, to provide notifications to the supplier's API. The `Bicycle` class had been internally released and was already in use. Cracking open the class to add the notifiers required by the supplier would force her to violate the open-

closed principle that states: you should never modify code that is in production by changing the class. Instead, she should find a way to extend the class.

Meanwhile, Kitty was on her phone talking to the president of a company that owns a large number of bicycle dealerships across the United States. The company wants to become the exclusive dealer for Bumble Bikes in the US. The sisters had envisioned selling directly to customers as well as forming agreements with small local bicycle shops. They were surprised a big chain of stores would be interested in their nascent product line. After a long discussion with the president, Kitty learned one of the requirements for doing business with the dealerships was that each bicycle ordered would need an owner's manual and would have to be printed with each dealership's details scattered throughout the manual. A large-scale printing system would be provided by the company that owned the dealerships, so the girls wouldn't have to come up with the capital for any equipment.

As fate would have it, the two hung up from their respective calls at the exact same time.

"You won't believe this!" Phoebe exclaimed.

"No, you won't believe this!" Kitty countered.

The two were excited as they discussed what had transpired. On Friday, they were a small bicycle start-up. The next Monday morning, they were positioned to be a serious competitor in the US bicycle market. Their dreams were coming true more quickly than they could have imagined. All they had to do was make a few modifications to their code.

"It's simple!" said Phoebe. *"We can just change the Bicycle class and add the new properties and methods to meet our new requirements."*

"Not so fast," replied Kitty, and then she continued, *"The Bicycle classes are already in production. Changing them conflicts with the open-closed principle. Besides, not every bicycle object needs the new manual printing behavior. If we force the behavior on every subclass, we're violating the interface segregation principle. We'd be violating two SOLID principles!"*

"Oh, yeah," Phoebe said dejectedly. The two did some research and came across an idea that seemed sound. If they could create a class that wrapped or decorated the `Bicycle` class, they could create an extended class with new behaviors without breaking any of their existing implementations.

The Decorator pattern allows you to add properties and methods to a class without touching the original class, while still honoring any con-

crete implementations. You can see a generic drawing for the Decorator pattern in *Figure 4.1*:

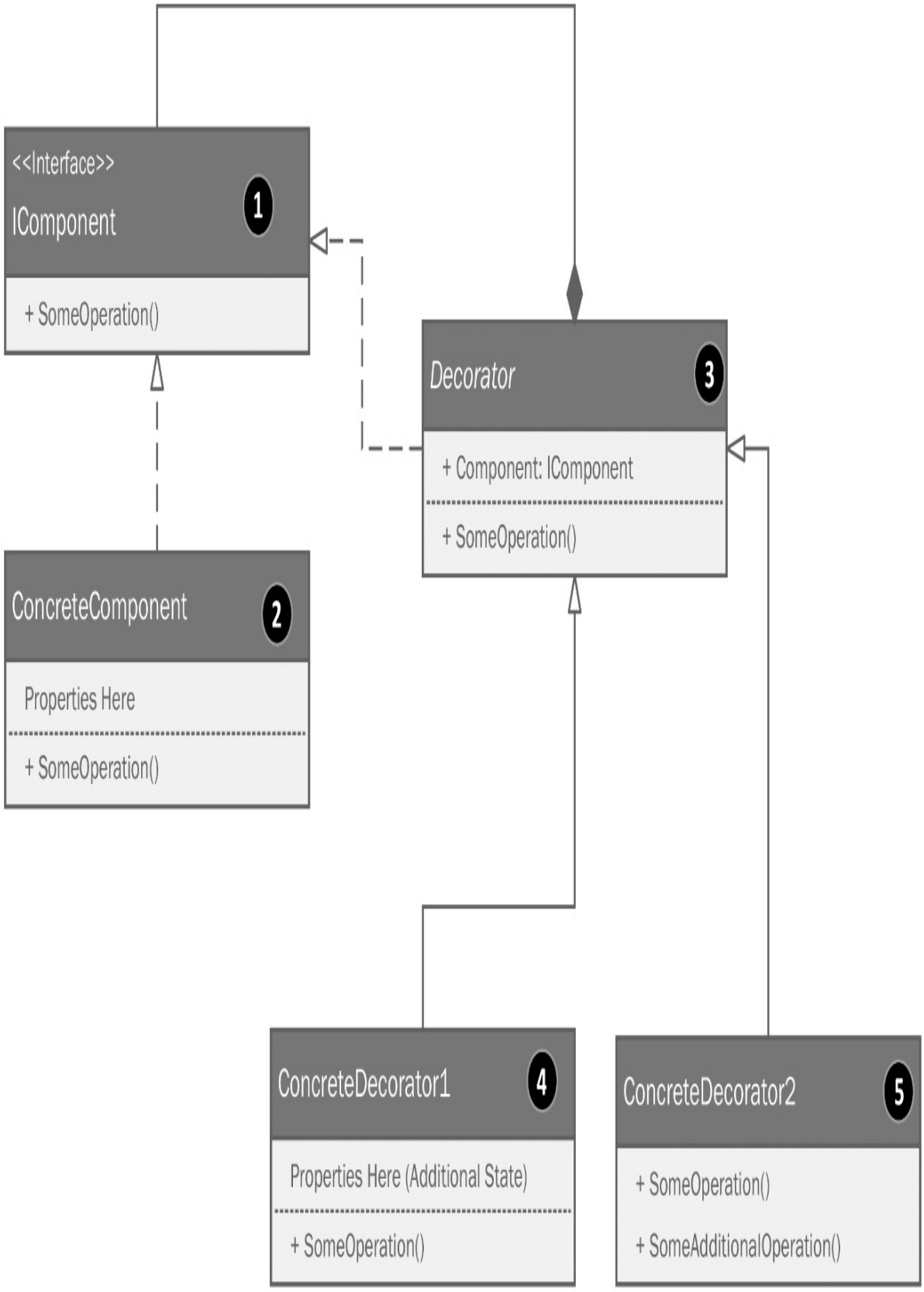

<<Interface>>
IComponent
1
+ SomeOperation()
Decorator
3
+ Component: IComponent
+ SomeOperation()
ConcreteComponent
2
Properties Here
+ SomeOperation()
ConcreteDecorator1
4
Properties Here (Additional State)
+ SomeOperation()
ConcreteDecorator2
5
+ SomeOperation()
+ SomeAdditionalOperation()

Figure 4.1: The Decorator pattern.

The parts of the pattern are categorized and explained here:

1. **`IComponent`** is the interface that defines the behavior we intend to wrap.
2. **`ConcreteComponent`** is the original implementation. In the example presented thus far, this would be the API from the microcontroller vendor.
3. **`Decorator`** is an abstract class that holds a reference to the concrete component via the interface.
4. **`ConcreteDecorator1`** is a concrete class that extends the decorator class but adds some additional state definitions in the form of properties or fields.
5. **`ConcreteDecorator2`** extends the decorator with additional operations.

Kitty and Phoebe needed to add two different behaviors to their **`Bicycle`** class. One behavior would interface with the raw material supplier's inventory control system. The other behavior would allow customized manuals to be printed at the factory and shipped with the bicycles to hundreds of dealerships throughout the US. Ideally, there should also be a way to add both behaviors to their **`Bicycle`** objects. In other words, they really wanted their decorators to stack. Stacking would allow them to potentially modify all their **`Bicycle`** classes to

work with the raw material supplier's system, but only deal with printing manuals for bikes sold through their dealership agreement. They should be able to add these behaviors or omit them as appropriate. Kitty headed to the whiteboard and ultimately settled on the structure shown in *Figure 4.2*:

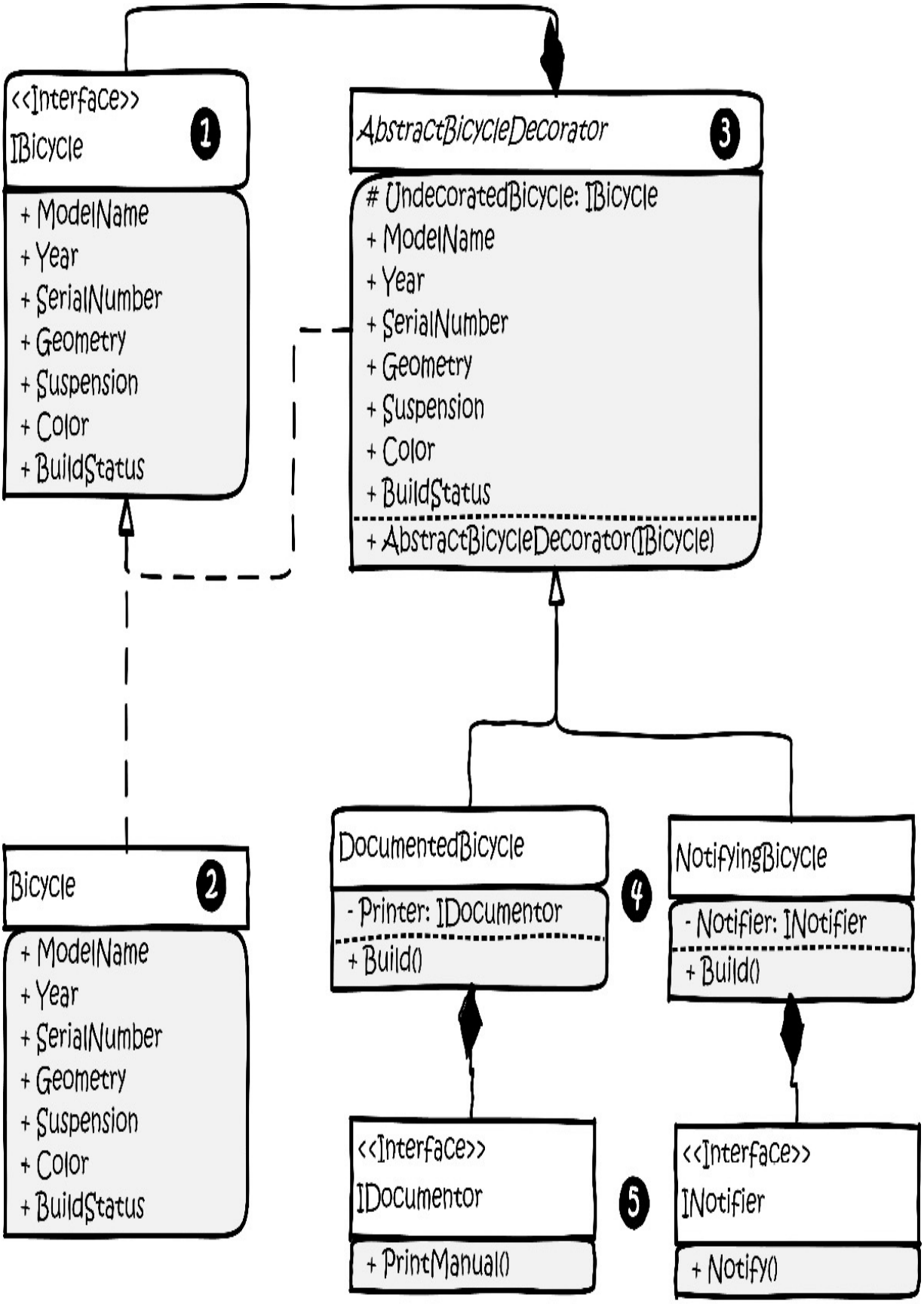

<<Interface>>
IBicycle
1
+ ModelName
+ Year
+ SerialNumber
+ Geometry
+ Suspension
+ Color
+ BuildStatus
AbstractBicycleDecorator
3
UndecoratedBicycle: IBicycle
+ ModelName
+ Year
+ SerialNumber
+ Geometry
+ Suspension
+ Color
+ BuildStatus
+ AbstractBicycleDecorator(IBicycle)
Bicycle
2
+ ModelName
+ Year
+ SerialNumber
+ Geometry
+ Suspension
+ Color
+ BuildStatus
DocumentedBicycle
- Printer: IDocumentor
+ Build()
4
NotifyingBicycle
- Notifier: INotifier
+ Build()
<<Interface>>
IDocumentor
+ PrintManual()
5
<<Interface>>
INotifier
+ Notify()

Figure 4.2: Kitty's implementation of the Decorator pattern.

Let's go over the classes in the diagram:

1. This is the interface we created in *Chapter 3, Getting Creative with Creational Patterns*, to represent our bicycles. Nothing has changed.
2. This is the abstract **`Bicycle`** class that implements the interface. Nothing has changed here either. In fact, neither the interface nor the abstract class has changed.
3. This is the abstract decorator class. Strictly speaking, you don't necessarily need the word *decorator* in the class name. It's here for the sake of clarity. Note that it uses composition to include a **`protected`** property, which contains a reference to a class that implements **`IBicycle`**. This object is set by the constructor while simultaneously implementing the **`IBicycle`** interface. At first glance, this doesn't seem very DRY. As you'll see, it will be. This will make sense when you see it in the code.
4. The Decorator classes. Here we have two: **`DocumentedBicycle`** and **`NotifyingBicycle`**. You can have as many as you need. You can stack them in your implementation, making it possible to have a **`Bicycle`** object with either a manual printer or a notifier, or both or neither. As Bumble Bikes expands and new business requirements are realized, we can potentially add and selectively stack more decorators without disturbing the original bicycle class itself.

5. The **`IDocumentor`** and **`INotifier`** interfaces define the decorating behaviors. Keeping them as interfaces prevents the decorator from being tightly coupled to concrete implementation.

Decorators are used to add new properties and methods (or if you prefer, behaviors) to objects without modifying the original class. This allows you to honor the open-closed principle by cleverly extending the class. In this case, we are extending the class by wrapping it, rather than merely extending using inheritance.

There are three steps involved in decorating a class:

1. Create a class with a **`private`** member containing the class you want to decorate. In our case, we'll be decorating the **`AbstractBicycle`** class. We need a class that contains a **`private`** property of the **`IBicycle`** type, and a constructor that allows us to set this property.
2. We need to implement all the properties and methods that are already in the **`IBicycle`** interface. When we implement the getter, setter, and regular methods for the decorator, we pass them through to the private instance. In effect, we've wrapped the class and the decorator performs exactly as in the original class.
3. We add the decorating properties and methods. If you intend to stack your decorators, it is important to have a common method that can link them together. We'll be using the **`Build()`** method.

Let's look at Kitty's code implementation. We'll work from the bottom up, beginning with the two interfaces. She adds the **`IDocumentor`** interface. This allows dealership customized manuals to be printed when the bicycle is built:

```
public interface IDocumentor
{
  public void PrintManual();
}
```

And then she adds the **`INotifier`** interface, which defines a function to communicate with the raw material supplier's inventory control system:

```
public interface INotifier
{
  public void Notify();
}
```

Next, let's look at the **`AbstractBicycleDecorator`** class:

```
public abstract class AbstractBicycleDecorator :
{
  protected readonly IBicycle UndecoratedBicycle;
```

The **`protected`** field to hold the original **`IBicyle`** object is crucial here. This is the original, undecorated object instance we'll be decorating:

```
protected AbstractBicycleDecorator(IBicycle bic
{
  UndecoratedBicycle = bicycle;
}
```

Next, we need to implement the **`IBicycle`** interface in the decorating class. We do this by passing everything through to the private undecorated object set by the constructor. Each call to the decorator class will be passed along to the undecorated instance:

```
public string ModelName
{
  get => UndecoratedBicycle.ModelName;
  set => UndecoratedBicycle.ModelName = value;
}
public int Year => UndecoratedBicycle.Year;
public string SerialNumber =>
  UndecoratedBicycle.SerialNumber;
public BicycleGeometries Geometry
{
  get => UndecoratedBicycle.Geometry;
  set => UndecoratedBicycle.Geometry = value;
```

```
    }
    public BicyclePaintColors Color
    {
      get => UndecoratedBicycle.Color;
      set => UndecoratedBicycle.Color = value;
    }
    public SuspensionTypes Suspension {
      get => UndecoratedBicycle.Suspension;
      set => UndecoratedBicycle.Suspension = value;
    }
    public ManufacturingStatus BuildStatus {
      get => UndecoratedBicycle.BuildStatus;

      set => UndecoratedBicycle.BuildStatus = value
    }
```

Finally, we'll implement the **`Build()`** function as abstract. In our original **`Bicycle`** class, which was also abstract, we implemented this method within the class. This is important because we're actually making changes to the way the bicycles are built. As you'll see, this method provides the means to stack our decorators:

```
  public abstract void Build();
}
```

Let's finally move into the two decorator classes. Kitty's requirement to print manuals comes first because she's the elder sister. At least, that's what she's always arguing. Here, the **`DocumentedBicycle`** class is extending **`AbstractBicycleDecorator`**:

```
public class DocumentedBicycle : AbstractBicycleD
{
```

There's a **`private`** field to hold an **`IDocumentor`** object:

```
  private IDocumentor _documentor;
  public DocumentedBicycle(IBicycle bicycle, Manu
                           printer) : base(bicycl
  {
    _documentor = printer;
  }
```

Here's the actual decoration. The **`Build()`** method that exists on any object passed into the constructor is called. Then, the additional decorating behavior – in this case calling the **`PrintManual()`** method on the decoration – is called:

```
  public override void Build()
  {
    UndecoratedBicycle.Build();
    _documentor.PrintManual();
  }
}
```

Phoebe's requirement to notify the supplier's inventory control system is implemented in its own decorator class:

```
public class NotifyingBicycle : AbstractBicycleDe
{
  private readonly INotifier _notifier;
  public NotifyingBicycle(IBicycle bicycle, INoti
                          notifier) : base(bicycle
  {
    _notifier = notifier;
  }
  public override void Build()
  {
    UndecoratedBicycle.Build();
    _notifier.Notify();
  }
}
```

The structure is identical, except, of course, for the decoration. This time, we pass in an object that meets the **`INotifier`** interface. We call the appropriate method when the **`Build()`** method of the original un-decorated object is called.

Now, we need some concrete classes to satisfy the **`INotifier`** and **`IDocumentor`** interfaces. We'll keep these simple. The **`IDocumentor`** interface is realized with a class called **`ManualPrinter`**:

```
public class ManualPrinter : IDocumentor
{
  public void PrintManual()
  {
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine("The manual is printing!");
    Console.ResetColor();

  }
}
```

The **`PrintManual()`** method is our added behavior. All this does for this example is print the line **`The manual is printing!`** Since there is a lot of text flying around in our console, I elected to make the decorator's output cyan-colored to make it easy to spot. An implementation of the **`INotifier`** interface might look as follows:

```
public class MaterialsInventoryNotifier : INotifi
{
  public void Notify()
  {
    Console.ForegroundColor = ConsoleColor.Yellow
    Console.WriteLine("The materials inventory co
      system has been notified regarding the manu
        this bicycle.");
    Console.ResetColor();
  }
}
```

Again, I added a splash of color to the output to make it easier to spot. This time, it is yellow. It has been a while since I've punned, so let's wrap this up with some code that makes use of our decorator classes!

These are the contents of the **`Program.cs`** file in the sample project files:

```
var regularRoadBike = new RoadBike();  //no decor
regularRoadBike.Build();
Console.WriteLine("++++++++++++++++++++++++++++++++
```

This is a normal, undecorated **RoadBike** object. BORING! We've seen that one before! Let's decorate it with the ability to print a custom manual:

```
var bikeManualPrinter = new ManualPrinter();
var documentedBike = new DocumentedBicycle(new Ro
                                           bikeMan
documentedBike.Build();
```

This time, we instantiated **DocumentedBicycle**, which is our decorator. It needs an undecorated **RoadBike** and a **ManualPrinter**. When we call the build method, the **DocumentedBicycle** class calls the **Build()** method on the **RoadBike** class. Then, it calls its own **Build()** method that adds the new manual-printing behavior.

If you didn't pick up on it, I'm adding some separators, so when we run the example, it will be easy to see each part of the run:

```
Console.WriteLine("++++++++++++++++++++++++++++++
```

That was a blast! Let's do another one! This time, let's try out the **NotifyingBicycle** decorator. It works the same way. First, we make an instance of **MaterialsInventoryNotifier**, which embodies the new behavior the decorator is adding to **RoadBike**:

```
var manufacturingInventoryNotifier = new Material
    Notifier();
```

Next, we instantiate the **`NotifyingBicycle`** class, passing in a new **`RoadBike`** for decoration, along with **`manufacturingInventoryNotifier`**:

```
var notifierBike = new NotifyingBicycle(new RoadB
    manufacturingInventoryNotifier);
```

Now, we call the decorator's **`Build()`** method:

```
notifierBike.Build();
Console.WriteLine("++++++++++++++++++++++++++++++++
```

Remember how this works: the decorator (**`NotifyingBicycle`**) has a **`Build()`** method. So does the **`RoadBike`** class it decorates. The decorator calls the **`Build()`** method on **`RoadBike`**, which produces a **`RoadBike`** object. Then, the decorator calls its own **`Build()`** method, which adds the notification behavior.

The cool thing about decorators is that they stack. For the pièce de résistance, we'll put both decorators on the **`RoadBike`** object at the

same time:

```
var notifyingDocumentedBike = new NotifyingBicycl
    DocumentedBicycle(new RoadBike(), bikeManualI
        manufacturingInventoryNotifier);
notifyingDocumentedBike.Build();
```

Yikes! That's hard to read. Let's break it apart. In the middle, you'll find a new undecorated **`RoadBike`** class:

```
var notifyingDocumentedBike = new NotifyingBicycl
    DocumentedBicycle(new RoadBike(), bikeManualI
        manufacturingInventoryNotifier);
notifyingDocumentedBike.Build();
```

Now, move out from there and you'll find where we make **`DocumentedBicycle`** using this new **`RoadBike`**:

```
var notifyingDocumentedBike = new NotifyingBicycl
    DocumentedBicycle(new RoadBike(), bikeManualI
        manufacturingInventoryNotifier);
notifyingDocumentedBike.Build();
```

We passed in **bikeManualPrinter**. In real life, be careful of re-using your instances in this way. We used **bikeManualPrinter** before and now we're passing it into a second bike. These are passed by reference, meaning if you change any property on **bikeManualPrinter**, it will affect both the value of **documentedBike** from the earlier example and this **notifyingDocumentedBike** we're building now.

Now, let's move all the way to the outer constructor:

```
var notifyingDocumentedBike = new NotifyingBicycl
    DocumentedBicycle(new RoadBike(), bikeManualP
        manufacturingInventoryNotifier);
notifyingDocumentedBike.Build();
```

The outermost constructor creates an instance of **NotifyingBicycle** using the new **DocumentedBicycle**, which uses a new **RoadBike**. At each level, we pass in a decorating behavior.

To put this another way, **notifyingDocumentedBike** is created using a new **DocumentedBicycle**, plus **ManualPrinter**. **DocumentedBicycle** is created using a new **RoadBike**.

You can see the output from the run program. *Figure 4.3* shows a run of the **DocumentedBicycle** decorator, followed by the **NotifyingBicycle** decorator. The final run shows both decorators on

the same object running successfully. The code indicates color coding for the output, but this book is not printed in color. Or maybe it is and you need an eye exam:

Figure 4.3: The output from the run of Program.cs for the DecoratorExample project (to save space, the output from the undecorated RoadBike is not shown).

You can see why the Decorator pattern is a crowd favorite among developers. You can use it when you need to add behaviors to an object without breaking existing implementations. Decorators can be used to create layers of business logic that can be stacked or combined as needed. For example, only bikes sold through the dealership network will need the **`DocumentedBicycle`** decorator, but all of them that use raw materials from our supplier will need the **`NotifyingBicycle`** decorator. If we make a bicycle with raw materials from another source that won't be sold through the dealer network, then we do not need any of the decorators.

You can also use a decorator to extend classes that are awkward or impossible to extend through regular inheritance. Consider a class that is sealed, meaning it can't be extended through inheritance. You can still extend it using a decorator! This can result in you feeling as though you're an outlaw. This is normal and may lead you to obtaining a black cowboy hat and learning the lyrics to every song written by Johnny Cash. You've been warned.

The Façade pattern

"Ugh!" Phoebe exclaims. It's 4 a.m. at Phoebe's workshop. Her formerly white lab coat is covered in grease and she's wading through aluminum shavings shed by an industrial lathe. She's trying to build

one of her robotic arms. She has several different designs, but this one is a heavy model that's bolted to the floor. There are three variations of the arm. One arm is equipped with a welder used to weld the aluminum alloy bicycle frames together, another one has a buffer used to perfect the finish on the bicycles after they are painted, and the last one is equipped with a gripper used to hold the bicycle during assembly. Phoebe wanted to build several arms of each type. She was constrained by her budget and could only afford to build 10 arms. She decided the best combination for the arms would be three welders, three buffers, and four grippers. After whiteboarding the process, she realized her factory would not be able to keep up with the demand generated by the sisters' stellar marketing campaigns. The Kickstarter alone had already generated hundreds of orders and the factory wasn't even operational yet! Phoebe was having trouble coming up with a way to make this factory work.

Her sister Kitty had been working on the control software. The two had agreed on how the software would work in advance of creating the robotics because sometimes making the software before the hardware is practical. The software design is fluid and easy to change. It can drive the hardware design, which is less fluid and more expensive to change. This is one of those cases. On a whim, Phoebe pulls Kitty's GitHub repo and switches to the branch where Kitty had designed the object structure for the control arms.

Phoebe finds Kitty's implementation of the Decorator pattern and it gives her an idea. She can think of the attachments for the robot arm as decorators! This way she can make 10 arms, but the arms can perform whatever action is needed by switching the decorator. To be clear, Phoebe isn't creating software. She's taking inspiration from the Decorator pattern and making a robotic arm with interchangeable attachments, thus allowing the basic arm to perform many functions as needed without building a new arm for each task.

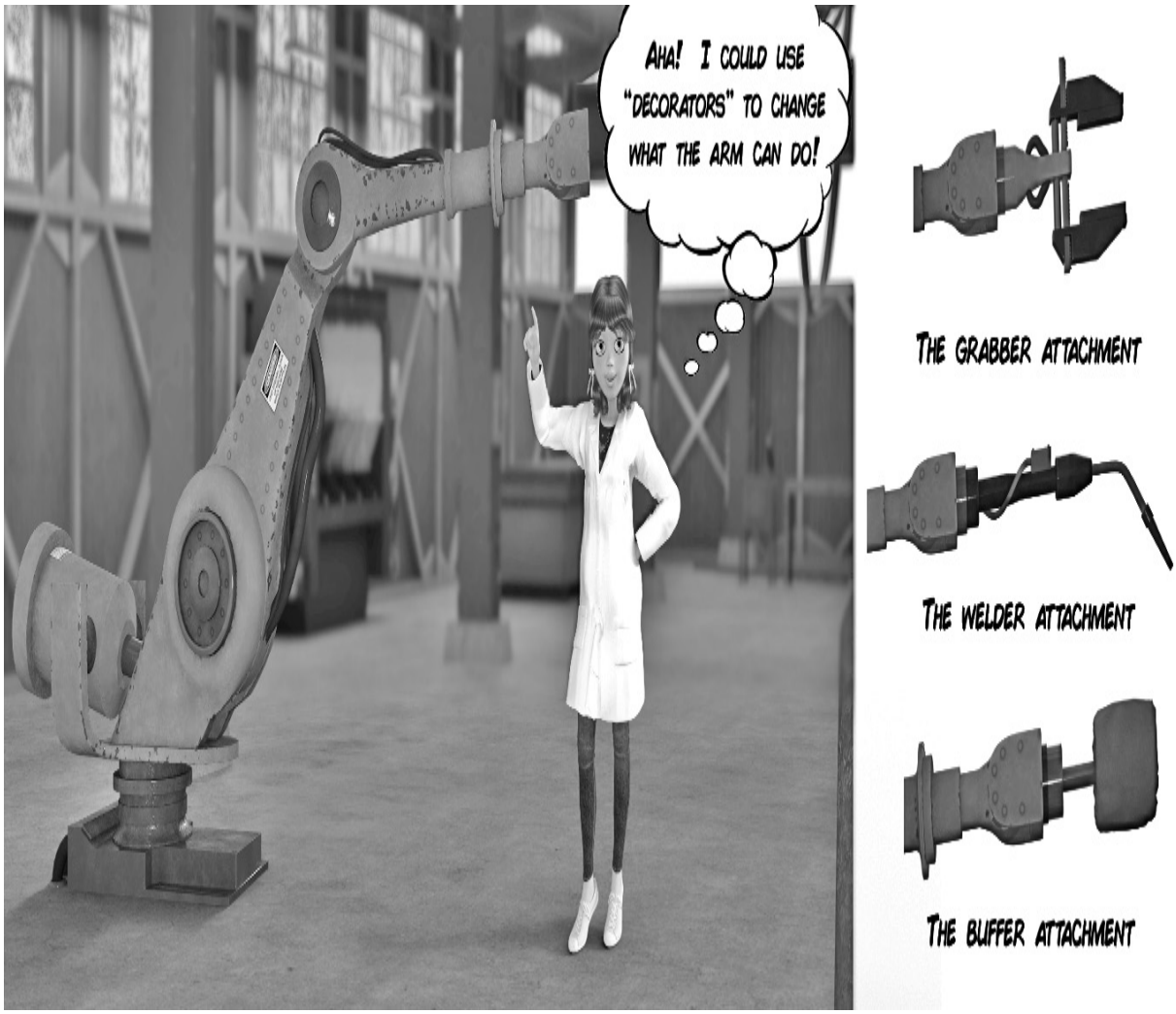


Figure 4.4: Phoebe finds inspiration in the Decorator pattern.

This is fantastic! However, there's another problem. Each attachment will be built from components from different suppliers and each will have a different API to control the arm's attachment.

A few weeks ago, Kitty left the s mall town of Alpine, Texas, where she was finishing up her Industrial Design degree at Sul Ross University. She drove her bright yellow Jeep Wrangler eight hours northeast to Dallas, Texas where her sister Phoebe was finishing her studies in engineering at Southern Methodist University. The purpose of Kitty's visit, aside from the superior collegiate nightlife in Dallas, was to research and source potential parts for their factory robotics. They already had access to some of the more basic needs for manufacturing bicycles. At one point during their father's career, he had spent a few years working for a helicopter factory in Fort Worth, Texas. Fort Worth is adjacent to Dallas and the residents refer to the area as the Dallas Fort-Worth metroplex. During their father's time at the helicopter factory, he made many contacts that could help the girls achieve their goals. Everything they needed from machine shops to advanced computer-controlled laser cutting and fabrication was but a phone call away. The electronics, however, were a different story. The girls had decided to develop the robotics using readily available **commercial off-the-shelf** (**COTS**) components.

Servomotors are electrical motors that allow for precise control of angular or linear positioning, along with velocity and acceleration. They are widely deployed in the development of robotic and human-controlled industrial machinery. The objective for Bumble Bikes is to have an automated factory. Kitty began looking into microcontrollers. Microcontrollers are tiny computers that allow **application programmer interface**-(**API**-) level interactions to control anything, including servomotors. The servomotors might be connected to the computer using pin connectors, sometimes called “hats”, that seat onto the microcontroller’s printed circuit board. The local computer store in Dallas had a robust section of their supermarket-sized facility dedicated to the purveyance of microcontrollers, servomotors, and all the related electronics needed.

One week and one large workbook of Libre Office Calc spreadsheets later, they had a working bill of materials for building the robot arms and their three different attachments. Assembling the parts into an electrically viable robotic arm is elementary. The hardest part is writing the software. The girls sourced three different microcontrollers with three different APIs for each arm attachment: one for the grabber, one for the welder, and another for the buffer.

A simple solution would be to write a stovepipe application that calls the APIs directly based on any required logic dictated by the needs of the manufacturing process. The sisters’ recent experience with facto-

ry patterns has trained them to be wary of such strategies. Quick and easy solutions are neither sustainable nor maintainable in the long term, and they are thinking long-term. They want to build their robots once and those robots, being well maintained, conceivably could run forever.

One of Kitty's professors, Professor Charles Dexter Ward, had taught a class titled *Introduction to Smart Product Design*. It was an entire semester on the exact problem the girls are solving: how to design efficient automated systems using sensors and microcontrollers. Dr. Ward had cautioned Kitty and her academic colleagues about vendor lock-in. Kitty and Phoebe are starting a business by turning a passion project into an enterprise they can pass to their children and grandchildren. It doesn't make sense to trust that the microcontrollers they buy now, and their related APIs, will remain unchanged for the long term or even a few years into the future. By tightly coupling to the current APIs directly, the sisters would be trusting that the API developers they use today will remain in business as long as Bumble Bikes. This would also assume that the APIs will continue to evolve and be maintained in accordance with their application's needs. Naturally, this would be a very naïve assumption.

A safer bet is that new microcontroller APIs will be introduced into the market and from different companies every few years. The method signatures and the way the API itself is invoked will be different than

they are today. Consider common technology for remote API calls 20 years ago. **Common Object Request Broker** Architecture (**CORBA**) was replaced by **Simple Object Access Protocol** (**SOAP**). SOAP, in turn, has been entirely displaced by **Representational State Transfer** (**REST**), common in web development and the **Internet-of-Things** (**IoT**) industry, which is still nascent in the year 2022. I'll wager many well-trained software developers reading this book have little to no exposure to CORBA or SOAP, just as your descendent colleagues will likely have something very different from REST. Any system that is tightly coupled to any API has a life span equal to the shortest life span of its tightly coupled components. Kitty has taken this particular lesson to heart. Within the software, Kitty had represented the control arm with an interface that implemented the robot arms performing abstract operations. These can be mapped to the API. She's using **the Façade pattern**.

Façade, in regular English, or in this case, French, means "face". In architecture (the kind with buildings, not software), it refers to the front of a building. A façade is usually ornate and represents what designers call *curb appeal*. One of the most famous façades I can think of is the castle at Walt Disney World in Florida, seen in *Figure 4.5.* A sidenote: Dear Internal Revenue Service, please accept my amended return with my deduction labeled "book research trip to Disney." It was purely business, I assure you.

Figure 4.5: The front façade of the castle at Walt Disney World in Florida presents an ornate appearance to the outside world.

In software architecture, the Façade pattern works in reverse. Instead of a fancy ornate face to an object or API, it's a simplified point of access. This nifty pattern can insulate your programs from vendor lock-in by situating itself between your code and the APIs you're calling. As an added bonus, the façade also allows you to simplify the interface to the API, or even multiple APIs by only exposing the parts that matter.

In the case of Bumble Bikes, Phoebe and Kitty only need basic functions that allow the robotic arm attachments to move and then call the specific API routines that allow for welding, buffing, and grabbing. The respective APIs could have thousands of exposed objects, methods, and properties between them, but we only need a few. Likewise, there might be a dozen of these APIs, but from our code's perspective, there is one library we're calling. The API could even be called in a non-obvious way such as CORBA (hopefully not), REST, or directly as an imported assembly dependency. The façade would take care of all this seamlessly. When the APIs change in future releases or are replaced with different APIs from different vendors, you only need to replace the façade. The underlying code remains untouched. If you've used an **Object Relational Mapper** (**ORM**) with a database, that qualifies as a façade because it gives you simplified access to

the database. This often allows you to switch out the database, say from Oracle to SQL Server, without changing your code.

This is how Kitty solved the API problem in the software. She created an interface that defined the behaviors of the robot arms. Then, she created a decorator, which is code that implements the interface. She then wraps the API calls, successfully mapping the abstractly defined behaviors to the specific calls to the API. Kitty, by designing the software in this way, has broken the dependency on the APIs, and by extension, prevented vendor lock-in. Any API can be decorated or wrapped following Kitty's behavioral interface. When the next generation of microcontroller APIs becomes available, Kitty just needs to write a wrapper for the new API that conforms it to her software's requirements encapsulated in the interface. The robotic control software remains closed for modification. This would not be the case if she tightly coupled it to the microcontroller's API. Had she done that, every revision to the API would require a partial rewrite, along with serious testing and validation efforts for Kitty's entire software suite. How often have you encountered, or even performed, a fix to one part of your software and the fix breaks something somewhere else? Have you ever said, "That's impossible! There's no way changing something in library A could affect the operation of library B!"? This is indicative of a program built on tightly coupled operations. Changing anything can and will have a cascading effect. The more complex the system, the more likely some stage of the cascade will have a delete-

rious effect on the overall system. This has been avoided by using the Façade pattern. While closed for modification, Kitty's program is open for extension because she can easily add new wrappers that follow the interfaces used by her software. She need only develop and test the new software, not the entire control program. The program at large is insulated from rippling failures through de-coupling. Kitty's engineering notebook documents her implementation of this pattern using *Figure 4.6*:

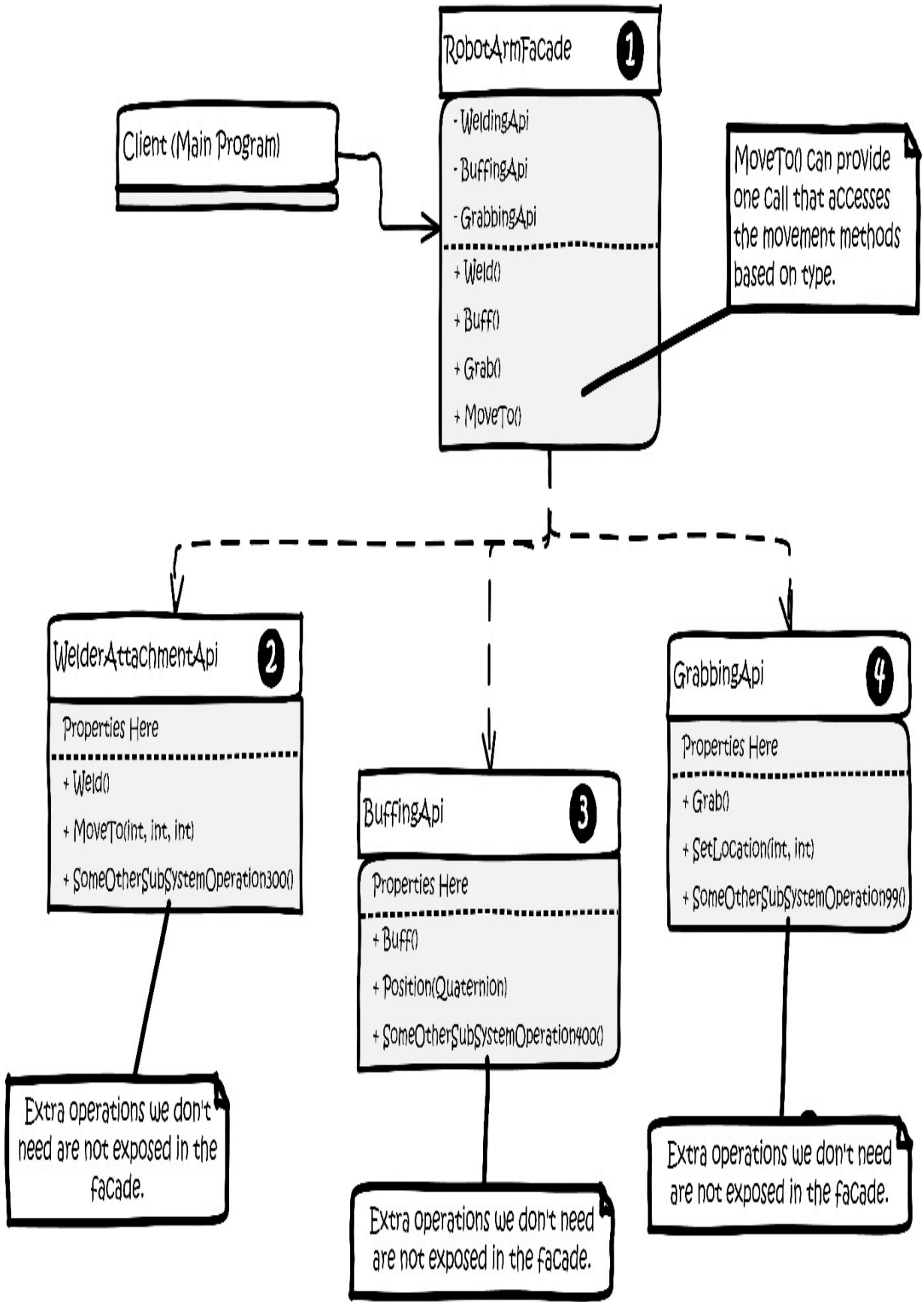

RobotArmFacade
1
- WeldingApi
- BuffingApi
- GrabbingApi
+ Weld()
+ Buff()
+ Grab()
+ MoveTo()
Client (Main Program)
MoveTo() can provide one call that accesses the movement methods based on type.
WelderAttachmentApi
2
Properties Here
+ Weld()
+ MoveTo(int, int, int)
+ SomeOtherSubSystemOperation300()
BuffingApi
3
Properties Here
+ Buff()
+ Position(Quaternion)
+ SomeOtherSubSystemOperation400()
GrabbingApi
4
Properties Here
+ Grab()
+ SetLocation(int, int)
+ SomeOtherSubSystemOperation99()
Extra operations we don't need are not exposed in the facade.
Extra operations we don't need are not exposed in the facade.
Extra operations we don't need are not exposed in the facade.

Figure 4.6: Kitty's engineering notebook diagram of her implementation of the Facade pattern.

The parts of the pattern implementation can be explained by the numbers:

1. The **`RobotArmFacade`** class is consumed by the client in place of direct references to the three third-party APIs represented by numbers 2, 3, and 4 in the diagram. Note the **`SomeOtherSubSystemOperation300()`**, **`SomeOtherSubsystemOperation400()`**, and **`SomeOtherSubSystemOperation99()`** are my clever way of hinting these third-party APIs are massive without drawing a diagram that fills the page with imaginary API methods. Our application only needs a few of these methods, so only those we need are exposed in the façade.
2. **`WelderAttachmentAPI`** represents a third-party library, perhaps a NuGet package, that controls the welding arm attachment on Phoebe's robot arm.
3. **`BuffingAPI`** represents a third-party library, perhaps a NuGet package, that controls the buffing attachment.
4. **`GrabbingAPI`** represents a third-party library, perhaps a NuGet package, that controls the grabbing attachment.

Note that many of the methods within the three third-party classes are similar.

`WelderAttachmentAPI` exposes **`MoveTo(int, int, int)`**, allowing the control software to position the welding attachment in three-dimensional space. **`BuffingAPI`** uses **`Quaternion`**, which is a three-dimensional coordinate combined with an angular rotation. If you ever work with the Unity3D game engine, which uses C# as its flagship coding language, you'll get to know and love quaternions despite their very complicated mathematical nature. Thankfully, with the Façade pattern, you don't need to fully understand the inner workings, such as Euler angles and the concept of gimbal lock, to use them. **`GrabbingAPI`** provides **`SetLocation(int, int)`** with two coordinates since that attachment only needs to move in two dimensions.

All three APIs have a common feature that moves the robot arm into position, but they are not identical in the way they are implemented or called. Each has a different method signature. This is a perfect use case for a façade because you can expose a single method to control movement and selectively call the correct API method based on which arm attachment is in use.

Similarly, each API has some method for activating is main function: **`Weld()`**, **`Buff()`**, and **`Grab()`** respectively. Again, the façade can hide the complexity of calling multiple methods. While it may seem they

achieve different goals, we can hide that complexity behind a single method that calls the correct API method based again on the type of attachment in use.

Let's look at the code in the **`RobotArmFaçade`** class. First, we have an **`enum`** that defines the attachments:

```
public enum ArmAttachments { Welder, Buffer, Grab
```

Next comes the class itself and its constituent member variables:

```
public class RobotArmFacade
{
  private readonly WelderAttachmentApi _welder;
  private readonly BuffingApi _buffer;
  private readonly GrabbingApi _grabber;
  public ArmAttachments ActiveAttachment;
```

You might note a similarity here with the Decorator pattern. We have private instances of the three APIs in the façade itself. It will be the job of the façade class to pass instructions through to the correct API in the correct format. The private members are initialized with a constructor:

```
public RobotArmFacade(WelderAttachmentApi welder,
    BuffingApi buffer, GrabbingApi grabber)
  {
    _welder = welder;
    _buffer = buffer;
    _grabber = grabber;
    ActiveAttachment = ArmAttachments.Welder;
  }
```

The **`ActiveAttachment`** member is arbitrarily set to default to the welder. Next, our façade will expose the methods for activating the currently active attachment. If it's a welder, it will weld. A grabber will grab and a buffer will buff, but it doesn't make sense to expose the methods by name. The façade makes things simpler. Kitty elects to call the façade method **`Actuate()`** and this method determines what is actually called behind the curtain:

```
public void Actuate()
  {
    switch (ActiveAttachment)
    {
      case ArmAttachments.Buffer:
        _buffer.Buff();
        break;
      case ArmAttachments.Grabber:
```

```
        case ArmAttachments.Grabber:
          _grabber.Grab();
          break;
        case ArmAttachments.Welder:
          _welder.Weld();
          break;
        default:
          throw new ArgumentOutOfRangeException();


      }
    }
```

Lastly, we need to expose a simple way to move the arms and position the attachments. This one requires a little more thought since each API method has a different method signature. Kitty decides the obvious solution is to pick the most complicated requirement, **`Quaternion`**. Quaternions are part of the .NET **`System.Numerics`** library. This struct holds four values, W, X, Y, and Z of the **`Single`** type per the documentation. Since it is the most complicated requirement, it can be made to service the simpler method signatures by ignoring the parts of the quaternion we don't need:

```
  public void MoveTo(Quaternion quaternion)
    {
      var roundX = (int)Math.Round(quaternion.X, 0)
      var roundY = (int)Math.Round(quaternion.Y, 0)
```

```
    var roundZ = (int)Math.Round(quaternion.Z, 0)
```

The two APIs, **`WelderAttachmentAPI`** and **`GrabbingAPI`**, need integers. First, we'll convert the single numbers in the quaternion into integers by rounding as shown in the preceding code. Next, we'll call the appropriate API based on the active attachment:

```
    switch (ActiveAttachment)
    {
      case ArmAttachments.Buffer:
        _buffer.Position(quaternion);
        break;
      case ArmAttachments.Welder:
        _welder.MoveTo(roundX, roundY, roundZ);
        break;
      case ArmAttachments.Grabber:
        _grabber.SetLocation(roundX, roundY);
        break;
      default:
        throw new ArgumentOutOfRangeException();
    }
  }
```

The buffer attachment requires a quaternion, so our code will just be passing through the original argument. The X, Y, and Z coordinates can be made compatible with the other two method signatures by

rounding the numbers and ignoring anything in the quaternion you don't need.

Phoebe decides her assembly line should be very simple. The robotics generally only need to worry about where along the assembly line they are positioned, so her control program will vary the X coordinate in every case. As a further measure of control, Phoebe decides to create a control program that knows precisely what the X coordinates are for each station on her assembly line.

Her control program consists of this code:

```
const int numberOfAssemblyStations = 20;
const float consistentY = 52.0f;
const float consistentZ = 128.0f;
const float consistentW = 90.0f;
```

First, we see a set of constants. We have a maximum of ten robotic arms owing to Phoebe's material and financial constraints. Since the arms can be made to work as a team, Phoebe discovers she can power 20 stations by systematically moving the robots from station to station, swapping out arm attachments as needed. She carefully calibrates her equipment and finds the ideal Y and Z coordinates. She sets them as constants along with a default rotational angle of 90 degrees, which works with all her processes.

Next, Phoebe creates an array to hold the coordinates of her 20 stations as quaternions:

```
var assemblyStations = new Quaternion
    [numberOfAssemblyStations];
```

Since the assembly line is literally a straight line, it is easy to evenly space the stations 25 feet apart along the line's X axis. A simple loop can then pre-populate the array of quaternions that represent the workstations on the assembly line:

```
for (var i = 0; i < numberOfAssemblyStations; i+-
{
  var xPosition = i * 25.0f;
  assemblyStations[i] = new Quaternion(xPosition,
      consistentY, consistentZ, consistentW);
}
```

We are now ready to set the robotic dance in motion.

Let's instantiate our **`RobotArmFacade`**, set the attachment to a welder, and move it to station zero, which is the first position in the array of quaternions. Once it's there, we'll tell it to perform a weld using the **`Actuate()`** method on the façade:

```
Console.WriteLine("RobotArm 0: Robotic arm contro
    activated!");
var robotArm0 = new RobotArmFacade(new
    WelderAttachmentApi(), new BuffingApi(), new
        GrabbingApi());
Console.WriteLine("Initializing welder function i
robotArm0.ActiveAttachment = ArmAttachments.Welde
robotArm0.MoveTo(assemblyStations[0]);
robotArm0.Actuate();
```

Next, let's move the arm to station **3** where we need a buffer to smooth out a metal extrusion on a bicycle frame. Once the arm is in place, we'll buff using the **`Actuate()`** method:

```
Console.WriteLine("Initializing buffer function i
robotArm0.ActiveAttachment = ArmAttachments.Buffe
robotArm0.MoveTo(assemblyStations[3]);
robotArm0.Actuate();
```

Splendid! The arm is now needed to grab a part and hold it in place for painting at station **7**:

```
Console.WriteLine("Initializing grabber function
    in arm 0");
```

```
robotArm0.ActiveAttachment = ArmAttachments.Grabb


robotArm0.MoveTo(assemblyStations[7]);
robotArm0.Actuate();
```

Initially, we needed to deal with three different APIs from three different vendors to work with three different pieces of hardware. By using the Façade pattern, we were able to deal with one common interface for all three APIs, which isolates the bulk of our code from changes made in the API. When the API changes, we may need to change the façade, but we won't need to change anything else.

The Composite pattern

Phoebe continued to work on the electronics. Kitty, however, was starting to worry about some of the fundamental considerations for her designs. Initially, the girls had agreed to use commercially available components, but Phoebe realized they could manufacture all the parts they need themselves. That way, assuming Bumble Bikes had access to all the raw materials, such as aluminum alloys, plastic, and rubber, they could have tighter control over the cost, durability, and weight of their final product. These factors influence everything from how Bumble Bikes sources its raw materials to the final sales price. The final sales price is factored by adding in the cost of goods sold, or

in this case, the cost to manufacture, package, and deliver a bicycle. Kitty had some preliminary spreadsheets on her iPad. They were complicated, though. Kitty really wanted to ditch the two-dimensional thinking presented by her spreadsheets and come up with a better way to represent the cost of making a bicycle.

Kitty opened her backpack and her heart sank. Her iPad, or what was left of it, spilled on to the worktable as a collection of broken glass and jagged plastic shards. She remembered her crash at Big Bend last weekend. She had misjudged the drop on a 3-foot (1-meter) ledge on Black Gap Road, flipped over her front handlebar, and landed on her back. Her backpack, which contained her iPad, had broken her fall. Everything was on that iPad! Thankfully, her father had drilled the number one house rule into her head. *"Always protect the gear!"* he would say. *"We make our livelihood with our tablets and our laptops. Others use them to play games and watch movies. There's nothing wrong with that, but we use ours to make our mortgage, so take care of your gear!"* He would usually shout this epithet very early in the morning when he would invariably trip trying to avoid someone's tablet, phone, or computer that had been left on the floor right along the path he took as he sleepily lumbered to the kitchen in the mornings in search of caffeine. Owing to this sage wisdom, the tablet was backed up. When Kitty had purchased it in Dallas from her favorite computer store, she had opted to get the extra 99 USD replacement guarantee that covered everything, including accidental dam-

age. The new iPad arrived in the mail the next day and Kitty, being a student of industrial design, reveled in the unboxing of her new digital compatriot.

For all the criticisms she had about Apple products, such as the inability to upgrade or repair them, there was one thing nobody could deny. They have the coolest packaging of any product in the tech industry. The packaging itself is a work of industrial art: from the heavier than necessary gauge of cardboard to the way all the packages fit together to take up as little space as possible.

As Kitty was waiting for her cloud backup to restore all her data and apps, a lightning bolt thought struck her. Apple had solved the very problem she was considering. The way the packages fit together, where some are inside of others, reminded her of a tree structure. The iPad shipped with the tablet, a charger, a USB-C style cable, and a tiny box containing a beautifully printed instruction manual that everybody just throws away. There was a warranty card and a few other printed cards touting Apple's responsible stance on the environment and an advert for AppleCare, Apple's own service plan. The iPad itself was housed inside a coated cardboard insert where all the pieces in the package fit together precisely. The charger and cord fit in a niche below a sunken niche carved out for the iPad itself. Everything was coated with plastic, so nothing would be marred or scratched during shipment. The intricate package, depicted in *Figure*

4.7 in a way that will not get me sued, contained as many small boxes and cardboard pieces as the actual electronics they held. Naturally, a product designer at Apple could tell precisely how much any of the smaller boxes or wrapped components within the larger package weighed at any time, as well as how much each of the intricate die-cut cardboard components cost to make. That designer probably agonized for months looking for ways to shave fractions of pennies off the package material cost, just as the iPad's designers had stressed over weight, power, and heat concerns:

Figure 4.7: Kitty's new tablet came intricately packaged. She realized it could be modeled as a tree structure, and perhaps bicycle components might be modeled in the same way.

So far, Kitty had successfully created class models for high-level bicycle components, such as the frame. She and Phoebe had commit-

ted themselves to the idea that they would manufacture everything for the bicycles themselves, including the crankset. The crankset consists of all the parts that make the bicycle move when you push down on the pedals. Bumble Bikes intended to position itself on the market as a cut above big-box stores in terms of its build quality. That meant every gram would be scrutinized by sophisticated customers who were conditioned to having to buy a bicycle, discard most of the components it came with, and replace them with better, lighter components. There is always a perfect balance between the weight and cost of the crankset for many riders. The racing community will pay a premium for parts that are several grams lighter, while casual riders want a product that's less expensive and they don't mind the extra weight.

The crankset, as with the boxes, can be modeled as a tree, as we'll soon see. If you're not personally familiar with the parts on a bicycle that comprise a crankset, and you're curious, see Kitty's CAD drawing in *Figure 4.8* where she was kind enough to diagram most of them for us:

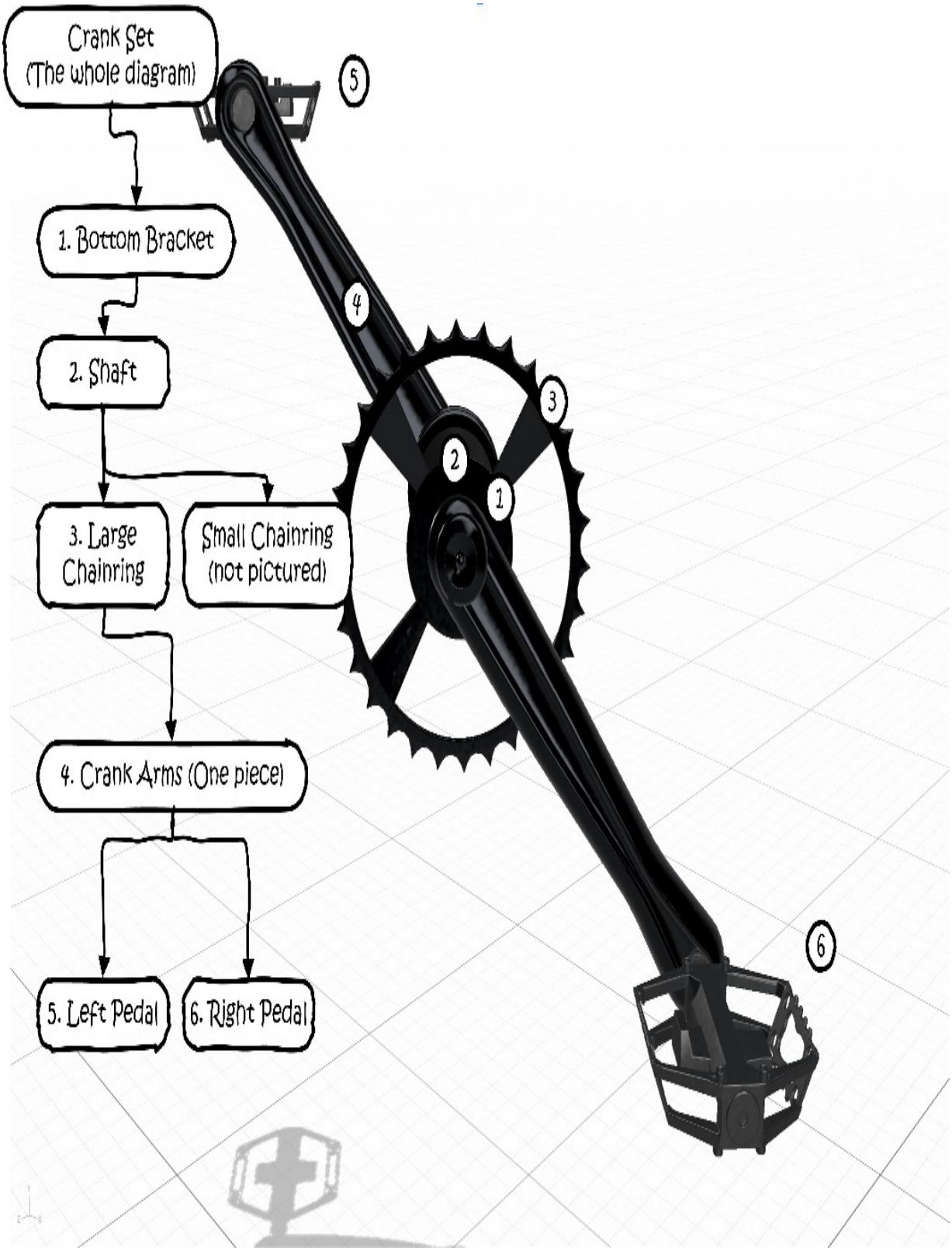

Crank Set
(The whole diagram)
1. Bottom Bracket
2. Shaft
3. Large Chainring
Small Chainring
(not pictured)
4. Crank Arms (One piece)
5. Left Pedal
6. Right Pedal
5
4
3
2
1
6

Figure 4.8: The crankset for a typical bicycle involves components that can be nested in a tree-like structure to solve our immediate problems with cost and weight.

When you are presented with modeling a tree-like structure of objects that can conform to a common interface, you should immediately think of the Composite pattern. The Composite pattern allows you to compose a tree of objects and then work with the structure as if it were a single object. The tree is composed of containers and leaves. A leaf is a tree element that doesn't have any sub-elements. A container is a tree element that has other leaves and containers within it. Graphically, this looks similar to a file folder structure on your computer. The files are leaves and the folders are containers:

C
D
E
0
1
C:Windows
C:\Windows
..
addins
appcompat
apppatch
AppReadiness
assembly
bcastdvr
Boot
Branding
CbsTemp
Containers
CSC
Cursors
debug
diagnostics
DiagTrack
DigitalLocker
en-US
GameBarPresenceWriter
Globalization
Help
IdentityCRL
IME
ImmersiveControlPanel
INF
InputMethod
L2Schemas
LiveKernelReports
Logs
Microsoft.NET
Migration
Minidump
ModemLogs
OCR
Offline Web Pages
Panther
Performance
PLA
PolicyDefinitions
Prefetch
PrintDialog
Provisioning
Registration
RemotePackages
rescache
Resources
SchCache
schemas
security
ServiceProfiles
ServiceState
servicing
Setup
ShellComponents
ShellExperiences
SKB
SoftwareDistribution
Speech
Speech_OneCore
System
System32
SystemApps
SystemResources
SystemTemp
SysWOW64
TAPI
Tasks
Temp
tracing
twain_32
Vss
WaaS
Web
WebManagement
WinSxS
bfsvc.exe
DirectX.log
DPINST.LOG
DtcInstall.log
explorer.exe
HelpPane.exe
hh.exe
lsasetup.log
MEMORY.DMP
mib.bin
notepad.exe
NvContainerRecovery.bat
PFRO.log
Professional.xml
regedit.exe
setuperr.log
skipsavetoini
splwow64.exe
system.ini
twain_32.dll
win.ini
WindowsUpdate.log
winhlp32.exe
WMSysPr9.prx
write.exe
Files are leaves
Folders are containers that hold other containers and leaves
1/99 Object(s) selected
12/7/2019 3:14:52 AM
(Free 1.51 TB)
C:\Windows\

Figure 4.9: The file structure of your computer's hard drive is represented as a tree consisting of leaves (files) and containers (folders).

We can model part of our group set this way as well. For our mechanically inclined readers, I'm not necessarily suggesting these parts literally fit inside each other in the physical world. I'm suggesting they can be modeled this way to solve the problem at hand in terms of weight and cost. In this model, the crankset consists of the bottom bracket, which is basically a big hole at the bottom of the bicycle frame fitted with bearings, and a shaft extends through the bottom bracket. The shaft is connected to a set of chainrings. Most bicycles have one, two, or three chainrings depending on what kind of bicycle it is. Road bikes usually have two: a large chainring for general riding and a small chairing that is used to climb hills. In our model, the small chain ring is treated as a leaf where all other components so far have been containers.

The chainrings connect to the crank arms, which are one big piece, even though they might appear to be two separate arms. The arms attach to the left and right pedals, which are the leaves and the end of our tree structure.

The composite pattern allows you to work with complex tree structures elegantly by allowing you to make full use of recursion and poly-

morphism. Just be careful that you work with classes that have a very common interface. You might be introducing code smell if you must shoe-horn a bunch of classes that don't really fit together. This pattern pairs nicely with the Builder pattern already in use because the builder can be made to assemble the tree structure. The basic structure of the composite pattern is shown in *Figure 4.10*:

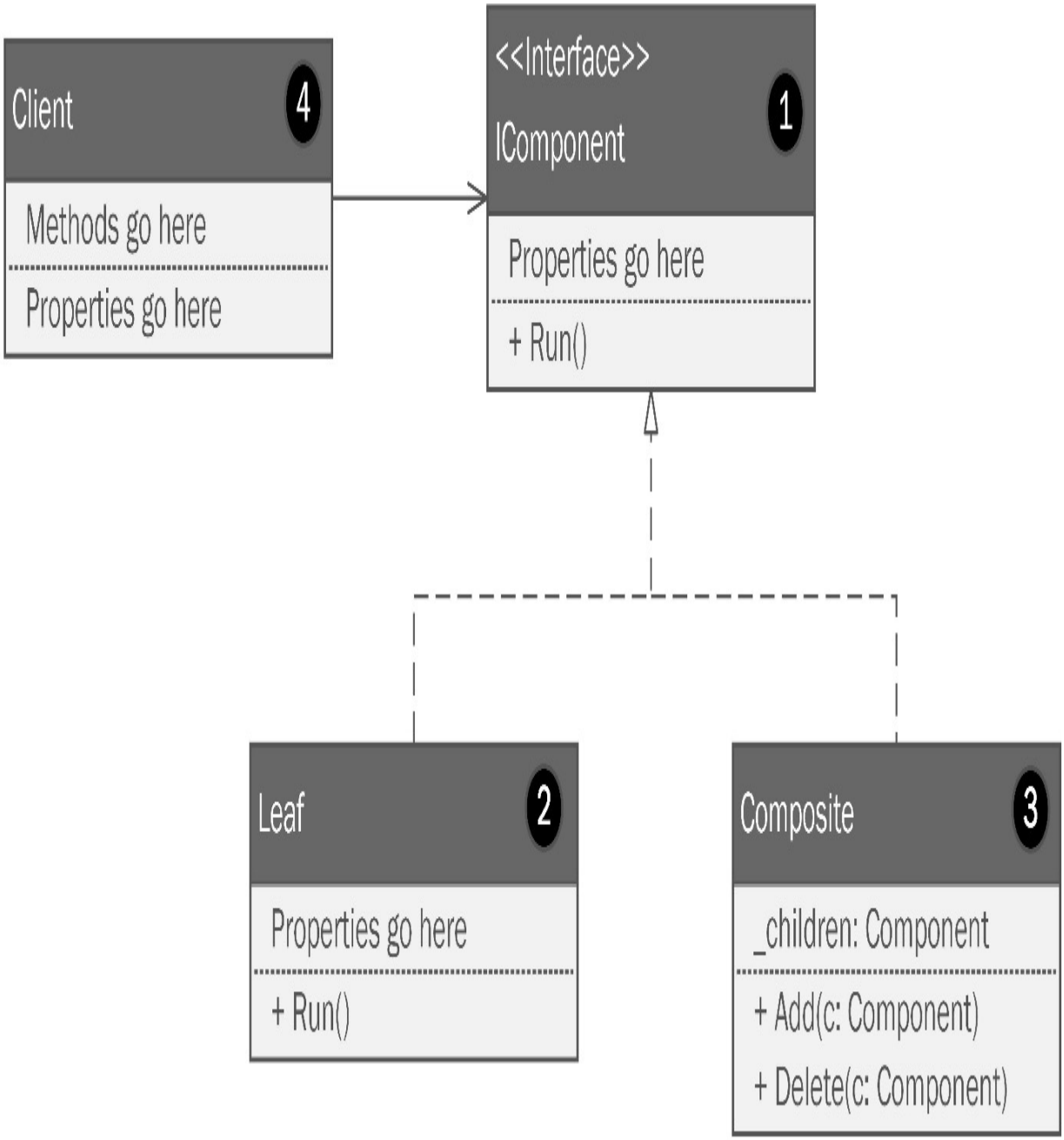


Figure 4.10: The Composite pattern.

Let's understand this figure in detail:

1. The **`Component`** class will implement some interface with the methods needed to access the overall functionality. Here, we're calling

that **`Run()`**. Components can contain leaves and other components.
2. The **`Leaf`** class represents nodes in the tree that can't contain anything else.
3. The **`Composite`** object allows for the creation and maintenance of a tree structure using components and leaves.
4. The client program accesses the composite tree as it needs and can treat simple and complex objects identically.

I want to reiterate that we are *not* attempting to model the physical structure of our bicycle. This model is a cost model and defines the relationships within a group of components, not the order of physical assembly. The fields we're interested in are cost and weight, and those fields will form an interface that describes the common properties of any bicycle component regardless of its form, construction, or purpose.

Kitty's version looks as follows in *Figure 4.11*:

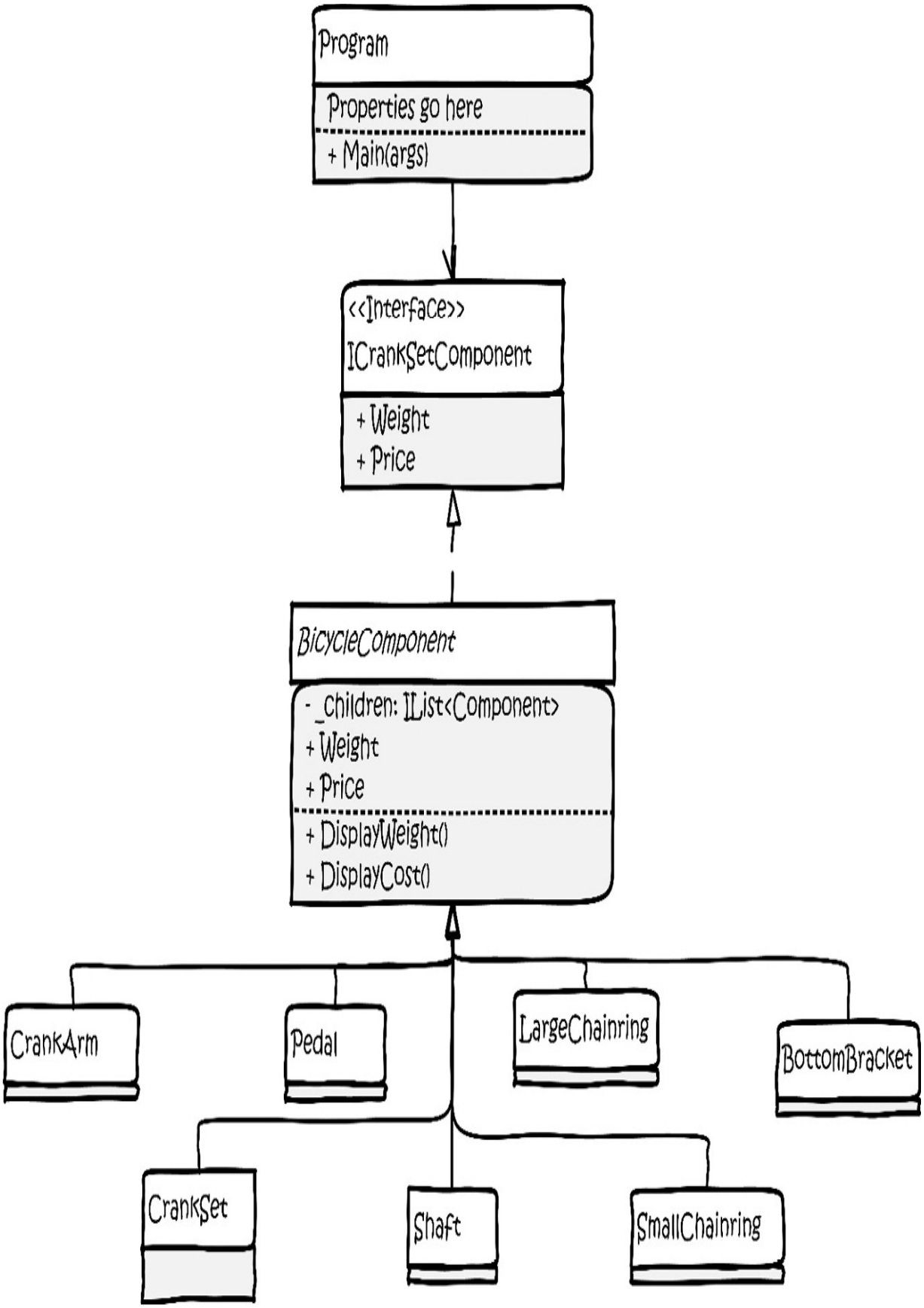

Program
Properties go here
+ Main(args)
<<Interface>>
ICrankSetComponent
+ Weight
+ Price
BicycleComponent
- _children: IList<Component>
+ Weight
+ Price
+ DisplayWeight()
+ DisplayCost()
CrankArm
Pedal
LargeChainring
BottomBracket
CrankSet
Shaft
SmallChainring

Figure 4.11: Kitty has changed the structure of the basic pattern found in Figure 4.x to suit her needs. Note this isn't the diagram of the full hierarchy, it's just the structure of the pattern

The implementation of this pattern only requires one abstract class and a horde of concrete classes based on the abstract class. The following are the contents of Kitty's abstract **`BicycleComponent`** class, which forms the vital common interface needed by the pattern. We need two private properties to hold the **`weight`** and **`cost`** values of the component:

```
public abstract class BicycleComponent
{
  private float Weight { get; set; }
  private float Cost { get; set; }
```

Next, we need a list to hold any subcomponents. Kitty specified an interface for the type instead of directly coupling to the **`List<>`** class:

```
  public IList<BicycleComponent> SubComponents;
```

These three properties are initialized in a constructor where we pass in **`weight`** and **`cost`** as floats. The **`SubComponents`** list is initialized as an empty list:

```
protected BicycleComponent(float weight, float
{
  SubComponents = new List<BicycleComponent>();
  Weight = weight;
  Cost = cost;
}
```

Next, we need two methods to display the weights and costs of the components along with any subcomponents. These can leverage recursion to print out the entire tree, which is handy for Kitty's cost analysis. We only need to do this on containers, not leaves, since leaves will be printed with their containers. We determine whether we're dealing with a leaf by checking **`SubComponents.Count`**. If it is zero, we are dealing with a leaf, and we simply return. Otherwise, we loop and print the weight and cost of the subcomponents:

```
public void DisplayWeight()
{
  if (SubComponents.Count <= 0) return;
  foreach (var component in SubComponents)
  {
    Console.WriteLine(component.GetType().Name
        " + component.Weight);
    component.DisplayWeight();
  }
```

```
    }
```

Here, we do the same thing with cost:

```
    public void DisplayCost()
    {
      if (SubComponents.Count <= 0) return;
      foreach (var component in SubComponents)
      {
        Console.WriteLine(component.GetType().Name
            $" + component.Cost + " USD");
        component.DisplayCost();
      }
    }
  }
```

The concrete code for the Composite pattern is generally very repetitive, as you'll see. In fact, Kitty does something here she doesn't do anywhere else. She makes one file containing many classes. She names the file **`CompositeParticipants.cs`**, the partial contents of which follow in the code below. She did it this way because effectively it's a collection of very simple concrete classes that inherit from a base class. If you want to see the whole class, consult Kitty's code in the chapter's sample code project:

```
public class Pedal : BicycleComponent
{
  public Pedal(float weight, float cost) : base(w
    cost)
  {
  }
}
public class CrankArm : BicycleComponent
{
  public CrankArm(float weight, float cost) : bas
    cost)
  {
  }
}
public class LargeChainRing : BicycleComponent
{
  public LargeChainRing(float weight, float cost)
    base(weight, cost)
  {
  }
}
```

As you can see, this isn't rocket surgery. Every part is simply modeled as a concrete implementation of the abstract class. I've shown

three classes here. There are seven in total that all look the same save for the class name.

At first glance, this might seem odd until you see how the composite's tree is constructed in the client code within the **`Program.cs`** file. Within that file, Kitty builds the tree from the bottom up. This isn't a requirement, but it does make it easy to understand. The leaves at the bottom of the tree are the petals or pedals:

```
var leftPedal = new Pedal(234.14f, 11.32f);
var rightPedal = new Pedal(234.14f, 11.32f);
```

The pedals connect to the crank arm. I'm suddenly reminded of the old song *Dem Bones* where the toe bone is connected to the foot bone. The foot bone is connected to the heel bone. The song continues up to the head bone followed by an invocation of the songwriter's creator. Here, the pedal bone is connected to the crank arm bone, except they aren't bones:

```
var crankArm = new CrankArm(432.93f, 34.32f);
crankArm.SubComponents.Add(leftPedal);
crankArm.SubComponents.Add(rightPedal);
```

We create the instance of **`CrankArm`**, then add the pedals to its **`SubComponents`** list. **`crankArm`** is connected to **`largeChainRing`**. So is **`smallChainRing`**, which itself becomes a leaf:

```
var largeChainRing = new LargeChainRing(57.0983f,
var smallChainRing = new SmallChainRing(52.57f, 1
largeChainRing.SubComponents.Add(smallChainRing);
largeChainRing.SubComponents.Add(crankArm);
```

The large chainring is connected to the shaft:

```
var shaft = new Shaft(82.03f, 19.55f); // can you

shaft.SubComponents.Add(largeChainRing);
```

The shaft fits through the bottom bracket:

```
var bottomBracket = new BottomBracket(284.834f, 1
bottomBracket.SubComponents.Add(shaft);
```

That's our crank set, but I'll add a top-level instance of **`CrankSet`** and pass in zeros for the cost and weight since the crank set itself comprises its subcomponents:

```
var crankSet = new CrankSet(0f, 0f);
crankSet.SubComponents.Add(bottomBracket);
```

Now for the magical part of our show. We'll call the two methods and get a recursive detail of the whole structure:

```
Console.WriteLine(" ---------- Weights ----------
crankSet.DisplayWeight();
Console.WriteLine(" ------------ Cost ----------
crankSet.DisplayCost();
```

The result of the run is shown in *Figure 4.12*:

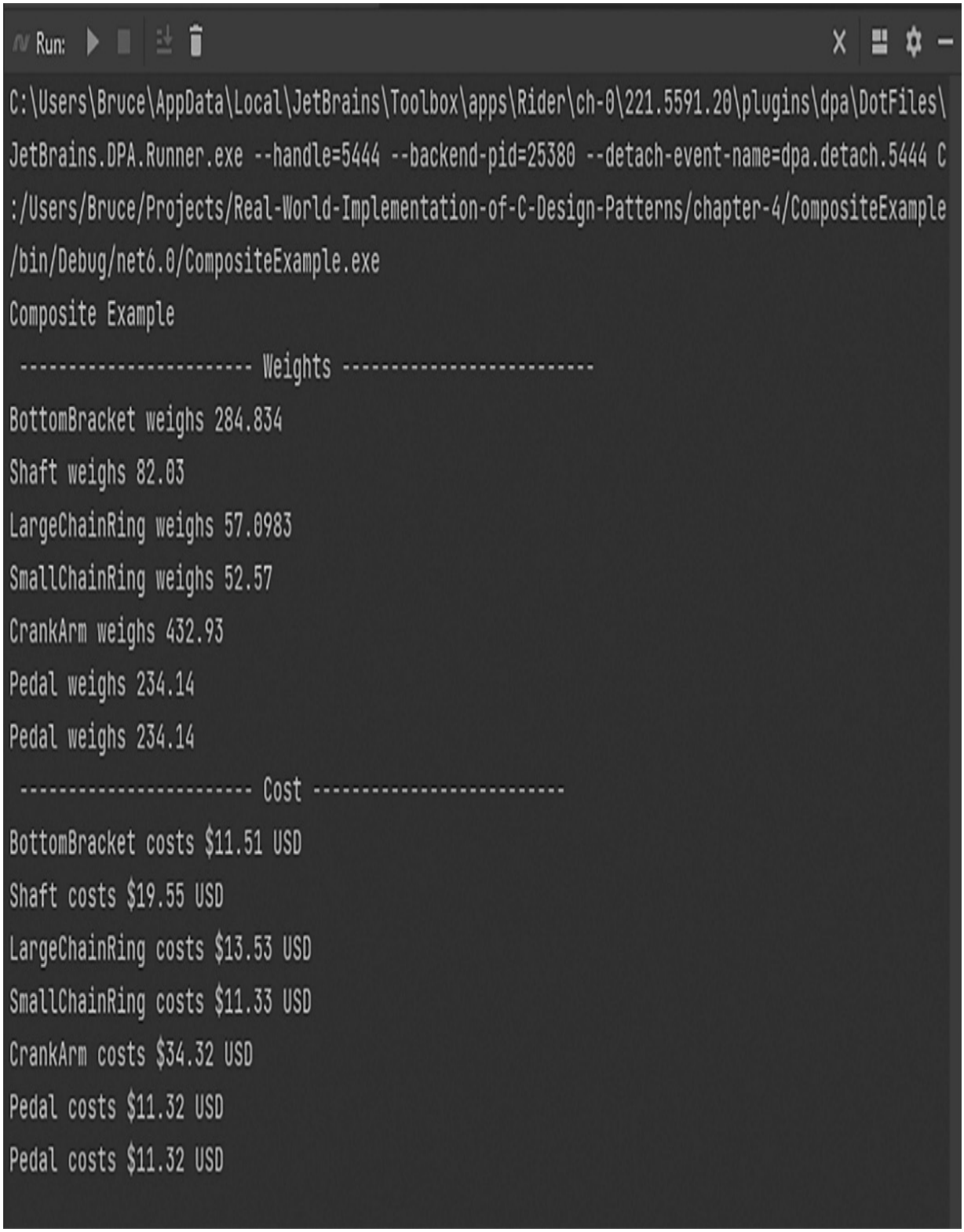


Figure 4.12: The run results of our Composite pattern project.

The Composite pattern is used whenever you need to process a hierarchical structure as a tree. The main requirement for the pattern to be effective is that every node in the tree must conform to a common interface. If that can be managed, you can use this pattern to process the tree in any manner you might need. You can add new class types to your tree, so long as they conform to the common interface. Using this pattern, you can create novel processing capabilities while honoring the open-closed principle. Recursion and polymorphism can be exploited to expedite your processing. The client code will treat nodes and containers identically since they have a common structure, which is really the hard part. You have to find a way to make everything in the tree conform to the common interface, which isn't always easy.

The Bridge pattern

The **Bridge pattern** is a structural design pattern that lets you split a large class or set of closely related classes into two separate hierarchies: abstraction and implementation. Kitty and Phoebe set up a Kickstarter page to promote Bumble Bikes and gauge the interest on the market. Backers can preview and pre-order the *Palo Duro Canyon Ranger*, Bumble Bike's flagship mountain bike design. The project has been well received, but the Kickstarter backers are complaining about the lack of color choices on the bikes. In the original design, the girls purposefully limited the color choices because they

were using inheritance for almost everything. The problem with using inheritance is becoming a clear theme: it can lead to a run-away proliferation of classes, as seen in *Figure 4.13*. Can you imagine supporting 20 colors per bicycle model, and expanding to 20 models of bicycles? That's a lot of subclasses!

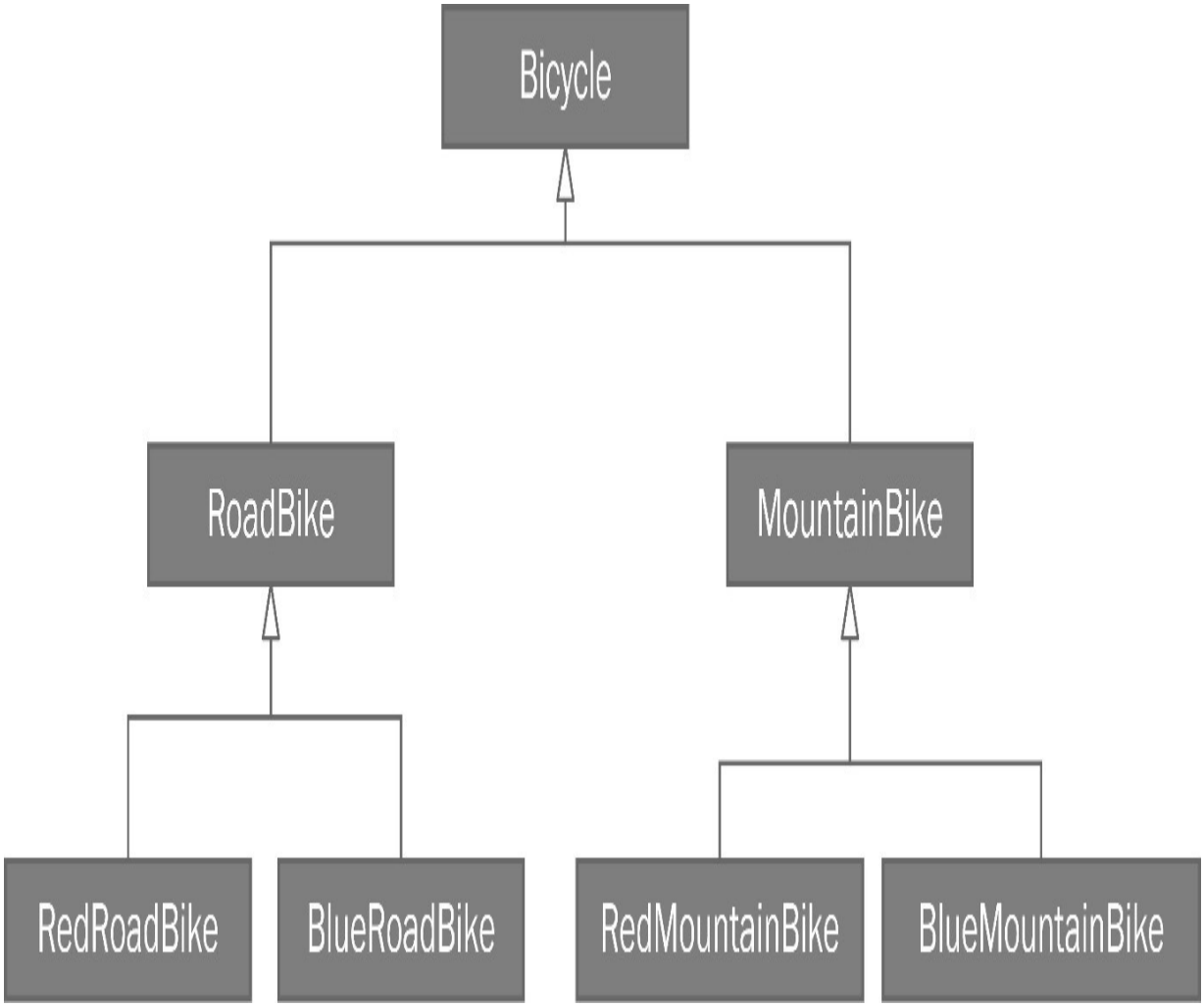


Figure 4.13: Class proliferation may sound as if it's a Marxist political construct, but it becomes a problem when you overuse inheritance (we clearly need a better way to represent a finite set of supported colors for our bicycles).

Perhaps the simplest way to solve this is to simply create a property on your base bicycle class to hold the color. Maybe have it store a common color structure such as **Red Green Blue** (**RGB**) or **Cyan Magenta Yellow Black** (**CMYK**), which is the color model used by printers. This works fine if you're dealing 100% with software and you need to represent a color within a gamut supported by your user's graphics card or printer.

In an industry such as bicycle manufacturing, this won't work, because we aren't just representing any possible color of light or paint. We need to represent a finite set of paints to be mixed and applied using machinery. This dimension of realism means there are limits to how the color system can work. The girls must keep the base colors and topcoats in stock, and they must take into account the setup and cleaning costs of their painting machinery. Limiting each bike to one available color handles all this nicely without any design problems. In effect, they were counting on controlling a variable in their software by making a single color a business requirement. As it turns out, the market won't bear that constraint. Competitors can offer a range of colors. Phoebe herself remembers the most important aspect of her new bike when she was 9 was that it was pink. She didn't care about where it came from or whether it had a fancy label on the frame. It had to be pink.

When we looked at the Decorator pattern earlier in this chapter, the problem was similar. We added external features, such as bells and lights, which could also be represented with an exponentially growing tree of subclasses. So, why not use a decorator here? Perhaps, you could think of a paint job as a decorator. Decorators, though, are designed to stack. We can stack a bell, headlights, taillights, fenders, mirrors, and even bicycle theft alarms on top of one another within the object hierarchy to build the perfect bicycle without altering the abstract bicycle base class. There's something not quite right about stacking paint jobs or even stacking your bell or lights onto a paint job. The paint, conceptually speaking, is more a part of the frame than something that decorates it. Beware of learning a pattern and then wielding it as a golden hammer. The decorator doesn't really fit here, even though not using the pattern has a similar side effect.

When you read about the Bridge pattern in the GoF book, and I always encourage you to go to the original academic sources when you can, you will find it couched in very academic language. They talk about a bridge between an abstraction and its implementation. We can vary each independently of one other by keeping the two separate. We have a business requirement for a new dimension that is an integral part of our bicycle's frame and we need to vary that dimension independently of the abstract bicycle. When you want to vary two or more dimensions independently, while avoiding a combinatorically increasing number of subclasses, you need the Bridge pattern.

You can see a representation of the Bridge pattern in *Figure 4.14*. I may have had a little fun with the visuals, but it does make it easy to see why this pattern is called a *bridge*:

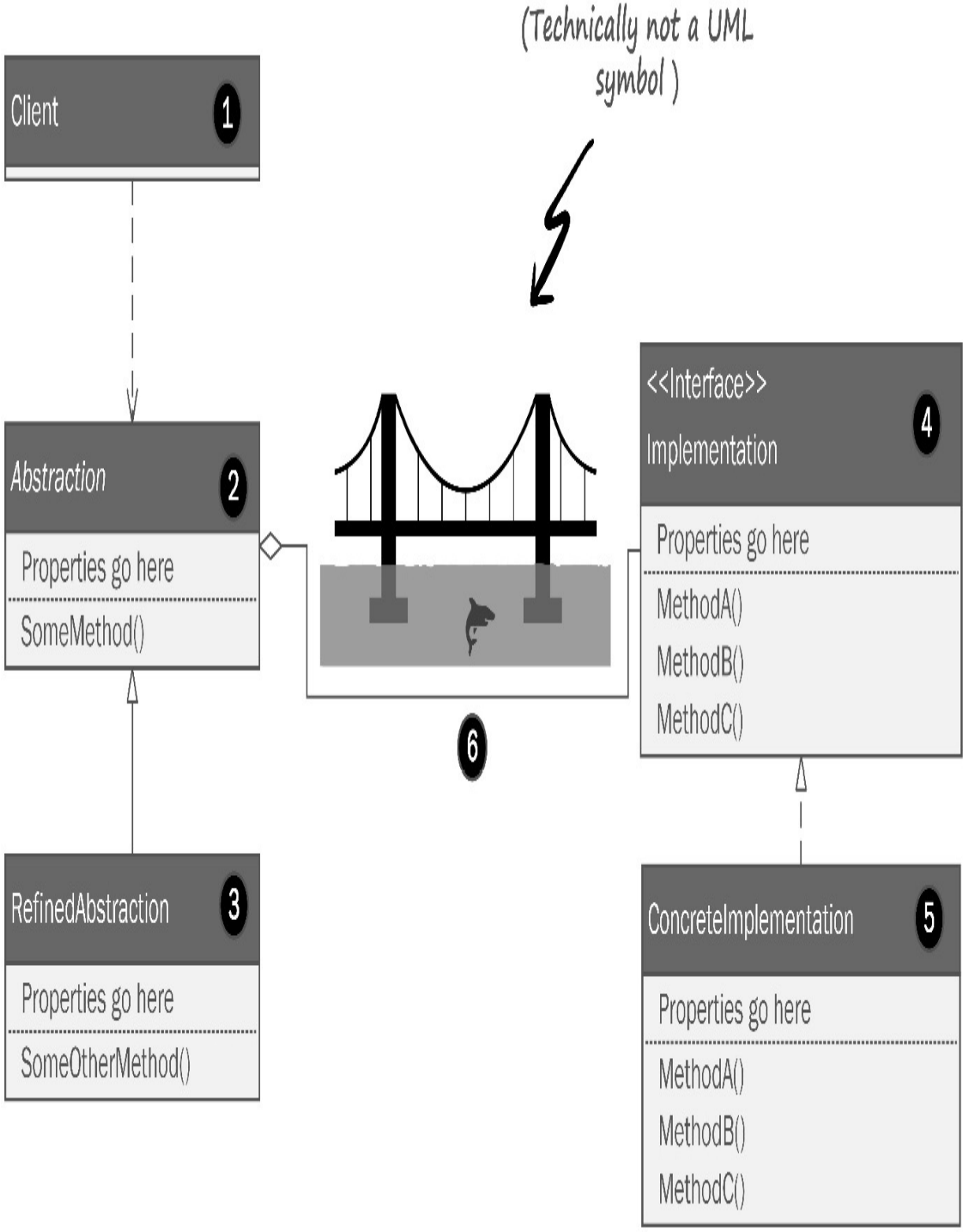

(Technically not a UML symbol)
Client
1
Abstraction
2
Properties go here
SomeMethod()
RefinedAbstraction
3
Properties go here
SomeOtherMethod()
<<Interface>>
Implementation
4
Properties go here
MethodA()
MethodB()
MethodC()
ConcreteImplementation
5
Properties go here
MethodA()
MethodB()
MethodC()
6

Figure 4.14: The Bridge pattern allows you to vary two sides of your object structure independently along two different dimensions.

Let's review the elements of the Bridge pattern in our diagram by the numbers:

1. This is the client that accesses the functionality in the abstraction.
2. This is the abstraction side of the bridge. This is usually the class structure that you started with and the structure that worked well until you realized you had additional dimensions you needed to model.
3. Refinements to the abstraction are subclasses that make sense and that don't create a runaway multi-dimensional class structure. We have several refinements by way of four different types of bicycles that inherit from the abstract bicycle class.
4. The implementation interface is on the other side of the bridge. This is where you model the dimension that will vary independently of the abstraction. Note the bridge itself between the two consists of a compositional relationship. The abstraction has an implementation of the second dimension.
5. This is the concrete implementation of the second dimension based on the interface.
6. That's a shark. Studies have shown that you are 68.342% more likely to be attacked by a shark if you don't use the Bridge pattern

to decouple your complicated classes. It doesn't even matter if you live inland. Nobody knows why. It's SCIENCE! Don't argue.

Let's take a look at Kitty's version of the Bridge pattern diagram applied to the bicycle paint problem in *Figure 4.15*:

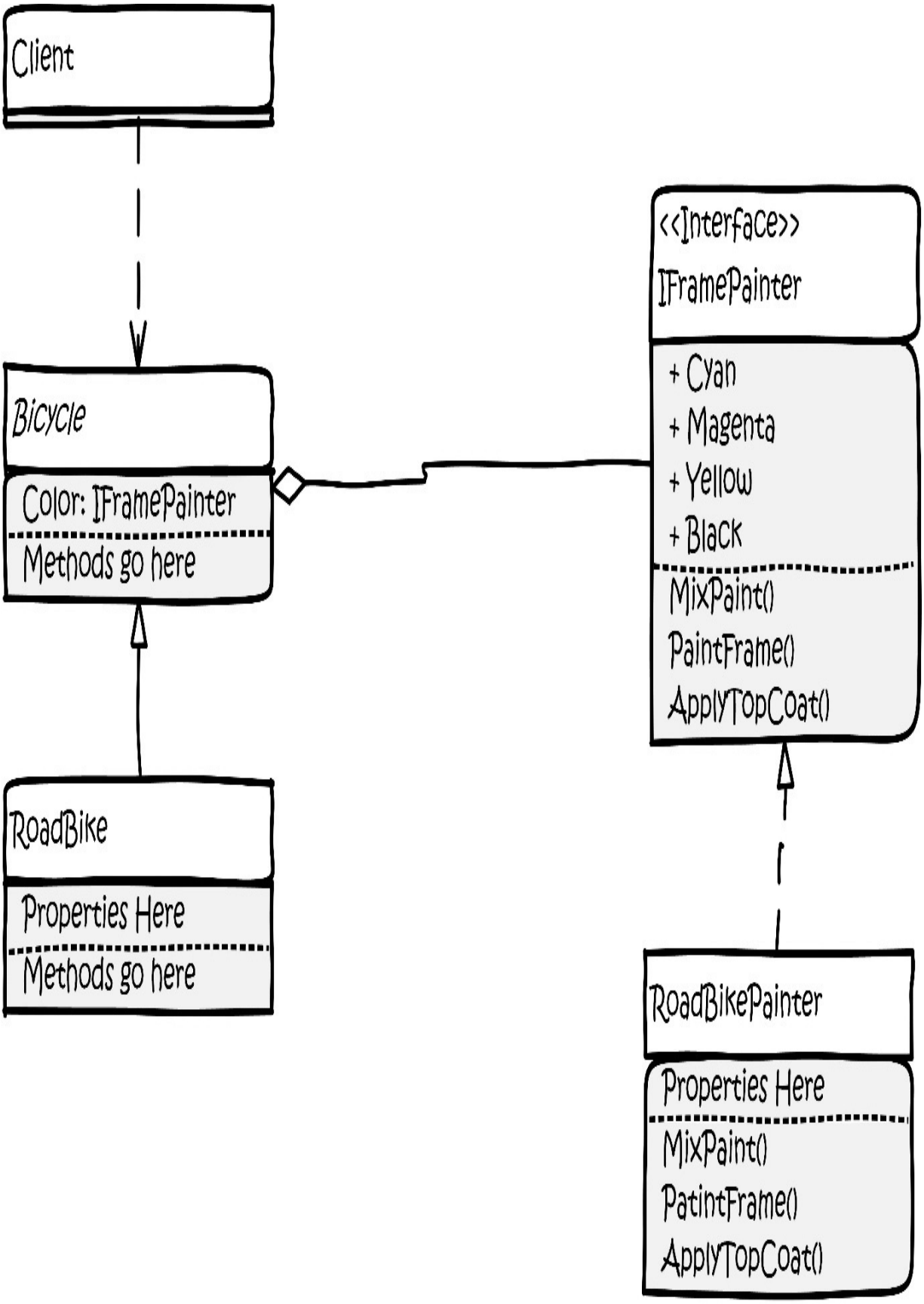

Client
Bicycle
Color: IFramePainter
Methods go here
RoadBike
Properties Here
Methods go here
<<Interface>>
IFramePainter
+ Cyan
+ Magenta
+ Yellow
+ Black
MixPaint()
PaintFrame()
ApplyTopCoat()
RoadBikePainter
Properties Here
MixPaint()
PatintFrame()
ApplyTopCoat()

Figure 4.15: Kitty's whiteboard rendition of the bridge pattern applied to her paint problem.

"Whoa, whoa, WHOA! Wait one fluffy minute!" Phoebe exclaimed as she stomped her foot on the ground. *"You can't do this! The only way to make this work is to break the open-closed principle! We've held this rule as sacrosanct!"* Kitty stared blankly at the board. Phoebe is right. She usually was, especially when she stomped her feet, a trait she got from her mother.

"We already have a color attribute in the abstract Bicycle class," Kitty observed. *"But because it's a stinkin' enum, we can't extend it, nor can we subclass it,"* Phoebe added. *"What about a decorator?"* Kitty asked. *"Maybe, but it seems as though we're going to wind up with a lot more code and complexity than we would if we just changed the original design,"* Phoebe said. Neither wanted to give up. For the next few hours, the two pored over articles on other patterns that might help them. The problem here is that the original model used an enumeration to define the colors. A limitation of the C# language holds that an enumeration cannot be extended.

If they subclass or decorate the original **`IBicycle`** interface or the abstract **`Bicycle`** class, they'll need to add the **`PaintJob`** attribute, but they'll be stuck with the existing **`Color`** attribute. Inheritance doesn't give us a way to hide or remove deprecated code. Covering it up with

a façade doesn't feel right either. Even if you covered the unused member with a façade, it would also change the interface you use to work with the bicycle classes.

"Gaga! We are going to have to change the type in the bicycle base class!" Phoebe said angrily. "Gaga" is a word Phoebe had trained herself to say in place of less socially acceptable words, such as the ones most people say when they stub their toe really, really, hard. The two railed against the realization with the fervor of defense attorneys trying to free an innocent man accused of capital murder. They pored over software engineering books and books on patterns. A common thread in all those books is that never once did any of the authors ever make a design mistake. They either presented trivial examples such as shapes, circles, and squares, or they presented patterns in a way that was too hypothetical – `ClassA` inherits from `ClassB`, and so on. The examples weren't very useful, but they were very safe, as far as examples go. The GoF book presents real-world use cases via a windowed software project mainly aimed at Unix users working with SmallTalk, which isn't widely used outside of government and academic circles.

Late one night, the sisters found the clarity that had been eluding them. Two of the most important books in the world of agile development with patterns are Robert Martin's books titled *Agile Software Development, Principles, Patterns and Practices*, as well as the re-

write for C# developers titled *Agile Software Development, Principles, Patterns and Practices in C#*. The GoF book introduced the world to formalized software patterns, but Uncle Bob's books introduced the world to SOLID and agile principles coupled with those patterns. Don't worry. They aren't tightly coupled. That would be a horribly recursive nightmare.

If you're reading the e-book version of this book, assuming there is one, get out your e-highlighter for the next three sentences. Analog highlighters can be used on analog books.

Uncle Bob reminds us that it is impossible to foresee every design consideration ahead of time. Nobody is prescient, which is a point we made in the first chapter of this book. Lack of prescience leads to design problems that are usually solved with stovepipe-style fixes.

It may seem as though we're abandoning our open-closed principle. You should never abandon the SOLID principles. But sometimes you must break them for the greater good. Martin's books describe this as taking a bullet and realizing the opportunity to learn from the mistake. He also reasons for you to consider every element within your code that can be refactored in an identical manner with the hope that you need only to take one such bullet. In the real world, as software grows, you will need to adhere to good principles, but you can't be shackled by them.

Patterns can help you avoid the need to change the base class, but occasionally you will need to change the design because not changing it will make everything worse. Equally as important is the exercise of removing dead code from the software. This too necessitates breaking SOLID principles. It is far better to remove the dead code than leave it in for the sake of dogma.

In this particular case, changing the way we implement the property allows us to serve our business requirements in a flexible way true to the spirit of SOLID principles without being fettered by them. This is a call you should find yourself making on rare occasions. Changing the interface to use the new **`IPaintJob`** interface adds significant business value to the overall design.

We now have a way to model the bicycle, which we've had for a while, and a business-savvy way to model the paint job. Kitty and Phoebe can now offer their customers bicycles in a variety of colors. We've gone from limiting customers to a finite set of one color per bicycle to a set that is limited only by the gamut of the paints. We've shifted from a software limitation on the business to a limitation of real-world chemistry and machinery. I've personally always held that business should define software, not the other way around.

The Bridge pattern isolates the **`Bicycle`** classes from being tightly coupled to an equally complicated model. There is little doubt that

over time, the painting model and the bicycle model will continue to grow in complexity, but now they do so in isolation.

Let's look at the code for the Bridge pattern. For this book's example code, I'll be leaving all the code we have written so far intact. In real life, I'd begrudgingly modify the **`IBicycle`** interface. For the sake of the book and continuity, I'll be putting Kitty and Phoebe's code into a slightly different format. Phoebe sternly objected, but the editor stepped in and she ultimately capitulated.

I'll start by making a new interface to use in place of **`IBicycle`**. All I'm going to do here is take out the offending **`Color`** property:

```
public interface ISimplifiedBicycle
{
  public string ModelName { get; set; }
  public int Year { get; }
  public string SerialNumber { get; }
  public BicycleGeometries Geometry { get; set; }
  public SuspensionTypes Suspension { get; set; }
  public ManufacturingStatus BuildStatus { get; s
  public void Build();
}
```

The bridge we're building has two sides. The bicycle side is represented by this new **`ISimplifiedBicycle`** interface, and the implementation side is used to independently model a complex object that describes how the bicycles might be painted.

Since classes and interfaces describing a paint job will probably be used across several different projects, Kitty adds a new namespace to the **`BumbleBikesLibrary`** we saw her create in *Chapter 3*. She adds a new folder called **`PaintableBicycle`**. The rest of the code presented in this chapter can be found in the library project which is in **`chapter-3/BumbleBikesLibrary/PaintableBicycle`**.

For the latter, Phoebe devised an interface she thinks can describe anything her robots can paint:

```
public interface IPaintJob
{
  public string Name { get; set; }
  public int Cyan { get; set; }
  public int Magenta { get; set; }
  public int Yellow { get; set; }
  public int Black { get; set; }
  public IPrimer Primer { get; set; }
  public IPaintTopCoat TopCoat { get; set; }
  public void ApplyPrimer();
```

```
    public void ApplyPaint();
    public void ApplyTopCoat();
  }
```

Every paint job created in the system will have a marketable name, which is stored in the **`Name`** property. The paint system Kitty wants to use is based on the same CMYK color space used by traditional printing. She also intends to use a paint finish she calls **`TopCoat`**. Since this too is likely going to be reused later, it belongs in the library Kitty created in *Chapter 3*. You'll find this in the book's sample code in **`chapter-3/BumbleBikesLibrary/PaintableBicycle`**. The paint finish will make the bicycle's paint job take on a beautiful glossy sheen and protect the paint from scratches and sun exposure. Here's an interface that describes the paint finish:

```
public enum PaintTopCoatTypes { TopCoatClear, Gla
    TurboClear, HigherSolidClear }
public interface IPaintTopCoat
{
  public string Name { get; set; }
  public PaintTopCoatTypes Type { get; set; }
}
```

"Hold up, Kitty!" Phoebe exclaimed. *"Why are we using another enum? That was nothing but trouble last time!"*

Kitty defended her design decision by saying, “We were short-sighted thinking we could get away with one color per bicycle. But when it comes to primers and paint finishes, there are classifications for each that rarely ever change. The code for the primer is probably going to be reused, so Kitty puts the code in the library she created in *Chapter 3*. You can find it in **`chapter-3/BumbleBikesLibrary/PaintableBicycle`**. The **`PaintTopCoatTypes`** enum contains the same set that has been in use for several decades. The makers improve the chemistry in the products, but they never introduce anything revolutionary. The same goes for primer.” Kitty brought up her **`Primer`** interface:

```
public enum PrimerColors { Gray, White, Black }
public enum PrimerTypes {Epoxy, Urethane, Polyest
    AcidEtch, Enamel, Lacquer, MoistureCure}
public interface IPrimer
{
  public string ManufacturerStockKeepingUnit { ge
  public PrimerTypes Type { get; set; }
  public bool IsLowVoc { get; set; }
}
```

Naturally, a good paint job starts with a good primer. To expand on Kitty’s point, these primers have been around for a long time. Kitty put more in the enum than she’d likely use just to be careful. The girls

source a few they might use, but for the most part, the good old-fashioned gray primer used for automotive painting seems to work best. She has fields for the manufacturer's **Stock Keeping Unit** (**SKU**) so they can order the primer easily. She also has a property, `isLovVoc`, that tells her whether the paint is considered to be low in **Volatile Organic Compounds** (**VOCs**). If you've ever painted a room with regular house paint, you might remember the smell being very strong. You probably needed to open a window after a while to let the fumes out. When working with industrial paint, you must be very careful with fumes, the source of which is the VOCs. This property lets us be safe. If it's `false`, we'd want to be sure to wear a respirator around the painting equipment.

The structure of the paint job model is taking shape. Compared to a simple color term, it is relatively complex. As you can see, the Bridge pattern truly helps us. The main objective of the Bridge pattern is to allow two complex object structures to be developed and maintained independently of one another. The bicycle models have gone through many interactions by name and have become more and more robust models. Now, we have this complex paint job model. The two need to go together, but a day could come when either the bicycle model or the paint model might undergo drastic changes. The bridge is, as with many of the patterns we've studied so far, insulating parts of our code from change while simultaneously making all of our code flexible and reusable.

To handle our use case for the Bridge pattern, I'll make another interface that inherits from **`ISimplifiedBicycle`** called **`IPaintableBicycle`**. We're doing it this way to maintain as much flexibility as we can:

```
public interface IPaintableBicycle : ISimplifiedB
{
 IPaintJob PaintJob { get; set; }
}
```

Kitty and Phoebe have four bicycle designs that need to implement this interface. It makes sense to set up an abstract class to implement the interface:

```
public abstract class PaintableBicycle : IPaintab
{
  public string ModelName { get; set; }
  public int Year { get; }
  public string SerialNumber { get; }
  public BicycleGeometries Geometry { get; set; }
  public SuspensionTypes Suspension { get; set; }
  public ManufacturingStatus BuildStatus { get; s
  public IPaintJob PaintJob { get; set; }
```

The **`Build()`** method requires an update as well. The **`Bicycle`** class has a line that prints the color of the bicycle. We still need to do that, but instead of deriving it from the enum, we'll work with the name of the paint job:

```
public void Build()
 {
   Console.WriteLine($"Manufacturing a
       {Geometry.ToString()} frame...");
   BuildStatus = ManufacturingStatus.FrameManufa
   PrintBuildStatus();

   Console.WriteLine($"Painting the frame
                   {PaintJob.Name}");
```

We also need to apply the other parts of the paint job:

```
   PaintJob.ApplyPrimer();
   PaintJob.ApplyPaint();
   PaintJob.ApplyTopCoat();
   BuildStatus = ManufacturingStatus.Painted;
   PrintBuildStatus();
```

The rest of the class can remain the same. You will find the full code in the chapter's source code available on GitHub.

The new paint system exceeds the Kickstarter backers' expectations. Not only can Bumble Bikes support custom colors for every bicycle but they can also support custom gradient paint jobs. This is a feature rarely seen in the bicycle industry outside of shops specializing in custom paint and assembly.

The Bridge pattern can be implemented any time you have two or more class hierarchies that are complicated and need to be used together. At its core, the Bridge pattern is little more than composition. The reason it is recognized as a pattern is it becomes a thoughtful exercise in decoupling. It usually arises within the design phase. You start with a class and it gets more and more complex as you design it. Ideally, you find this growth in complexity when you are modeling the class, and you decouple it before it ever reaches code. In the real world, it comes up several iterations or even several years into the project, and you need to remember this has happened before. It will happen again. Eventually, it will happen to you. There is a solution, and that's the heart and soul of software development patterns.

Summary

Kitty and Phoebe's modeling of the bicycle products and the automated manufacturing process and apparatus is growing more sophisticated as the two learn and invent their new business. This is how it

works, and from the outside looking in, I'd say they're doing an amazing job. It is very normal for a software developer to be an expert at software design and development, but far less of an expert in their understanding of the business problems they are tasked to solve. Software projects are really an evolution involving the developer's understanding of the business and the needs of their customers and stakeholders.

In this chapter, we learned several structural patterns that allow us to continue to make our software more sophisticated yet simple to maintain and extend. You've also noticed a common theme: the basic tools of inheritance and composition afforded by an object-oriented programming language are not, by themselves, enough to build robust software. These structural patterns allow us to use the tools of composition and inheritance to maximum effect without boxing our designs into a quagmire of spaghetti-like object hierarchies.

The Decorator pattern allows us to extend existing classes by decorating or wrapping new functionality around the original class. The decorators can stack as a Russian Matryoshka doll does, where one doll is nested inside another.

Figure 4.16: A Russian Matryoshka doll (each doll nests inside the larger one, the same way decorators wrap around base classes or stack on other decorators).

Phoebe mastered the Façade pattern, which allows us to abstract and insulate our software from complex dependencies. The pattern

allows you to put a simple face on a complex API by uniformly exposing operations, even if they aren't uniform under the covers. You can also use a façade to only expose the elements in a complex API or structure that are important to your implementations. If in the future, the third-party API changes significantly, you can replace the façade without having tightly coupled API calls sprinkled throughout your code.

Kitty was able to model the bicycles' group sets in a tree-like structure that allowed her to easily use recursion to find the cost and weight of the components together or in combination. Any time you need to deal with a tree, you should think, as Kitty did, of the Composite pattern.

With the cost and weight analysis problem solved, the girls teamed up to deal with the market's demand for a wider color choice in the Bumble Bikes line-up. Initially, Kitty and Phoebe had decided to limit color selections to make the job of modeling the bicycles easier. However, their Kickstarter campaign indicated a high demand for more color choices. When Kitty tried to solve the problem using inheritance, she found herself faced with a huge number of subclasses. For each model of bicycle and each color, the number of classes grew out of control. Kitty solved the problem by modeling the two dimensions, the bicycle and the paint colors, independently using the Bridge pattern. They were even able to create a system capable of custom gradient

paint jobs, to the delight of their Kickstarter backers. Having applied the Bridge pattern, the bicycle and the paint systems can grow independently from each other yet remain related through composition.

At this point, we've covered two of the three groups of patterns. Creational patterns helped us with object instantiation. Structural patterns helped us with new ways of thinking about how we build complicated objects with more sophistication than simply using inheritance and composition. The last group, covered in the next chapter, is a set of patterns to help you design "well-behaved" classes with behavioral patterns.

Questions

1. Why is the Decorator pattern sometimes referred to as a wrapper?
2. How can you use a decorator to extend a sealed class?
3. Which pattern is most effective at decoupling complex object structures in a way that allows them to mature separately?
4. When is the best time to consider using the Façade pattern?
5. Which pattern allows you to leverage recursion and polymorphism with a tree-like structure?
6. Have you ever run into a situation where you had to violate SOLID principles? Can you think of any way to avoid Phoebe and Kitty's resolution when applying the Bridge pattern?

Further reading

- Martin, Robert C., James Newkirk, and Robert S. Koss. *Agile software development: principles, patterns, and practices*. Vol. 2. Upper Saddle River, NJ: Prentice Hall, 2003.
- Martin, Robert C., and Micah Martin. *Agile principles, patterns, and practices in C# (Robert C. Martin)*. Prentice Hall PTR, 2006.
- The companion website for this book is https://www.csharppatterns.dev. Go check in and see what's new.

5

Wrangling Problem Code by Applying Behavioral Patterns

Do you want to have some fun that doesn't involve code for once? Next time you find yourself in a tall building with an elevator, get with three or four friends and ride to the top floor. Here's the fun part: have everyone in your group face the back of the elevator. As other people get on the elevator, they will almost always follow your lead and face the rear. This is because human behavior follows patterns! There are entire fields of study devoted to this fact, including psychology, sociology, and the applied fields of marketing and human relations.

Software is a human invention, so it should come as no surprise that software can be made to follow behavioral patterns too. Behavioral patterns are patterns that deal with algorithms implemented within your classes and how those classes interact and share responsibilities for executing those algorithms.

As our story continues, Kitty and Phoebe will be facing challenges that will require them to learn and implement four of the most popular behavioral patterns:

- **The Command pattern**
- **The Iterator pattern**
- **The Observer pattern**
- **The Strategy pattern**

As you ride along with them on their journey, you will learn how to diagram and implement the four most popular behavioral patterns using the **C#** language.

Technical requirements

Throughout this book, I assume you know how to create new C# projects in your favorite **Integrated Development Environment** (**IDE**), so I won't spend any time on the mechanics of setting up and running projects in the chapters themselves. There is a short tutorial on the three most popular IDEs in *Appendix 1* of this book. Should you decide to follow along, you'll need the following:

- A computer running the **Windows** operating system. I'm using **Windows** 10. Since the projects are simple command-line projects, I'm pretty sure everything here would also work on a **Mac** or **Linux**, but I haven't tested the projects on those operating systems.
- A supported IDE such as **Visual Studio**, **JetBrains Rider**, or **Visual Studio Code** with C# extensions. I'm using **Rider** 2021.3.3.

- Some versions of the the **.NET SDK**. Again, the projects are simple enough that our code shouldn't be reliant on any particular version. I happen to be using the **.NET Core 6 SDK**, and my code's syntax may reflect that.

You can find the completed project files for this chapter on GitHub at https://github.com/Kpackt/Real-World-Implementation-of-C-Design-Patterns/tree/main/chapter-5.

Meanwhile, back at the bicycle factory

"W00t!" Phoebe exclaimed. Kitty was startled and amazed. Her sister had somehow managed to pronounce the hacker slang word while intimating excitement in such a way that Kitty could hear the zeros that replaced the phonetics of the double-o. *"What?"* Kitty inquired. Phoebe didn't answer right away, so Kitty looked up and found Phoebe dancing in circles. Phoebe was clad in dirty coveralls and had her hair pulled back in a ponytail. The sight of a dancing Phoebe was not unusual. Kitty realized there was a line of 10 robotic arms bolted to the floor. The arms were mimicking Phoebe's dance moves to the best of their limited ability.

The unchoreographed ballet of arms was unimpressive by the standards of the Vaganova academy. However, within the context of a hand-built robotic factory created by two fourth-year college students in an abandoned warehouse, it was an astonishing achievement. *"That's amazing!"* Kitty laughed. The sisters held hands and danced in circles, which almost caused a large collision as the robots attempted their emulation. They stopped dancing and laughed again. *"So, we're done?"* Kitty asked. *"No. Right now, they just mimic what they see,"* Phoebe replied. *"What do you mean by see? Vision was never a part of our specification,"* Kitty pointed out. *"I know,"* said Phoebe. *"I found one of Boomer's old Xbox's in a crate upstairs. It had a Kinect attached to it, so I dusted it off and connected it to our test client program."* Boomer is Kitty and Phoebe's cousin. He is a year older than Kitty and graduated from the University of Las Vegas last year. He went to college on an eSports scholarship and is now a pro. Naturally, he always had the best gaming gear.

Kitty remembered that Kinect was a camera system Microsoft sold years earlier as part of its Xbox console game platform. It provided a rudimentary computer vision system that was capable of recognizing the shape of a human body. There had been games based on the idea that you could move in front of the camera and the characters in your game would act according to how they "saw" you move. Naturally, being a Microsoft technology, the SDKs were readily available for C#.

“What a neat idea!” Kitty exclaimed. Phoebe breathlessly continued, *“So, we have a good object structure from our work with Creational patterns and we also have the skeleton of a working system in place because of our structural patterns.”* Kitty countered, *“True, but we have not done much with the actual command and control system that drives the robots. The robots can function on their own with our test programs. They can even make bicycle parts by themselves."* Phoebe finished Kitty’s thought in a way only sisters can do: *“What they can’t do is work together! I did some more reading last night. It turns out there is a whole collection of patterns that will enable the robots to work together. They are called Behavioral patterns! I think we can use them to control a lot of our systems and orchestrate the robots so that they may work together.”* Phoebe walked to the white-board and explained the next steps needed to code a control system for the necessary robotics.

The Command pattern

At this point, Phoebe has designed two different models of the robotic arm. One set of arms was large and bolted to the floor. These arms were stationary. The second set of arms was mounted on tracks and could move. These were mainly used for moving the parts, materials, and partially finished bicycles around to different stations. Once at a station, the larger arms would do most of the real work.

The larger arms that were bolted to the floor had interchangeable attachments that allowed them to perform different tasks. Phoebe designed this behavior based on the Decorator pattern. Remember, a decorator allows you to add new behavior to an existing class without you having to modify it directly. This is done by creating a new class that wraps around the structure of the original class, then adds the additional behavior. In this case, the decorators are physical hardware. Phoebe marveled at the patterns. She understood how working with the patterns might be considered a design philosophy as much as a software engineering practice. Phoebe remembered that patterns were devised by architects of the physical world at a time when software engineering was only done by scientists in top-secret military labs.

Each of the large stationary robotic arms could affix a different attachment for welding or another for buffing and polishing. Each arm could be programmed to take a handoff of materials brought by the smaller, mobile track-mounted robots.

Phoebe drew out her idea for Kitty, explaining how she could model a command for the necessary robotics in a way that was flexible.

"Think of it this way, Kitty," Phoebe began. *"When we order clothes and shoes online, we pick out the clothes we want to buy. We decide which dresses went want, as well as their color and size. Then, we tell*

the store where we'd like the clothes to be shipped. Finally, we give the retailer the payment details and a way to contact us if there are questions or problems. The order is a structure that holds all of that information. That's what I want to do here. I want a pattern where I can send a command. The command should contain everything that the robot needs to know so that it can do its job. The command shouldn't be tightly coupled to the robot's control API. In effect, I should be able to send a command to any piece of hardware we have. It isn't specific to any one robot, or even the attachment it's using at the time."

"I see," said Kitty. *"So, the order that contains all the information for the clothes we want to buy could just as easily be sent to any store in the world. We are the senders of the command. We compile the information and send it to a receiver. The receiver isn't unique to a single store and that receiver could be anything equipped to receive our command. We're packaging up everything needed to complete an order or a command and it's up to the receiver to act on it."*

"You've got it, sis!" Phoebe beamed. They were excited to get started. Phoebe brought up a diagram she had found online that showed the pattern in its generic form, as shown here:

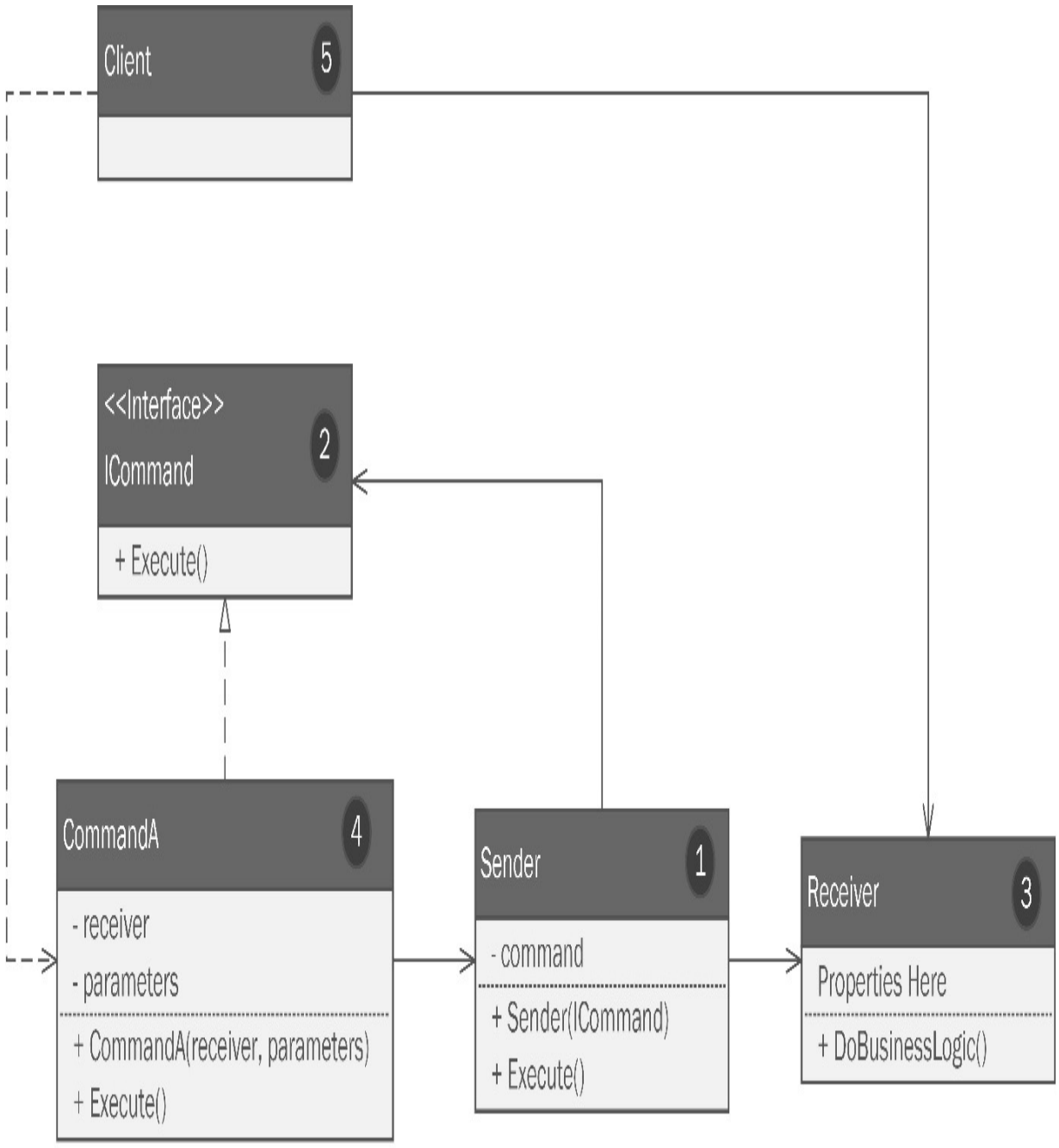


Figure 5.1 – The Command pattern.

Let's review the different parts of the pattern, which have been numbered appropriately:

1. A **`Sender`** object is responsible for invoking a request. For example, when Phoebe orders a dress online, the website is gathering the necessary information and is responsible for sending the order.
2. The **`ICommand`** interface defines a single method that's used to execute a command. The sender doesn't create the command. Instead, it receives it in the constructor, or it can be set as a property.
3. The **`Receiver`** class contains business logic and performs the actual work. The shop that receives the online order is the receiver when Phoebe orders online. The business logic might be different from store to store. Each might have a different process for the way the order is picked, pulled, settled, then shipped. The logic is up to the receiver. It receives the command that contains what it needs to perform the logic independently from the sender.
4. Concrete command classes implement the **`ICommand`** interface, but also contain a reference to the receiver that will execute the command, along with any properties or parameters needed to execute the command.
5. The client instantiates the concrete command class (**4**) and passes in or sets any parameters or properties needed by the command, including an instance of the receiver.

Applying the Command pattern

Phoebe drew her pattern idea while adapting what she had learned from the generic example. The following is her drawing:

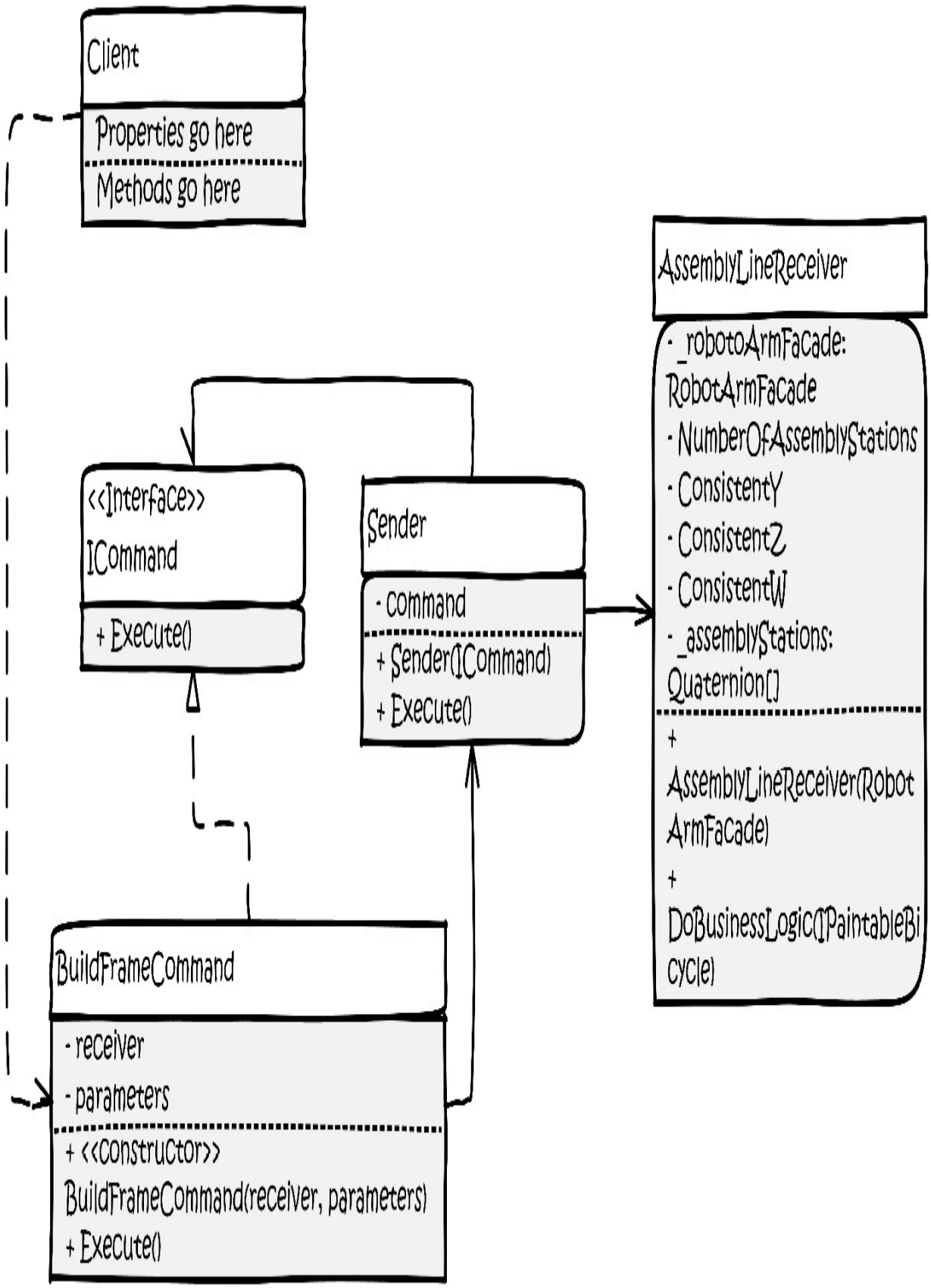

Client
Properties go here
Methods go here
AssemblyLineReceiver
- _robotoArmFacade: RobotArmFacade
- NumberOfAssemblyStations
- ConsistentY
- ConsistentZ
- ConsistentW
- _assemblyStations: Quaternion[]
+ AssemblyLineReceiver(RobotArmFacade)
+ DoBusinessLogic(IPaintableBicycle)
<<Interface>>
ICommand
+ Execute()
Sender
- command
+ Sender(ICommand)
+ Execute()
BuildFrameCommand
- receiver
- parameters
+ <<constructor>> BuildFrameCommand(receiver, parameters)
+ Execute()

Figure 5.2 – Phoebe's drawing of the Command pattern.

Kitty studied the diagram for a few minutes. Phoebe watched her closely. Kitty always got a look on her face before an idea clicked. After a minute, there it was. *"I got this!"* Kitty said as she walked to her corner of the warehouse and she started to work. After a few hours, she produced the implementation of the Command pattern. You can review this in the **`CommandExample`** project in this chapter's code, which can be found in this book's GitHub repository.

Coding the Command pattern

Kitty started by creating the **`ICommand`** interface:

```
public interface ICommand
{
    public void Execute();
}
```

Next, she made a concrete command that implements this interface:

```
public class BuildFrameCommand : ICommand
{
    private AssemblyLineReceiver _assemblyLineRec
```

Remember, a command object has everything a receiver needs to execute the intent, except the business logic. If you remember Phoebe's explanation, it even includes a **`private`** field to hold the receiver itself. A command is self-contained. In addition to the receiver, it needs the information on the bicycle that the command is responsible for building. Kitty added a project reference to **`BumbleBikesLibrary`**, which we covered in *Chapter 3, Getting Creative with Creational Patterns*. We extended the library to include the **`IPaintableBicycle`** interface that was created using the Bridge pattern in *Chapter 3, Getting Creative with Creational Patterns*. As a reminder, always use an interface wherever you can. Concrete objects should come into play as late as possible. This keeps your design flexible and honors **SOLID** principles:

```
    private IPaintableBicycle _bicycle;
```

Next comes a constructor where we pass in the receiver and **`IPaintableBicycle`**:

```
    public BuildFrameCommand(AssemblyLineReceiver
        assemblyLineReceiver, IPaintableBicycle b
    {
        _assemblyLineReceiver = assemblyLineRece
        _bicycle = bicycle;
```

```
        }
```

Lastly, we have the **`Execute()`** method, which is required by the **`ICommand`** interface. All it does is run the business logic contained within the receiver:

```
    public void Execute()
    {
        _assemblyLineReceiver.DoBusinessLogic(_bi
    }
}
```

The **`BuildFrameCommand`** class is done. Now, we need a sender. As mentioned previously, there isn't much going on in the sender. All it needs is a command, which Kitty specified as **`ICommand`**, and a method to execute the command, which Kitty has specified as **`DoCommand()`**. The **`DoCommand()`** method executes the command. This seems a little counterintuitive; you'd think the receiver executes the command. It does, but not directly. If you coded it to execute more directly, you'd likely be tightly coupling the sender, receiver, and command logic together, which is exactly what we want to avoid:

```
public class Sender
{
```

```
        private ICommand _command;
        public Sender(ICommand command)
        {
            _command = command;
        }
        public void DoCommand()
        {
            _command.Execute();
        }
    }
```

Kitty finishes up with the receiver, which she calls **`AssemblyLineReceiver`**. The idea behind this class is that it acts as a master control for the whole assembly line. This is a major part of the orchestration that the girls are trying to achieve.

The **`AssemblyLineReceiver`** class needs a project reference to the work we did with the Façade pattern in *Chapter 3, Getting Creative with Creational Patterns*. As you may recall, the Façade pattern allows us to present a myriad of complicated APIs as a single, easy-to-use point of contact. In this case, the APIs for the robotics came from the manufacturers. The microcontrollers for the different robot arm attachments (the welder, the buffer, and the grabber) all came from different manufacturers with different APIs. In *Chapter 3, Getting Creative with Creational Patterns*, Kitty and Phoebe wrote a Façade pattern to make this easier to use and to insulate them from repercussions inflicted by future changes to the third-party code, which will

evolve independently of the business requirements of Bumble Bike's manufacturing process.

The Façade pattern also contained an encapsulation of how the line of floor-mounted robots should work. The girls have opted to use **quaternions**, a `struct` common to the gaming industry, to represent the spatial layout of their assembly line. The quaternion struct can be found in the `System.Numerics` library of the .NET framework.

The first part of the `AssemblyLineReceiver` class just sets up what we need to use for the façade:

```
public class AssemblyLineReceiver

{
    private readonly RobotArmFacade _robotArmFaca
    private const int NumberOfAssemblyStations =
    private const float ConsistentY = 52.0f;
    private const float ConsistentZ = 128.0f;
    private const float ConsistentW = 90.0f;
    private readonly Quaternion[] _assemblyStatio
```

Quaternions are a complicated concept. I'm not selling short your intelligence; I'm paraphrasing the documentation for Unity 3D, a popular video game framework written in C#. In video game work with Unity, you can't swing a virtual cat without hitting a quaternion. The posi-

tion and angle of the virtual cat would be defined using a quaternion. The Unity documentation straight-up tells you that quaternions are an advanced mathematical concept. The short version is that a quaternion is a combination of three points in space represented as *X*, *Y*, and *Z*, along with a rotational vector represented as *W*. Since we don't need to get into the guts of quaternions in a book on patterns, we have simplified the layout of the assembly line by deciding it is a straight line. As such, only one coordinate in the quaternion is different from station to station, and for us, that is the *X* coordinate. The remainder can be held constant at a standard height (*Y*) and a standard depth (*Z*) within the confines of the factory floor. We will also hold the rotation constant at 90 degrees for the sake of simplicity. To summarize, to represent where a robot sits on the assembly line, we have four coordinates (*X*, *Y*, *Z*, and *W*) but we keep three constant. Only *X* varies as you move from one end of the assembly line to the other.

The location of each assembly for the stations is held in an array of quaternions. 10 arms can service 20 stations. In the receiver's constructor, we set up our façade, along with the locations of each station, by populating the **`_assemblyStations`** arrays with 20 quaternions:

```
public AssemblyLineReceiver(RobotArmFacade
    robotArmFacade)
{
    robotArmFacade = robotArmFacade;
```

```
        _robotArmFacade = robotArmFacade;
        _assemblyStations = new Quaternion
            [NumberOfAssemblyStations];

        for (var i = 0; i < NumberOfAssemblyStati

        {
            var xPosition = i * 25.0f;
            _assemblyStations[i] = new Quaternion
                (xPosition, ConsistentY, Consiste
                    ConsistentW);
        }
    }
```

That was a lot of setup, but it is also a nice example of how patterns can be combined. They are all pieces of the bigger puzzle. Next, we will look at the interesting part: the code that performs the business logic. This is going to look familiar.

The girls had this code in their test **`Program.cs`** file in the **`FacadeExample`** code we saw back in *Chapter 4, Fortify Your Code with Structural Patterns*. The logic itself isn't really important. In this case, it's a set of steps to move a bicycle frame from station to station as it is assembled and painted. The key takeaway is that this is where the actual business logic lives. It is kept separate from the sender. In our earlier example, store owners would not want Phoebe telling them how best to pick, pack, and ship the dress she selected. The

business logic is not driven by the sender. Likewise, we don't want the logic in the command itself. The command represents everything needed to perform the logic. It's the order, not the store workers, executing the order.

In Kitty's program, the **`DoBusinessLogic`** method takes the **`IPaintableBicycle`** object (essentially the order for the bicycle) and uses the façade to manipulate the robots to manufacture the bicycle:

```
    public void DoBusinessLogic(IPaintableBicycle
    {
        // Now let's follow an abbreviated imagi
        // assembly of a bicycle frame by contro
        // arm.
        // grabber gets the frame parts and takes
        // station 1

        _robotArmFacade.ActiveAttachment =
            ArmAttachments.Grabber;
        _robotArmFacade.MoveTo(_assemblyStations
        _robotArmFacade.Actuate();
        _robotArmFacade.MoveTo(_assemblyStations
… see the rest in the sample code.
```

Let's move on to the last part, which is the client.

Testing the Command pattern's code

Rather than showing you all of Kitty's client code, which is extensive and complex, I'll just show you her simple test program for the logic found in **`Program.cs`** in the sample code.

Remember, it is the client's job to create the command. In this example, Phoebe's job, as the customer, is to pick out a dress online, specify its color, and give her payment and shipping details. Here, the client is specifying what type of bicycle to build and draws on the Bridge pattern. It can also specify the paint job. Here, we're going with a simple black paint job that is standard for our mountain bike:

```
var blackPaintJob = new BlackPaintJob();
var standardMountainBike = new PaintableMountainB
    (blackPaintJob);
```

We need access to the control logic for the robot arm façade:

```
var robotArmFacade = new RobotArmFacade(new
    WelderAttachmentApi(), new BuffingApi(), new
        GrabbingApi());
```

We must create a command and pass in the data needed to complete the command:

```
var cmd = new BuildFrameCommand(new AssemblyLineR
    (robotArmFacade), standardMountainBike);
```

Finally, we must make a sender object and set everything in motion by calling the **`DoCommand`** method on the sender. As promised, the sender initiates the action, but it doesn't perform any action itself. When Phoebe chooses a dress online, she submits the order. It is the receiver that does the work:

```
var sender = new Sender(cmd);
sender.DoCommand();
```

The Command pattern is one of the most useful and popular of all the patterns we'll discuss. The *GoF* book, along with many others, discusses how it can be applied to the user interface layer of a desktop or web application. Commands within a UI can be created and sent to other parts of the program from several different parts of the UI. An easy example of this is when you save a file – you typically have a **File** menu option in your program. You likely also have a menu bar with a **Save** button, and a key combination such as *Ctrl*/*Command* +

S. Those are all senders. The Command pattern allows you to encapsulate the receiver logic in one place.

You can use the Command pattern any time you have logic to perform an action and you want to isolate it from tightly coupling to anything that may want to call that logic. The giving, receiving, and execution of commands in the real world is a pattern familiar to anyone who has been a parent giving a command and their child executing it. The command is initiated by a sender and executed by a receiver. The chain of events represents a pattern of behavior we all understand and recognize.

A full bicycle order would, in reality, comprise many of these commands. Here, we have built the frame and painted it. Kitty will build the rest of the logic needed to control the assembly process using commands, so we'll leave her to do that while we explore more patterns.

Next, we'll address another fundamental concept in C#: collections. While discussing collections, we'll focus on a pattern you have used many times maybe without even knowing it was a pattern: the Iterator pattern.

The Iterator pattern

Things were going well for Kitty as she worked on the control system, making liberal use of the Command pattern. Then, she hit something of a roadblock. Kitty's command receiver logic was just taking bicycle orders from the customer ordering systems, such as direct sales, and the dealerships. The requests were processed in the order they were received. Due to this, Kitty and Phoebe noticed a slowdown. While the algorithms in `AssemblyLineReceiver` were optimized to efficiently produce bicycles, Kitty failed to consider the painting process.

The costliest part of the painting process in terms of time and money was setting everything up to paint a bicycle. This was very easy earlier when each bicycle was only allowed to be built in one color. Now, Bumble Bikes supports custom paint jobs. The sisters were losing time and money on custom jobs because the paint equipment had to be thoroughly cleaned and reset when a custom order was placed. The paint system would have to mix the colors requested, apply them to the custom order, and then reset to a more common color such as red or black. This was happening many times per day, so when a custom job request was received, it held up all the other bicycles behind it. Phoebe pointed out that when her father had done printing work many years ago, he would group his jobs based on ink color requirements to minimize the number of times he had to clean and reink the press. The sisters needed to group their orders by paint job type so that they could do all the custom work in batches and only need to clean and reset once per day. They could have adjusted cus-

tomer expectations concerning delivery dates on custom paint requests, but that isn't an abnormal retail situation. Kitty needed to figure out how to group the paint orders efficiently and flexibly.

Kitty's first thought was to incorporate **Language Integrated Query** (**LINQ**), a feature unique to .NET languages that allows you to filter lists the way you may do with any query language such as **SQL** or **MongoDB Query Language** (**MQL**). Many developers like LINQ query syntax because it negates the need to create and manage a loop by working directly against a collection. Kitty did some research and discovered that, anecdotally, LINQ is likely between *2x* and *10x* slower than a traditional `foreach` loop. While it is neat, it could be a problem at the scale she hoped her software might one day require.

In C# work, `foreach` loops against collections and allows you to iterate over its elements – that is, you can process each item in the collection, one at a time. Collections in C# are strongly typed, meaning every object in the collection, such as a `List`, is of the same type. The incoming orders for bicycles were stored in a database and loaded into a `List<BicycleOrder>` in batches throughout the day. The code for the `BicycleOrder` class looks like this:

```
public class BicycleOrder
{
    public Customer Customer { get; set; }
```

```
    public IPaintableBicycle Bicycle { get; set;
    public BicycleOrder(Customer customer,
        IPaintableBicycle bicycle)

    {
        Customer = customer;
        Bicycle = bicycle;
    }
}
```

The standard iterator that's used by C# in the `foreach` loop is what returns the orders in the order they were added. This is to say, they operate **first-in, first-out** (**FIFO**). The first order you add to the list is the first one that the `foreach` loop processes. What Kitty needed was an iterator that gave her all the regular paint job orders first, and the custom paint jobs last. In short, she needed a custom iterator. Iterators in C# follow the Iterator pattern. This should hardly be surprising, though you may not have known it was a pattern at play. The Iterator pattern is shown graphically using UML in the following diagram:

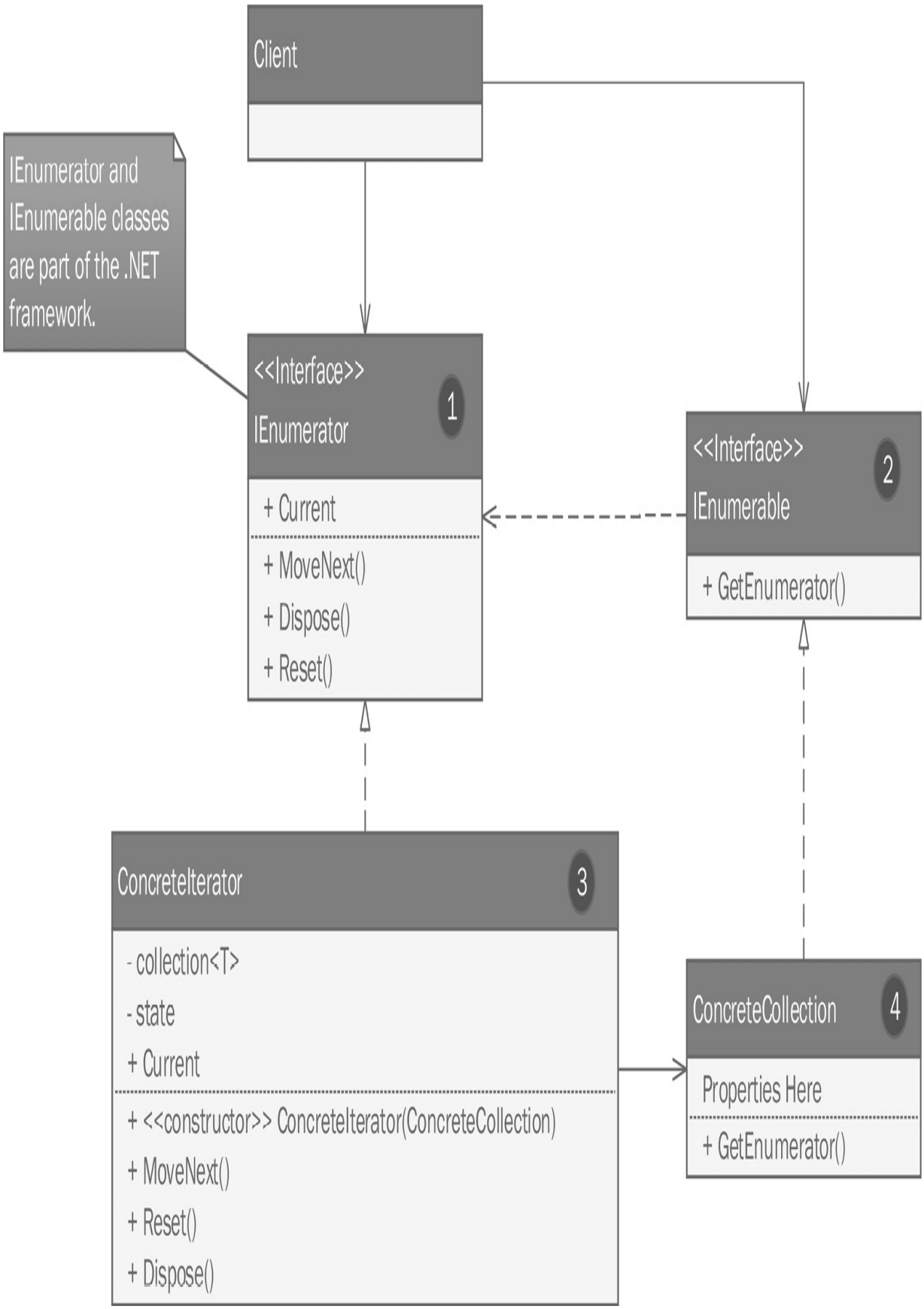

Client
IEnumerator and IEnumerable classes are part of the .NET framework.
<<Interface>>
IEnumerator
1
+ Current
+ MoveNext()
+ Dispose()
+ Reset()
<<Interface>>
IEnumerable
2
+ GetEnumerator()
ConcreteIterator
3
- collection<T>
- state
+ Current
+ <<constructor>> ConcreteIterator(ConcreteCollection)
+ MoveNext()
+ Reset()
+ Dispose()
ConcreteCollection
4
Properties Here
+ GetEnumerator()

Figure 5.3 – A custom iterator follows the Iterator pattern, which is partly already implemented for you in C#.

Let's review the different parts of the pattern, which have been numbered appropriately:

1. The **`IEnumerator`** interface is part of the .NET framework, so this time, you won't need to make it from scratch. The interface requires one property, **`Current`**, which returns the current element in the iteration. The required **`MoveNext()`** method is the mechanism used to advance the iteration to the next element.
2. **`IEnumerable`** is another interface that comes with C#, so again, you don't need to create it. It requires a method, **`GetEnumerator()`**, which is what provides an instance of our custom enumerator.
3. A concrete iterator that implements the **`IEnumerator`** interface will have the collection in a **`private`** field with the class. Collections, and by extension iterators, work using generics, as indicated by **`<T>`**. This means they can adapt to any type, including any classes you create. The class has a constructor that takes a concrete collection. By this, we mean the collection you intend to iterate that was implemented in **`ConcreteCollection`**.
4. A concrete collection.

As described, a standard iterator just gives you a way to work through a collection. Every collection in C# – and there are many – has a

common iterator that steps through the collection in order. Any time your business logic requires your iteration to be anything besides FIFO, it is best to encapsulate the algorithm in a custom iterator. We have reached one of those times. Maybe your iterator will have a novel algorithm for moving through the collection. A contrived example might be to have an iterator that only iterates on odd or even elements in the collection. Maybe you need an iterator that iterates strings in alphabetical order. Kitty's problem is a real-world one – she needs the collection to be filtered and sorted based on paint requirements before a normal iteration.

Applying the Iterator pattern

Let's look at how Kitty applied the pattern, as shown in the following diagram. Please note that **`IEnumerable`** and **`IEnumerator`** are part of the .NET framework and do not require actual implementation:

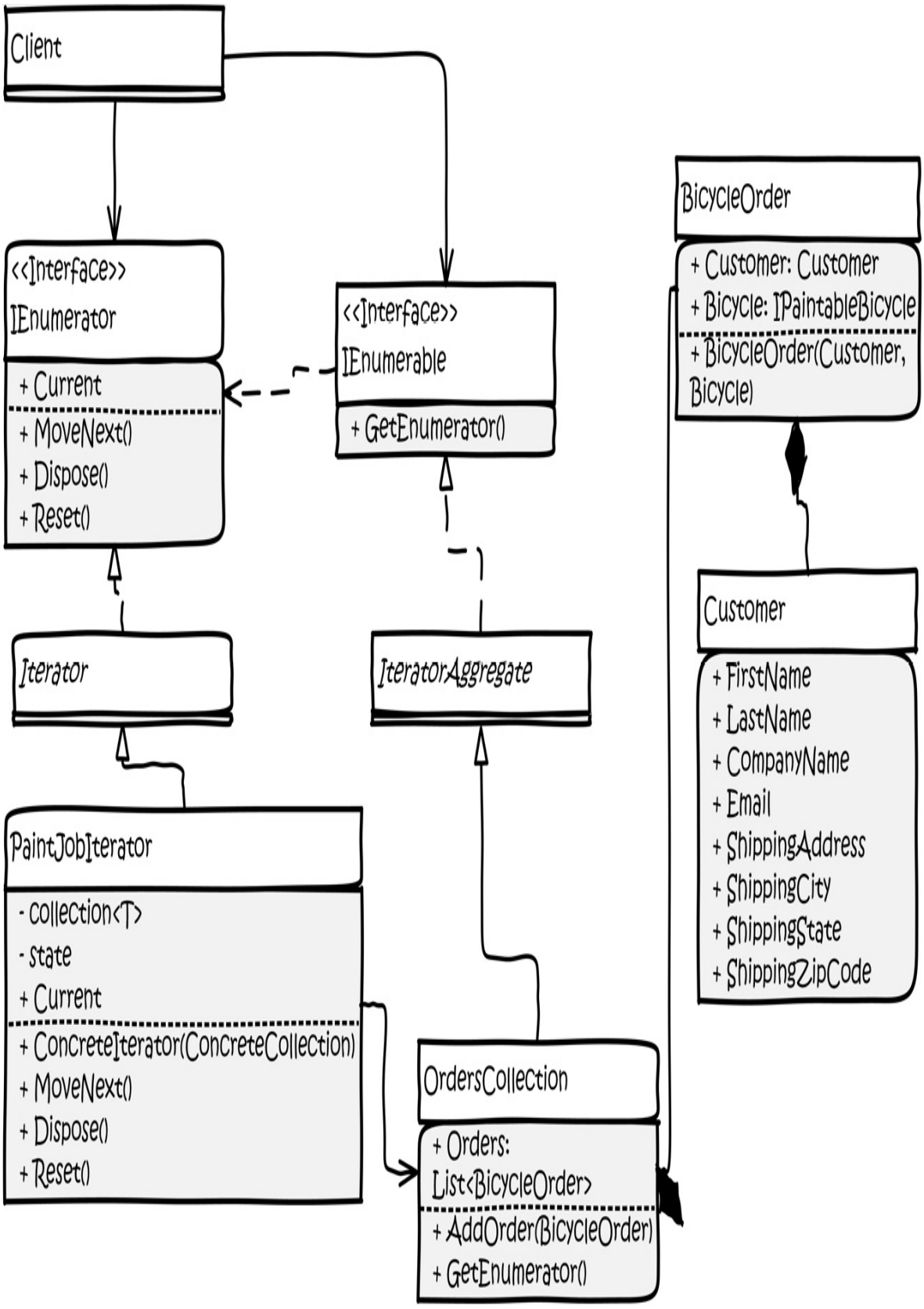

Client
<<Interface>>
IEnumerator
+ Current
+ MoveNext()
+ Dispose()
+ Reset()
<<Interface>>
IEnumerable
+ GetEnumerator()
BicycleOrder
+ Customer: Customer
+ Bicycle: IPaintableBicycle
+ BicycleOrder(Customer, Bicycle)
Customer
+ FirstName
+ LastName
+ CompanyName
+ Email
+ ShippingAddress
+ ShippingCity
+ ShippingState
+ ShippingZipCode
Iterator
IteratorAggregate
PaintJobIterator
- collection<T>
- state
+ Current
+ ConcreteIterator(ConcreteCollection)
+ MoveNext()
+ Dispose()
+ Reset()
OrdersCollection
+ Orders: List<BicycleOrder>
+ AddOrder(BicycleOrder)
+ GetEnumerator()

Figure 5.4 – Kitty's implementation of the Iterator pattern.

Iterators always have the same parts. Our concrete collection is called **`OrderCollection`**, which ultimately implements the C# **`IEnumerable`** interface via an abstract class called **`IteratorAggregate`**. The concrete iterator is a class called **`PaintJobIterator`**, which inherits from an abstract base class that will implement the requirements of the interface. The **`PaintJobIterator`** class contains the logic that will sort the collection based on the type of paint job. The custom jobs will be done last so that we can get our standard order bicycles shipped right away. Our customers are okay with waiting an extra day for their custom paint jobs, so those are done last.

The **`OrderCollection`** and **`PaintJobIterator`** classes inherit from the **`IteratorAggregate`** and **`Iterator`** base classes, respectively. These classes are just abstract classes that implement the respective interfaces. They are useful if you intend to create more than one custom iterator within your project.

Coding the Iterator pattern

We've seen the code for the **`BicycleOrder`** class already. This contains a reference to another class called **`Customer`**, which is pretty

much what you'd expect. Kitty used the **`MailAddress`** class from **`System.Net.Mail`**. Everything else is just strings:

```
public class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string CompanyName { get; set; }
    public MailAddress Email { get; set; }
    public string ShippingAddress { get; set; }
    public string ShippingCity { get; set; }
    public string ShippingState { get; set; }
    public string ShippingZipCode { get; set; }

}
```

Remember that the **`IEnumerator`** and **`IEnumerable`** interfaces are part of the .NET framework from **`System.Collections`**, so we don't code those. However, Kitty did opt to make abstract classes for them. The first is in the **`Iterator.cs`** file:

```
public abstract class Iterator : IEnumerator
{
    object IEnumerator.Current => Current();
    public abstract int Key();
```

```
        public abstract bool MoveNext();
        public abstract void Reset();
        protected abstract object Current();
    }
```

This is just an abstract class that implements the **`IEnumerator`** interface. Kitty has another for the **`IEnumerable`** interface in the **`IteratorAggregate.cs`** file. One method is required to comply with that interface:

```
public abstract class IteratorAggregate : IEnume
{
    public abstract IEnumerator GetEnumerator();
}
```

So far, this is just boilerplate code. Now, let's move on to the good parts. The first part is the customized collection Kitty called **`OrdersCollection`**. This is just a wrapper around a **`List<BicycleOrder>`**:

```
public class OrdersCollection : IteratorAggregate
{
    public List<BicycleOrder> Orders { get; set;
```

A simple parameterless constructor ensures we start with an empty list:

```
public OrdersCollection()
{
    Orders = new List<BicycleOrder>();
}
```

Here, we have a simple pass-through to the **`Add`** method of **`List`**. Pop quiz: does this look like another pattern we've covered already? Maybe the Decorator pattern?

```
public void AddOrder(BicycleOrder order)
{
    Orders.Add(order);
}
```

Here's the magical part. When we implement our collection, we need the **`GetEnumerator`** method to return our custom iterator, which we haven't written yet. We are overriding the abstract method in the abstract **`IteratorAggregate`** class mentioned earlier and we are returning **`PaintOrderIterator`**, as shown here:

```
public override IEnumerator GetEnumerator()
```

```
        {
            return new PaintOrderIterator(this);

        }
    }
```

Finally, we have the heart of this pattern – the actual iterator itself:

```
public class PaintOrderIterator : Iterator
{
```

Most of the contents in this class are implementations of the requirements expressed in the **`IEnumerator`** interface, which we express in the abstract **`Iterator`** class we saw earlier. First, we can see a **`private`** field that holds a reference to the **`OrdersCollection`** class we just saw:

```
    private readonly OrdersCollection _orders;
```

As the iterator moves through the collection, we need to track its position:

```
    private int _position;
```

A constructor takes in **`OrdersCollection`** and sets the **`private`** field, as well as the initial position. The initial position starts at **`-1`** because we haven't begun iterating yet and if we set it to **`0`**, we'd be indicating an actual position in the collection:

```
public PaintOrderIterator(OrdersCollection or
{
    _orders = SeparateCustomPaintJobOrders(or
    _position = -1;
}
```

We may need a way to get the private **`_position`** field. We'll use a read-only **`Key()`** method for this:

```
public override int Key()
{
    return _position;
}
```

The iterator is invoked via a **`foreach`** loop, which invokes the **`MoveNext()`** method to move the iteration forward until it reaches the end of the collection:

```
public override bool MoveNext()
```

```
    {
        var updatedPosition = _position + 1;
        if (updatedPosition < 0 || updatedPositic
            _orders.Orders.Count) return false;
        _position = updatedPosition;
        return true;
    }
```

The interface requires that have a way to reset the iterator position back to the beginning of the collection:

```
    public override void Reset()
    {
        _position = 0;
    }
```

`Current()` gives us the current iteration's object in the collection. Don't confuse this with the **`Key()`** method we saw a minute ago. **`Key()`** gives you the numeric index or the position, whereas **`Current()`** gives you the contents at the key's position:

```
    protected override object Current()
    {
        return _orders.Orders[_position];
    }
```

Here's our customization. All we're doing here is reordering the collection before the iteration takes place. Kitty made two lists. One will hold **`BicycleOrder`** objects, where the class in the **`PaintJob`** property is a standard paint job, while the other will hold the custom paint job orders. When we have separated them, it is simply a matter of recombining the list with the custom jobs at the end:

```
    private OrdersCollection SeparateCustomPaintJ
        (OrdersCollection orders)
    {
        var customPaintJobOrders = new List
            <BicycleOrder>();
        var standardPaintJobOrders = new List
            <BicycleOrder>();
        foreach (var order in orders.Orders)
        {
            var paintJob = order.Bicycle.PaintJob
```

If you remember when we built this during our coverage of the Bridge pattern, standard one-color paint jobs are expressed with the **`IPaintJob`** interface. Then again, so are custom paint jobs. Our custom paint job was defined in the **`CustomGradientPaintJob`** class, which is a different implementation of **`IPaintJob`**. Kitty used an abstract class in between the **`IPaintJob`** interface and the actual implementation. The intermediary class is called **`CustomGradientPaintJob`**,

which means we can detect the base class of the paint job and act accordingly:

```
            bool isCustom = paintJob.GetType().Is
                (typeof(CustomGradientPaintJob));
            if(isCustom)
            {
                customPaintJobOrders.Add(order);
            }
            else
            {
                standardPaintJobOrders.Add(order)
            }
        }
```

Now that we have a list of regular paint orders and a list of custom paint orders, we can replace the contents of the original **`orders`** list:

```
        orders.Orders.Clear();
```

Then, we can add the standard orders back in:

```
        orders.Orders.AddRange(standardPaintJobOr
```

This is followed by the custom paint orders:

```
            orders.Orders.AddRange(customPaintJobOrde
            return orders;
        }
    }
```

Done! Now, all Kitty needs is a quick program to test with in **`Program.cs`**. When I say *quick*, it's a few lines long because we need to create all the parts in the test program, including the customer, the bikes, the order list, and the custom iterator via a **`foreach`** loop. Remember that we're pulling in some classes from different packages. The **`Customer`** class uses **`System.Net.Mail`**, which is part of the .NET framework. The bicycles are going to be from **`BumbleBikesLibrary.PaintableBicycles`**, while the paint jobs are going to be from **`BumbleBikesLibrary.PaintableBicycle.CommonPaintJobs`**:

```
using System.Net.Mail;
using BumbleBikesLibrary.PaintableBicycle;
using BumbleBikesLibrary.PaintableBicycle.CommonP
using IteratorExample;
```

An iterator needs something to iterate, so let's make our empty **`OrdersCollection`** class:

```
var orders = new OrdersCollection();
```

Now, we need a customer. In real life, there would be several, but for our example, we'll just use one:

```
var dealership = new Customer
{
    FirstName = "John",
    LastName = "Galt",
    CompanyName = "Atlas Cycling",
    Email = new MailAddress("johngalt@whois.com")
    ShippingAddress = "123 Singleton Drive",
    ShippingCity = "Dallas",

    ShippingState = "Tx",
    ShippingZipCode = "75248"
};
```

Now, we need bicycles to put in the orders. Let's not waste time and make one with a custom paint job! This way, we know there's one at the front of the list. When the iterator reorders the list, all these should be at the end:

```
var amarilloPeacockPaintjob = new
     AmarilloPeacockPaintJob();
var bicycle0 = new PaintableMountainBike
    (amarilloPeacockPaintjob);
```

Once you have a bicycle with a paint job and a customer, you can make an order and add it to the **`orders`** list:

```
var order0 = new BicycleOrder(dealership, bicycle
orders.AddOrder(order0);
```

Next, let's do the same thing with some standard paint jobs that will wind up at the front of the list when we iterate. First, let's add a turquoise cruiser bike:

```
var turquoisePaintJob = new BluePaintJob();
var bicycle1 = new PaintableCruiser(turquoisePain
var order1 = new BicycleOrder(dealership, bicycle
orders.AddOrder(order1);
```

How about a white road bike?

```
var whitePaintJob = new WhitePaintJob();
```

```
var bicycle2 = new PaintableRoadBike(whitePaintJo

var order2 = new BicycleOrder(dealership, bicycle
orders.AddOrder(order2);
```

To keep things interesting, let's add another custom bike. This time, we'll add a recumbent model with a custom gradient paint job:

```
var bicycle3 = new PaintableRecumbent
    (amarilloPeacockPaintjob);
var order3 = new BicycleOrder(dealership, bicycle
orders.AddOrder(order3);
```

Here's a standard red road bike:

```
var redPaintJob = new RedPaintJob();
var bicycle4 = new PaintableRoadBike(redPaintJob)
var order4 = new BicycleOrder(dealership, bicycle
orders.AddOrder(order4);
```

That should be enough for a meaningful test.

Trying out the new iterator

Now, let's try out our iterator. If all goes well, this should look exactly like any of the iterations you see in C#:

```
foreach (BicycleOrder order in orders)
{
    Console.WriteLine(order.Bicycle.PaintJob.Name
}
```

The **`foreach`** loop is almost anti-climactic, isn't it? That's how we know we did a good job. You can't tell that our custom iterator is any different from any of the common iterators that ship with C# or .NET. The regular **`foreach`** loop contains the mechanism to pull out our custom iterator and use it to move through the collection via the **`MoveNext()`** and **`Current`** methods we have in our concrete classes. When Kitty runs the test program, she can see what we'd hoped for:

Figure 5.5 – Kitty's test run of her custom iterator.

As Kitty learned, the Iterator pattern is one of the most important patterns in our daily work. It isn't very complex, despite having a few different parts. You can use this pattern any time you need to process any kind of collection in a manner other than the standard FIFO processing order.

You should always be watching for chances to use this pattern. *Watching*… Hey, that reminds me of our next pattern!

The Observer pattern

Our worst fear is becoming a reality. Bumble Bikes has become so popular that Kitty and Phoebe are starting to have logistics problems. *"Don't get me wrong,"* Phoebe said. *"This is a good problem to have. We could be more profitable if we could optimize our shipping costs. The hardest part is the first mile. How can we be more efficient at getting our bikes to a national shipper's depot?"* Kitty arranged a **Zoom** call with *ExFed*, a small business owner who provides packing and shipping support as a service. Cathy, the *ExFed* representative near Phoebe's factory in Dallas, and John, the representative based in Alpine, where Kitty's factory was located, listened carefully to Bumble Bikes' predicament.

"The key to a good logistics workflow," Cathy said, *"is to make sure every time a truck leaves your factory, it's full of bikes. When the truck comes back, it should be full of raw materials for the next batch of bikes."* The girls had not considered the second part of what Cathy had explained. John concurred. The four worked out some details on the call after Phoebe had explained how her automated factory worked.

"You've already got a signal going to your raw materials supplier that tells them when you consume the materials to make a bike," John pointed out. Cathy picked up on his train of thought and completed it. *"Right – all we'd need to do is get a signal to our systems that lets us know when you have a truckload of bicycles for us to pick up. Our*

trucks can take the bicycles to the national shipper's depot where they ship out like any other freight."

Phoebe said, *"No problem! We'll just add another decorator class to our bicycle object."* John and Cathy stared blankly at their respective cameras. *"Phoebe! They don't speak nerd as we do!"* Kitty chided. *"Besides, that won't work. The raw material usage is reported for every bicycle. We don't want to send a pickup signal for every bicycle. That would be inefficient."*

"It would," John said. *"If you did that, you'd risk us leaving with a half-loaded truck. Based on the dimensions of your packaging, we need at least 10 bicycles in every load to make our service cost-effective."*

"What we need," Kitty began, *"is something that sends a signal when there are at least 10 bicycles ready to be picked up."*

"Right," said Cathy. *"You need to have someone at the end of the assembly line count the bicycles and when they get 10, click a button on our website. After that, it will take about 30 minutes to get the truck to your dock."*

Phoebe rolled her eyes imperceptibly. Her sister smirked. Think about all those *normal* people in the world who don't speak nerd, and don't understand software automation. What difficult and unfulfilled

lives they must lead! Phoebe and Kitty knew they were going to find a way to automate their supply requests.

"But during that delay," continued Phoebe, *"we've probably made another 5 to 10 bicycles. Is that a problem?"*

"No," said John. *"We need just a minimum of 10."*

John and Cathy dropped off the call with a list of to-dos and a virtual handshake deal to handle Bumble Bikes' first-mile logistics. Kitty and Phoebe stayed on Zoom and continued brainstorming. *"Cathy said we needed someone to count bicycles as they come off the assembly line. I don't want to pay someone to just sit around and observe."*

"That's it! Kitty, you're a genius!" Phoebe exclaimed. Kitty beamed quizzically. She wasn't sure what she'd done to merit this rare moment of sisterly praise.

Phoebe realized this situation was calling for the **Observer pattern**. Cathy had envisioned a person observing the process and reacting when the bicycle count reached a minimum of 10 bicycles. To automate this, the girls will have to write software that can *observe* the production process and generate a signal to the logistics company when the requisite bicycle inventory has been reached. A generic diagram of the Observer pattern is as follows:

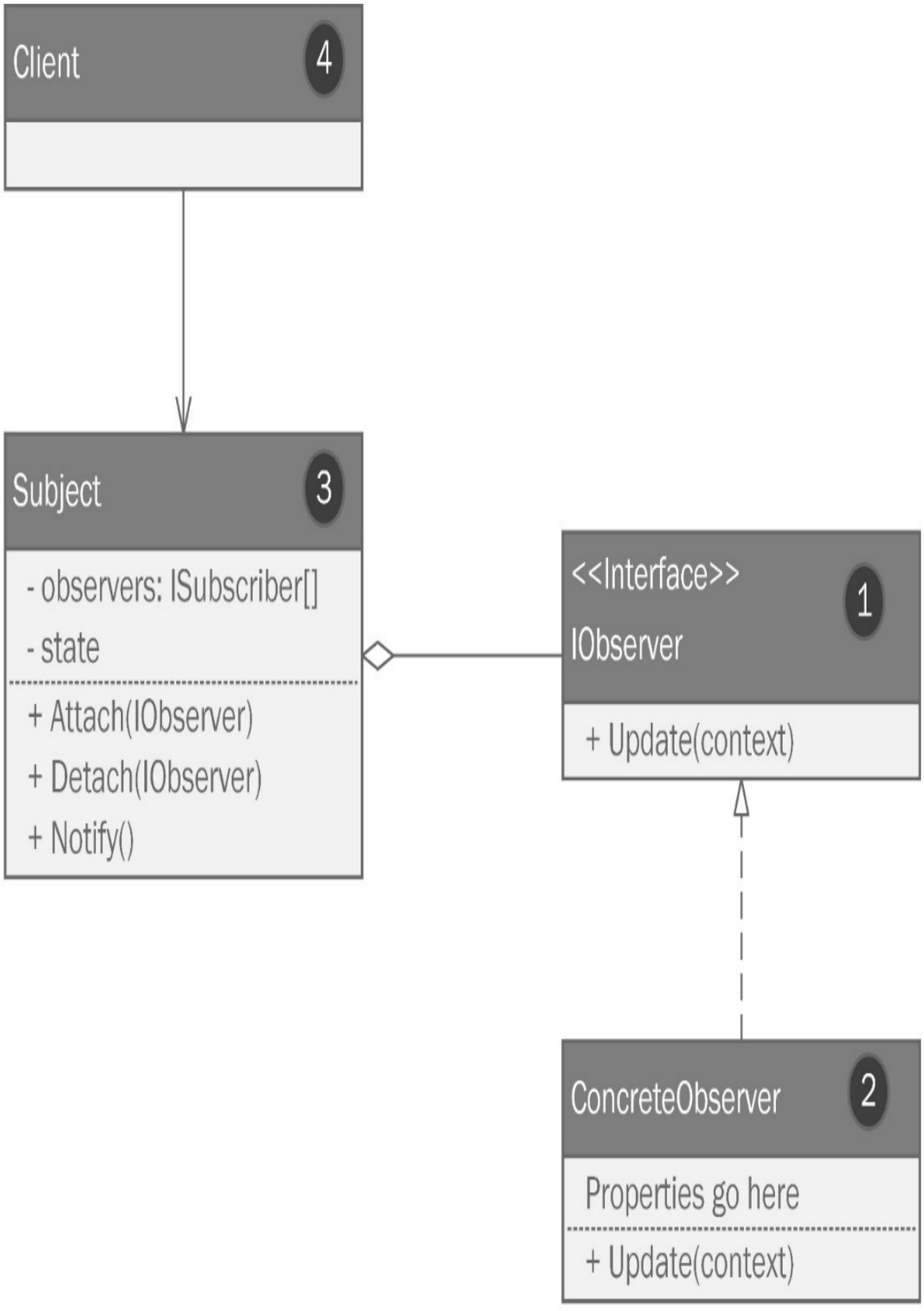

Client
4
Subject
3
- observers: ISubscriber[]
- state
+ Attach(IObserver)
+ Detach(IObserver)
+ Notify()
<<Interface>>
IObserver
1
+ Update(context)
ConcreteObserver
2
Properties go here
+ Update(context)

Figure 5.6 – The Observer pattern

There are two basic parts to the Observer pattern – a **subject** and one or more **observers**. Let's review the different parts of the pattern, which have been numbered appropriately:

1. An interface describes the method requirement of the observer. This interface defines a public `Update()` method. `Update()` is called whenever the observer "sees" something interesting happen in the object it is observing.
2. Our concrete observer contains the implementation logic for the behavior that happens when the observer "sees" some interesting change in the subject's state.
3. The subject is doing some useful piece of work and maintains its state while the observer waits. Note that the observers are contained inside a collection within the subject. Also, note that the `state` property is `private`. We'll need some way to let the observers know something interesting happened. For this, we have a function called `Notify()`. When a triggering condition takes place, the `Notify()` method can iterate over each attached observer and call its `Update()` method.
4. The whole process is invoked by some larger program we'll call *the client*.

Applying the Observer pattern

Kitty brings up Zoom's whiteboard and they collaborate on their implementation of the Observer pattern that will solve this problem, as shown in the following diagram:

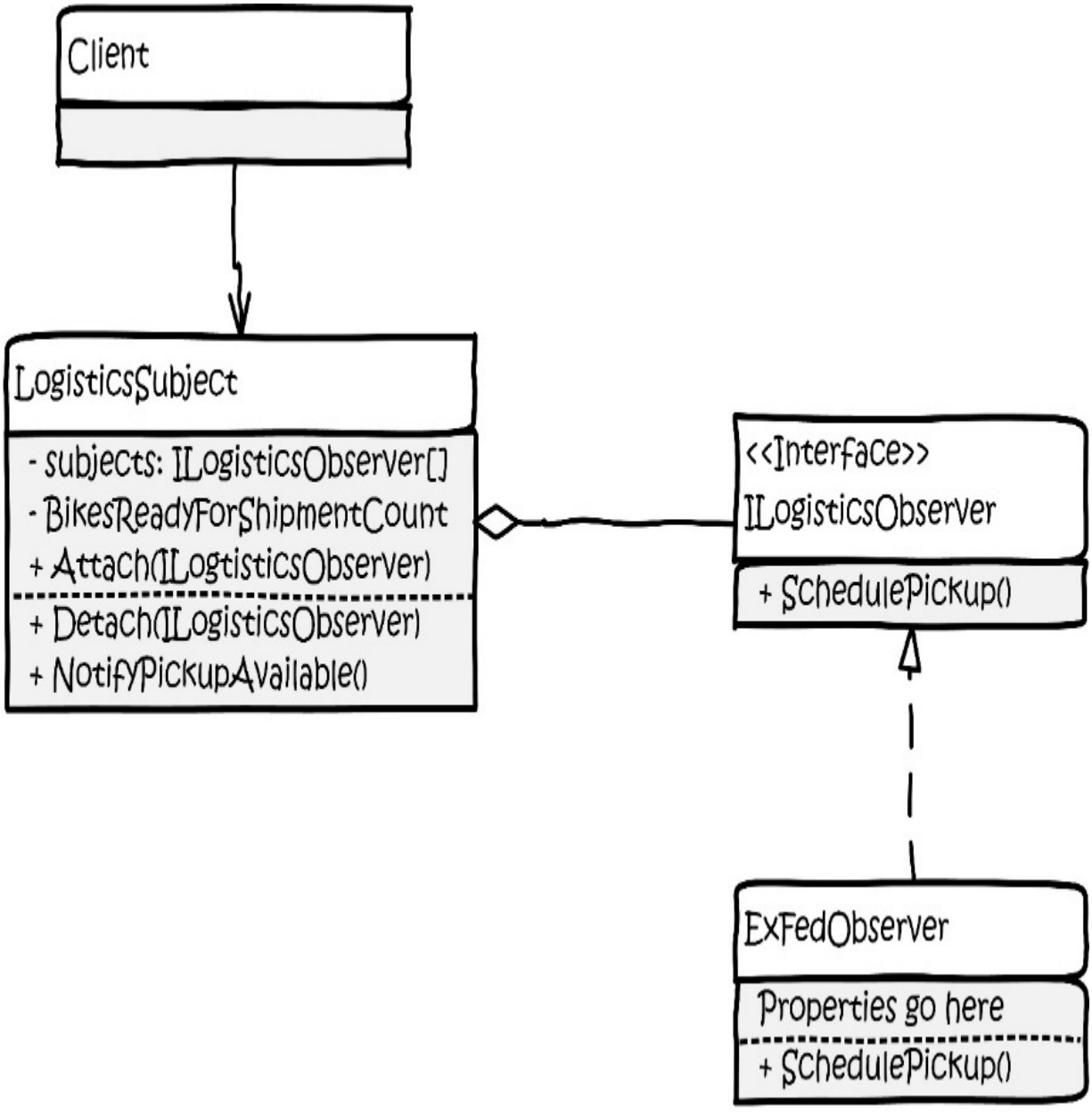


Figure 5.7 – Phoebe and Kitty's implementation of the Observer pattern that can signal ExFed's trucks to pick up an order of

bicycles.

This one is pretty simple – we need a subject and an observer. The girls used an interface, **`ILogisticsObserver`**, to prevent tight coupling between the **`LogisticsSubject`** class and the concrete observer class called **`ExFedObserver`**.

Once they had drawn the diagram, they each opened their favorite IDE, which contains a collaborative coding feature. This means the girls can code together as though they were sitting next to each other.

Coding the Observer pattern

Phoebe makes short work of the **`ILogisticsObserver`** interface:

```
public interface ILogisticsObserver
{
    public void SchedulePickup();
}
```

Kitty follows up with a concrete observer that consumes Phoebe's interface:

```
public class ExFedObserver : ILogisticsObserver
{
```

```
        public void SchedulePickup()
        {
            Console.WriteLine("ExFed has been notifie
                shipment is ready for pick up.");
        }
    }
```

I left out the actual API call because Kitty and Phoebe were worried that I might accidentally reveal their API key. As we all know, only amateurs check API keys into GitHub, and the code won't work without it, so I put it in a **`Console.WriteLine`** statement as a substitute.

The **`LogisticsSubject`** class is where the real action lies. It's a little longer, so the girls work on it together:

```
    public class LogisticsSubject
    {
```

Kitty adds a **`List<ILogisticsObserver>`** field to hold all the observers. She follows that with a typical constructor, which initializes the field:

```
        private readonly List<ILogisticsObserver>
            _logisticsObservers;
        public LogisticsSubject()
        {
```

```
{
    _logisticsObservers = new
        List<ILogisticsObserver>();
}
```

Phoebe adds an **`Attach`** method that allows us to add one or more observers, which are objects that conform to the **`ILogisticObserver`** interface:

```
public void Attach(ILogisticsObserver observe
{
    _logisticsObservers.Add(observer);
    PrintObserversCount();
}
```

Likewise, she also adds a method that can remove the observer from the list:

```
public void Detach(ILogisticsObserver observe
{
    _logisticsObservers.Remove(observer);
    PrintObserversCount();
}
```

Kitty realizes their testing will be easier if they have a way to see the observers in the list, so she adds a quick method to provide some output for the initial runs. The method simply prints the number of observers stored within the private **`_logisticsObservers`** field:

```
    private void PrintObserversCount()
    {
        switch (_logisticsObservers.Count)
        {
            case < 1:
                Console.WriteLine("There are no
                    observers.");
                break;
            case 1:
                Console.WriteLine("There is 1 obs
                break;
            default:
                Console.WriteLine("There are " +
                    _logisticsObservers.Count + "
                        observers.");
                break;
        }
    }
```

Lastly, we need to notify our observers. The generic UML diagram specified a **`Notify()`** method. We called ours

`NotifyPickupAvailable()`. It simply iterates through the observers and calls the **`SchedulePickup()`** method on each one in the list:

```
    public void NotifyPickupAvailable()
    {
        foreach (var observer in _logisticsObserv
        {
            observer.SchedulePickup();
        }
    }
}
```

Following her sister's lead, Phoebe writes the test program in **`Program.cs`**. First, she creates an instance of **`LogisticsSubject`**:

```
var logisticsSubject = new LogisticsSubject();
```

Then, she makes the observer and attaches it to the subject:

```
var exFed = new ExFedObserver();
logisticsObserver.Attach(exFed);
```

Next, let's simulate making 100 bikes. Each time we have 10, we'll send a notification:

```
var pickupOrder = new List<Bicycle>();
for (var i = 0; i < 99; i++)
{
    var bike = new MountainBike();
```

Here, Phoebe is just simulating a passage of time in an attempt to keep it a real simulation. She wishes her robots could build a bike in 3 seconds. After the delay, she writes out the bike with the **`ToString`** method and adds the bike to the **`pickupOrder`** list:

```
    Thread.Sleep(3000);
    Console.WriteLine(bike.ToString());
    pickupOrder.Add(bike);
```

Our observer logic checks whether we have enough bikes. If so, it triggers the **`NotifyPickupAvailable()`** method, which loops through all the observers and calls their **`SchedulePickup()`** methods:

```
    if (pickupOrder.Count > 9)
    {
        logisticsSubject.NotifyPickupAvailable();
```

In the real world, 30 minutes would pass before a truck arrived at Bumble Bikes to take their inventory. However, no one wants to simu-

late that, so we'll simply pretend we did and clear out the order:

```
            pickupOrder.Clear();
        }
    }
```

When we are done making bikes for the day, we can detach. Phoebe understands the importance of work-life balance for her factory robots:

```
    logisticsSubject.Detach(exFed);
```

As the girls' software operation saw more and more patterns in use, the number of problems left to solve was dwindling rapidly. They were moving into the phase you'll see in every software project, where the work shifts from developing new code to maintaining it. Usually, about this time, the senior developers start to get bored because all the big problems have been solved. Those developers must now make a choice: stick with the project or find another project with new challenges. That's what happens in most software shops. Surely Phoebe, being the brilliant owner of a bicycle manufacturing startup, would never succumb to the temptations fostered by monotony?

The Strategy pattern

"I'm BORED!" Phoebe yelled across the room at her sister. Kitty had come up from Alpine to go over some spreadsheets with Lexi, the head of accounting for Bumble Bikes. Phoebe and Lexi had been friends for many years, so when Phoebe had the chance to recruit her, she took it. Lexi, who was used to Phoebe's peculiarities, smiled at Kitty, folded her laptop, and said, *"I'll have this done for you tomorrow."*

As Lexi left the office, Phoebe flopped upside down on the couch, flipping through channels on the TV with the sound off. She wound up on channel 52,381, which was *The Bike Channel*. Bumble Bikes advertised heavily on this channel, and at that moment, a talking head was reviewing bike computers. A **bike computer** is an electronic device that reports your speed and distance traveled. Fancy models can keep track of your cadence, which is the pace at which you pedal. Some even track your heart rate and the electrical wattage produced by the effort a rider imposes on the pedals.

Phoebe's face was starting to turn red. She had been upside down too long and blood had rushed to her head. She flopped the rest of the way over and revealed that look she got when she had a million-dollar idea.

Her sister could see it without even looking at her. *"What?"* Kitty asked.

Phoebe stared at the ceiling a little longer, her eyes flitting back and forth. She was inventing something in her mind. Kitty looked up and could see the wheels turning in Phoebe's head.

"Bike computers!" Phoebe yelled at last. *"Why are they so boring? I mean really! All they do is tell jocks how great they are. Who needs a computer for that?"* It was an odd statement, but coming from Phoebe, it rated at most a 4 on a scale from 1 being something such as a pizza order to 10, which might be a non-sequitur completion of an engineering problem she'd given up on privately a week prior.

"What if a bike computer did something cool, like what we have with our car computers? Sure, it would be track speed, distance, wattage, and whatever jocks pay for in a bike computer. but It would also have navigation routes, trail, and road conditions. This is information a rider would want to know and have so that they can complete their next epic ride," Phoebe blurted. This was usually the part where Phoebe ran to her lab and ordered a stack of pizzas and fizzy water. Then, she would disappear for a few days. The sister's recent success gave the girls the ability to move their labs to their respective factories. Phoebe's lab had a bathroom and a bed that stretched between the far wall and what looked like a *Van de Graaff* generator. *"Why does she need that?"* Kitty thought silently. She'd long since given up asking out loud. The bed, covered in pizza boxes from previous engineering adventures, appeared to have never been slept in.

Kitty didn't respond to her sister's question concerning the use of a bike computer. She knew it was a question that wasn't meant to be answered. When Kitty came in the next morning, she noticed a stack of discarded pizza boxes and empty cans of fizzy water littered throughout the office. A small black box was mounted to a handlebar assembly on Phoebe's workbench. Phoebe was asleep in her lab.

Curious, Kitty fiddled with the buttons on the box's apparent interface. Phoebe had made a small computer with some sensors attached. Kitty was able to find the navigation feature that had captured Phoebe's imagination. It was crudely designed, but functional. *"Hey, sis,"* Phoebe muttered sleepily. *"It's not working yet. I'm stuck on the navigation."* Phoebe's head flopped back down on the pillow. Kitty continued fiddling with the navigation. She saw the problem: the UI on the navigation allowed you to pick from several kinds of terrains. You could search for routes on paved roads, gravel trails, or mountain trails. However, the search results consistently showed only paved roads. It was easy to figure out why.

Phoebe was leveraging well-known GPS APIs to compute her route. Naturally, these were favoring paved roads. Phoebe was able to create a façade for the API and a decorator that slightly altered the default behavior so that the API stayed away from recommending busy highways, even when they were the most direct route.

For the next few hours, Kitty researched alternative mapping APIs that would focus more on the roads less traveled and gave preference to those inaccessible to cars. She found that each time she added different APIs and pathfinding algorithms, her code started to become complex. She could easily see the beginnings of a big ball of mud on her growing plate of spaghetti.

After some refactoring and some thinking, she settled upon a strategy that should work. Spoiler alert: *strategy* is the name of the pattern she used. This pattern is easy to explain because the pattern means the same thing as the word *pattern* means in plain English. If you're trying to achieve a complicated objective, you create a strategy. In software engineering, the Strategy pattern refers to working with a set of algorithms flexibly and interchangeably. An *algorithm* is simply a set of steps you can follow to solve a problem that gives a consistent result within a reasonable amount of time. If you were to create an algorithm to create a peanut butter and jelly sandwich, the steps would be simple:

1. Put two slices of bread on a plate.
2. Open the peanut butter.
3. Using a dull knife, spread peanut butter on one side of one slice of bread.
4. Close the peanut butter jar.
5. Open the jelly.

6. Using a different dull knife (don't you hate it when someone uses the same knife and gets peanut butter in your jelly?), spread some jelly on one side of the second slide of bread.
7. Close the jelly jar.
8. Place the first slice of bread on top of the second so that the peanut butter and jelly meet between the slices of bread.

That's an algorithm. If you follow that algorithm, I can guarantee that you'll always wind up with a peanut butter and jelly sandwich. This algorithm can be completed in a few minutes, which to me, is a reasonable amount of time. I could easily make a second algorithm to make a turkey sandwich, and another to make a cheese sandwich. If I encapsulate each algorithm using a common interface, I can choose a sandwich-making strategy based on what kind of sandwich I desire.

Kitty has a situation where she needs an algorithm to get from point A to point B. She needs three different strategies, which are coded as three different algorithms. The first algorithm will find a path strictly on paved roads. The second will try to make use of gravel or unpaved roads. The third will find a path with no roads at all, but that is otherwise passable on a bicycle. She needs these routes from A to B to follow a common interface, and she needs the algorithms themselves to follow a common interface so that she can swap them as appropriate.

We can express this with UML, as shown in the following diagram:

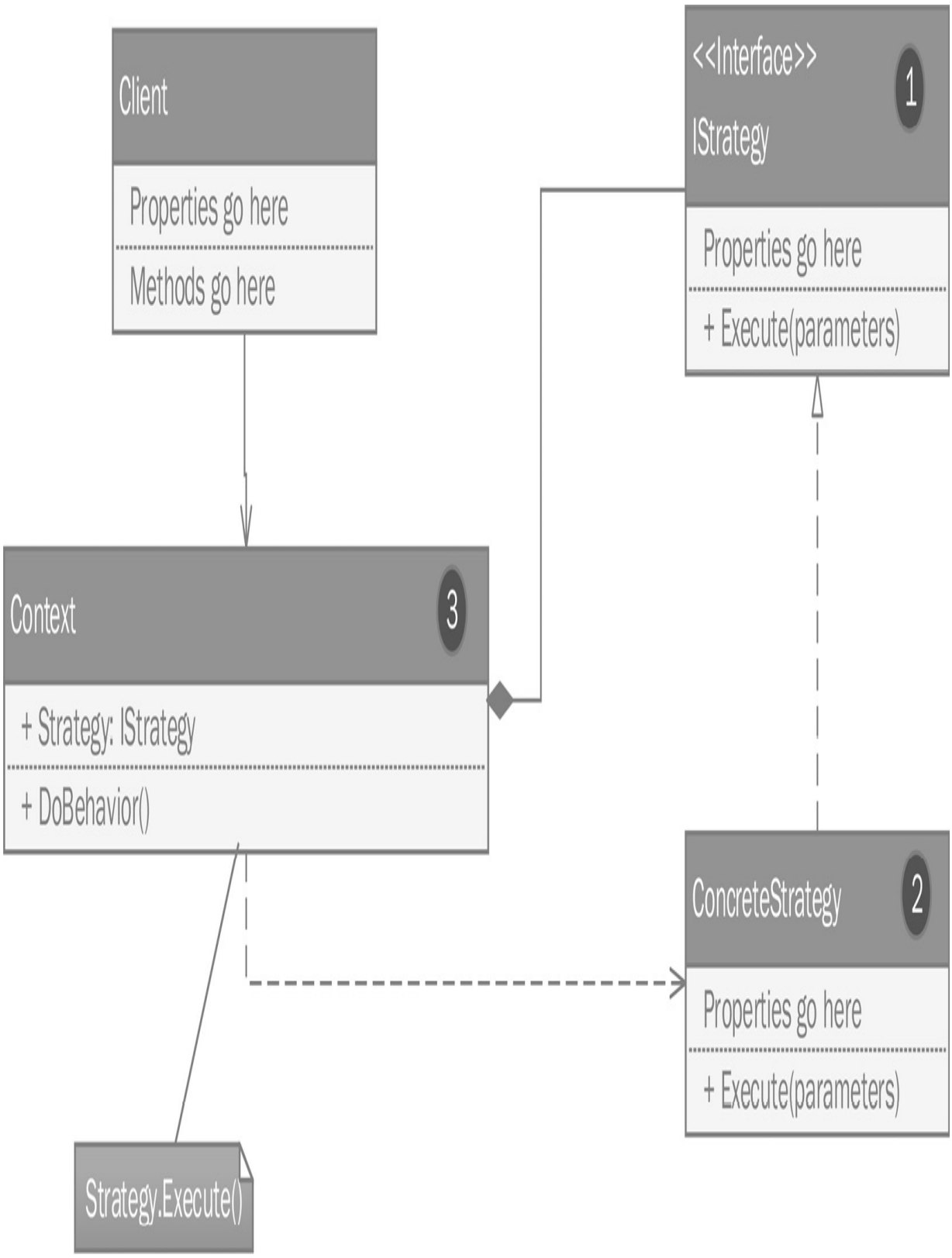


Figure 5.8 – The Strategy pattern.

Let's review the different parts of the pattern, which have been numbered appropriately:

1. The **`IStrategy`** interface defines a method that implements your algorithm. In our case, it will be called **`Run`**, and we can pass in any data the algorithm may need, such as the geospatial coordinates of your starting and ending locations.
2. A concrete strategy object that implements the **`IStrategy`** interface will implement your algorithm.
3. A context object holds the strategy and can execute it with some method. We have called ours **`DoBehavior()`**.

The key here is that all the algorithms follow the **`IStrategy`** interface. This means I can pass any algorithm contained in a concrete strategy object containing a **`Run`** method to invoke the algorithm. At this point, the algorithms are interchangeable.

Applying the Strategy pattern

Kitty comes up with a diagram for her implementation, as shown here:

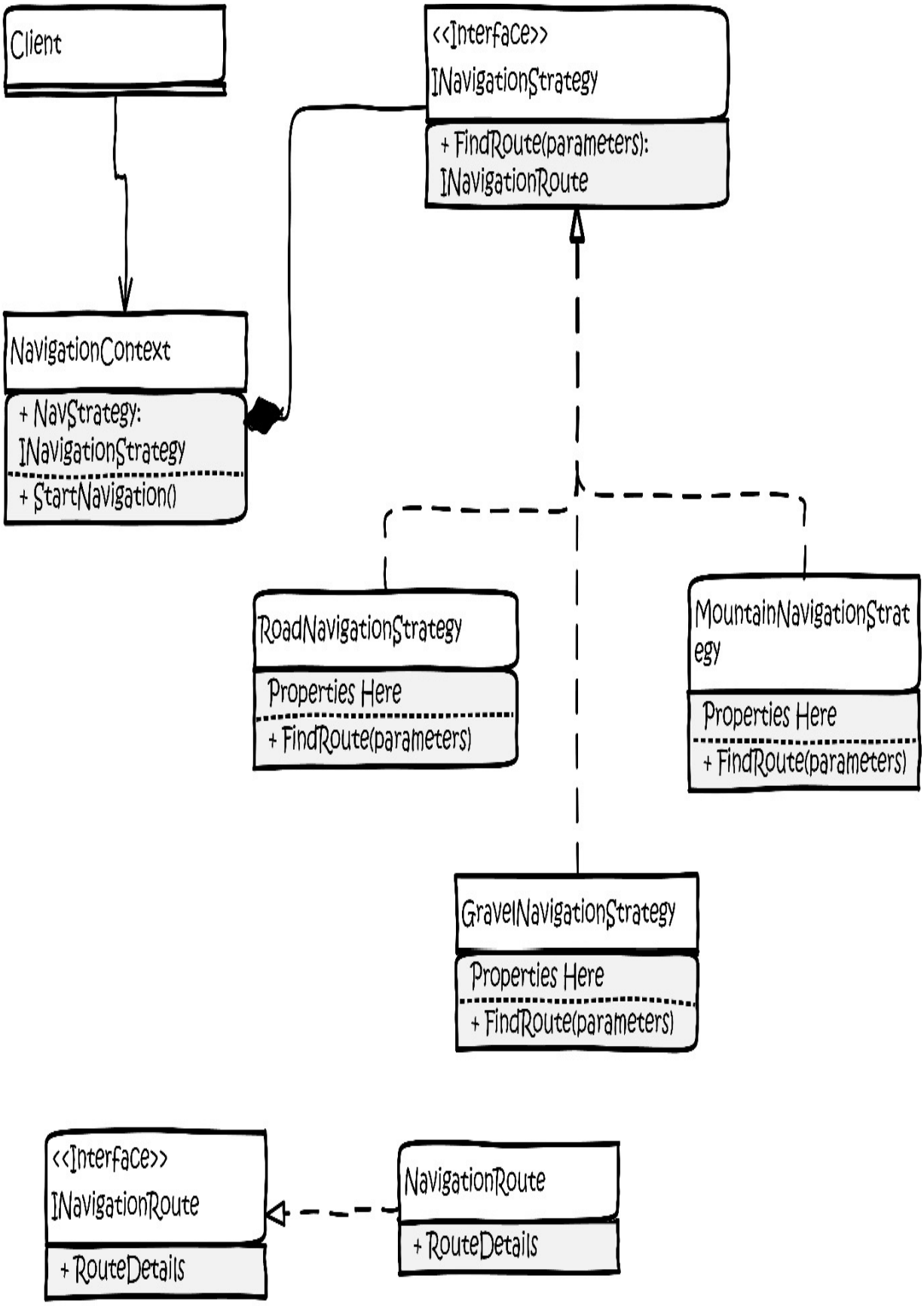

Client
<<Interface>>
INavigationStrategy
+ FindRoute(parameters):
INavigationRoute
NavigationContext
+ NavStrategy:
INavigationStrategy
+ StartNavigation()
RoadNavigationStrategy
Properties Here
+ FindRoute(parameters)
MountainNavigationStrat
egy
Properties Here
+ FindRoute(parameters)
GravelNavigationStrategy
Properties Here
+ FindRoute(parameters)
<<Interface>>
INavigationRoute
+ RouteDetails
NavigationRoute
+ RouteDetails

Figure 5.9 – Kitty's drawing of their implementation of the Strategy pattern.

Kitty looked over at Phoebe who was in a deep sleep. Maybe with the Strategy pattern, she could fix up this software before she wakes up.

Coding the Strategy pattern

Phoebe already had the data structures at the bottom of the diagram. She had typed **`INavigationRoute`**:

```
public interface INavigationRoute
{
    public string RouteDetails { get; set; }
}
```

She also had a concrete **`NavigationRoute`** class. Unfortunately, Karina, the attorney representing Bumble Bikes, will not allow me to show this part of the code. That's okay. We're here for the pattern. Neither the interface nor the class is part of the pattern. They're just structures used in the implementation. In place of Kitty's highly proprietary data structure, I'll give you a simple string:

```
public class NavigationRoute : INavigationRoute
{
```

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

Let's move on to the code that is part of the strategy. We'll start with the **`INavigationStrategy`** interface. This interface is used to conform your algorithms to a common structure so that they can become interchangeable:

```
public interface InavigationStrategy
{
    public InavigationRoute FindRoute(string para
}
```

As mentioned previously, Kitty needs three concrete implementations. The first is for finding routes on paved roads. Phoebe had this one working, so Kitty simply refactored it to fit the interface:

```
public class RoadNavigationStrategy : INavigation
{
    public INavigationRoute FindRoute(string para
    {
        // This is where your amazing algorithm g
        // since this is a book on patterns and n
        // algorithms...
        return new NavigationRoute
```

```
            {
                RouteDetails = "I'm a road route."
            };
        }
    }
```

Then, Kitty created an algorithm to find gravel road routes:

```
public class GravelNavigationStrategy : INavigati
{
    public INavigationRoute FindRoute(string para
    {
        // This is where your amazing algorithm g
        // since this is a book on patterns and n
        // algorithms...
        return new NavigationRoute
        {
            RouteDetails = "I'm a gravel route."

        };
    }
}
```

At this point, I predict you will not be surprised by the third implementation:

```
public class MountainNavigationStrategy :
    INavigationStrategy
{
    public INavigationRoute FindRoute(string para
    {
        // This is where your amazing algorithm g
        // since this is a book on patterns and n
        // algorithms...
        return new NavigationRoute
        {
            RouteDetails = "I'm a mountain route.
        };
    }

}
```

The only thing left is the **`NavigationContext`** class:

```
public class NavigationContext
{
```

Kitty used a simple property to hold her navigation strategy. Naturally, she declared the interface, not a concrete object:

```
public INavigationStrategy NavigationStrategy
    set; }
```

Next is a standard constructor. She set the default to a road navigation strategy since that is the default for Phoebe's device:

```
public NavigationContext()
{
    NavigationStrategy = new RoadNavigationSt
}
```

Finally, we have a method to start the algorithms on finding the path we're looking for based on the current strategy in the **`NavigationStrategy`** property:

```
public void StartNavigation()
{
```

There was a lot of cool business logic here. Eventually, Kitty generates the route using the strategy:

```
        var route = NavigationStrategy.FindRoute
```

```
            ("parameters go here");

        Console.WriteLine(route.RouteDetails);
    }
}
```

Honestly, this is the most innovative set of algorithms I have ever seen. Too bad the lawyers got involved. However, we did get to see the pattern, which turned out to be very simple compared to some of the more complicated patterns we've seen.

The Strategy pattern is used when you need to be able to choose from a set of related algorithms all aimed at a common objective. Whether it's making sandwiches or devising a novel set of interchangeable geospatial pathfinding algorithms, using a Strategy pattern will help keep your code maintainable and easy to read. Kitty can easily add more algorithms as she thinks of them, without breaking any of the existing strategies.

Summary

Behavioral patterns work with algorithms in ways that keep your software manageable. In this chapter, we looked at four very useful and popular patterns that can be employed to solve a variety of design problems.

The Command pattern can be used to isolate instructions from the objects responsible for executing them. This is one of the most common causes of the antipatterns we discussed in *Chapter 1, There's a Big Ball of Mud on Your Plate of Spaghetti*. Tightly coupling logic with concrete structures yields software that is brittle and prone to grow in complexity. The Command pattern will help you avoid this trap.

The Iterator pattern is used any time you need to iterate over a collection in some manner not handled by the standard .NET iterator. This pattern works with a collection and starts with the first item before iterating in a straight line to the last. This can take the form of manipulating the collection before processing, or it might be a novel way of moving through the collection to meet a business requirement. Some of the basic building blocks for this pattern are built into the .NET framework within the **`System.Collections`** namespace. Before building an iterator, you should check if one already exists.

The Observer pattern consists of a subject and one or more observers. The subject notifies the observers of a particular trigger condition within the subject's state. The Observer pattern is widely used with a myriad of applications. Many software developers who have used event listeners have seen and understand the power of this technique.

The last pattern we discovered was the Strategy pattern. We use the Strategy pattern any time we have a set of algorithms with a common purpose. The Strategy pattern works by conforming the algorithms to a common interface that is injected into a context. Then, the algorithms can be used interchangeably as business needs require.

In the next chapter, a fateful turn of events will force Kitty and Phoebe to enlist the help of a stranger. Bumble Bikes began as a passion project and turned into a business venture, but it will quickly become a humanitarian outreach. Too few software developers, engineers, and architects understand they have a superpower that can change the world. The stakes are high, and Kitty and Phoebe are stretched too thin. They need someone who understands SOLID principles and patterns to head up a very important project. Will this understated stranger be up to the task? Would you be?

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which pattern is used to make algorithms with a common purpose interchangeable within a context?
2. Which pattern is used to encapsulate and send instructions to a receiver, while avoiding tight coupling between the data needed to

execute the instruction and the logic that executes the instruction?

3. Which pattern involves a *subject* and an *observer*? Please note that if you miss this question, it will go on your permanent record.
4. Which pattern is used to process collections in a way other than FIFO?
5. What two interfaces from the .NET framework are useful when implementing the Iterator pattern?

CAUTION

Part 3: Designing New Projects Using Patterns

Having learned about some patterns, let's take a look at the design process. So far, we've been making things up as we go along. How much easier would things be, and how many problems could we avoid, if we took a step back first and designed our project with patterns and UML instead of diving into the code? This section works through a new project from this perspective. We will design the new project purely as a set of diagrams first in *Chapter 6, Step Away from the IDE! Designing with Patterns Before you Code*. Then, we will implement the project in *Chapter 7, Nothing Left but the Typing: Implementing the Wheelchair Project*. The final chapter wraps up the book and aims to show you there are more patterns out there. In fact, they're everywhere! There are development patterns beyond the usual Gang of Four sets and even more beyond the realm of **Object-Oriented Programming** (**OOP**). You'll even learn the documentation process for creating and publishing your very own patterns! I've included an appendix at the end to review the basic concepts in case you're new to C#, OOP, or UML.

This part covers the following chapters:

- *Chapter 6, Step Away from the IDE! Designing with Patterns Before You Code*
- *Chapter 7, Nothing Left but the Typing: Implementing the Wheelchair Project*
- *Chapter 8, Now You Know Some Patterns. What Next?*
- *Appendix 1, A Brief Review of OOP Principles in C#*
- *Appendix 2, A Primer on the Unified Modeling Language*

6

Step Away from the IDE! Designing with Patterns Before You Code

The main focus of this book is to learn the most popular and widely used **Gang of Four** (**GoF**) **patterns**. The secondary focus of this book is to consider using those patterns with real-world applications, with realistic and changing business requirements. Too many books on complicated topics such as patterns are full of abstract, contrived, or trivial examples. You have learned some popular patterns, but they do not benefit from your coding if you can't apply them. Over the years, I've developed a process that works for me, and I hope it works for you too. My process eases you into patterns one step at a time as you consider a new design.

This might be a disturbing revelation, but this chapter will not present any code samples. We'll be following the practice of diagramming our code before we start writing it.

We'll cover the following topics in this chapter:

- We'll watch as the new software architect for Bumble Bikes creates a UML design for the new project using a two-pass method.
- The new architect will make the first pass by diagramming the classes and basic structures for the project.
- The architect will then make a second pass, wherein they will identify the appropriate patterns and update the diagram as needed.

Our story of the two sisters continues in this chapter. A huge change in the business requirements at Bumble Bikes is about to occur. We'll see how to adapt to and overcome the inherent obstacles presented by these changes using patterns.

Technical requirements

Since this chapter contains diagrams rather than code, my usual spiel about **integrated development environments** (**IDEs**) doesn't apply. This time, we need a different set of tools. There are many tools for drawing UML diagrams. In the real world, a diagramming exercise often happens on a whiteboard. At several points throughout this chapter, I stop and challenge you to a diagramming exercise.

If you'd like to take up the gauntlet, you'll need some way to create and work with UML diagrams. If you've never drawn a UML class diagram before, there is a brief tutorial in *Appendix 2* of this book. A

whiteboard is fine for ephemeral drawings, but for this book, my diagrams need to be a bit more permanent, so here's what I'm using:

- A computer running the **Windows** operating system. I'm using **Windows** 10. Honestly, this doesn't matter since diagramming tools are plentiful for all operating systems, and there are many that work in your browser.
- A diagramming tool. I'm using **Microsoft Visio**. There are quite a few UML tools on the market. Here is a short list of tools I've used over the years:
- **Microsoft Visio**
- **StarUML**
- **Altova UModel**
- **Dia**
- **Umbrello**
- **UMLet**
- **OmniGraffle** (**Mac** only)

You can find the completed project files for this chapter on GitHub at https://github.com/Kpackt/Real-World-Implementation-of-C-Design-Patterns/tree/main/chapter-6.

As our story continues, we see a curve ball that threatens to unravel all the work the sisters have accomplished. In the real world, business requirements change all the time. Sometimes they are small

changes and they do not really affect the project. Other times, the changes are radical. The sisters are about to face a personal crisis. How should they respond? Should they take the quick and easy path and throw some stovepipe patches in place? I think by now, we've learned that patches lead to a big ball of mud on a large plate of spaghetti! However, if we use a disciplined and considerate approach, we might be able to avoid that kind of devolution.

Let's see what happens.

A bad day at the agency

Tom's phone rang at 3:58 on a Friday afternoon at his agency. The phone was on his desk, and he had been talking with his colleagues in another cubicle. Their development team had been working on a software project for a client. The first few releases had gone off without a hitch, and the customers were delighted. The customers were so happy that they even sent over a boatload of new feature requests and signed a contract extension. It was shortly after the contract extension that things started to turn ugly. Tom's development team was way behind schedule. Implementing the last round of feature requests had left their client's product unstable and it was crashing often. Tom was currently in a code review meeting. The group had concluded the project's code was in dire trouble. Their management, in

an effort to steal a quick win and impress their new client, had ordered the team to ship the first prototype. The next few releases came quickly. The client's impression of the agency was that they were staffed entirely by miracle workers. However, beneath that façade (not the pattern this time), they were having trouble finding a way to extend and fix the code without a full rewrite. They found they had created a big ball of mud.

It took Tom a minute to maneuver his wheelchair around the cubicle to get back to his desk and answer the ringing phone. It was his boss' boss, Stephanie. "*That's odd. She never calls me,*" Tom thought. "Also, isn't she on vacation in Antigua?" She wasn't supposed to come back until next week.

Puzzled, Tom answered his phone and Stephanie started right in with her thick accent. "Tahm! We need to tawk." It was a short conversation. It ended abruptly, along with Tom's employment with the agency. He'd been fired. He was in shock. Everything had been going so well until, suddenly, it wasn't. Stephanie was notoriously mercurial and often punished her employees at the slightest hint of any dissatisfaction from her coven of supervisors. She was on the fast track to the top and she wasn't about to let a bunch of nerds derail her trajectory.

Tom remembered one time, on a different project, he had devised a method to save time and money on production by automating a diffi-

cult and error-prone process. It worked. The project went through **quality assurance** (**QA**) and all the way to production. The customers had no complaints with the release and Tom's new methods would save a lot of time on future projects, thus making those projects more profitable. Stephanie hadn't shown interest in any of Tom's work until she heard about his process changes while eating at one of the management team's power lunches. Stephanie was livid. She hated the changes so much that she berated Tom and made him do the whole project over again. She pulled the release from production and, in front of the company owners, blamed Tom for the release being late.

Tom had hoped the success of the current project had placed him back in management's good graces. Apparently, it had not. Tom sat there stupefied for a few minutes, and then he heard Travis' phone ring. Travis, a hot-shot C# developer, had the cube next to Tom's. Travis disliked Stephanie so much he had assigned a ring tone to her number on his phone. It was the chorus to the song *Evil Woman*, by the band Electric Light Orchestra (ELO). When he heard the song's signature guitar riff, he knew what was happening. It wasn't just Tom being let go—it was the whole team. One after another, the phones rang, and after a short, demoralizing one-sided conversation, a heavy awkwardness settled like a thick fog.

Suddenly, the team's bullpen area was inundated with the agency's security team who were demanding that the developers box their personal effects. The developers were ordered not to touch a computer, not even a keyboard. Travis helped Tom with his boxes as he loaded them into the back of Tom's specially configured Toyota Sienna. Tom had gotten the van tricked out with special equipment that allowed him to drive, something his doctors had told him he would never be able to do on his own. The steering mechanism was his own design. Tom wondered how he would continue making the payments on the van now that he was unemployed.

For the next few days, Tom looked around on all the job boards, hoping to find a new job. His disability made this difficult. He was confined to a wheelchair and had limited use of his hands. Tom typed his code with his feet using a special keyboard. It was mounted on a platform to his wheelchair, just below his ankles. Tom also had a speech impediment that made him difficult to understand. Travis and his other colleagues had adapted to his voice, so for them, it was no problem. They understood him as if there were nothing wrong. Tom was realizing that wherever he went, he was going to have to start over and it wasn't going to be easy. As Tom mulled this over, he realized he was late for his volunteer shift. Once a month, Tom visited patients at a Dallas hospital who had recently suffered injuries that left them unable to walk. He helped them adjust and showed them very practical tips for getting around in a wheelchair. Tom smiled. Maybe he

could put in more time with his charity work while he looked for a new job.

Bumble Bikes factory – Dallas, Texas

It was midnight on a dark and stormy night. Phoebe was wide awake. While she had slept during the day, her sister, Kitty, finished the code for Phoebe's revolutionary bicycle computer. Phoebe felt a mixture of jealousy and pride as she eyed the results of her sister's successful implementation of the Strategy pattern. Now it was Kitty's turn to crash. She stretched out on the breakroom couch at Bumble Bikes. Phoebe's phone rang. It was her mother. It wasn't unusual for her mother to call late at night. Phoebe picked up the phone and her expression stiffened. "KITTY, WAKE UP! DADDY IS IN THE HOSPITAL!" she shouted to her sister.

Kitty, half asleep, let her sister drive the Jeep to the hospital. She hadn't realized the vehicle was capable of such speed. Kitty was wide awake from white-knuckled terror by the time they completed the short trip up Interstate Highway 75 to the hospital.

They found their mother, Karina, outside the hospital room. "The doctor is with him now," she explained. "What happened?" Phoebe

asked. The girls could hear the fear in their mother's voice. Their father was strong. He rarely got sick, and he never complained about the common aches and pains everyone gets from time to time. None of the three women had considered the possibility they would ever need to take him to the hospital.

"He's been tired for the last few days. This last weekend, he took one of your new road bikes to ride in the *Cow Creek Classic*," Karina explained. The girls knew this was a bicycle rally for charities supported by the local Rotary Club in a nearby small town. "He's also been complaining about a rash on the top of his head. We thought it was an infected tick bite or something. This morning when he woke up, he couldn't lift his head off his pillow. He couldn't sit up, and he was in tremendous pain. He just started crying and couldn't stop. I called an ambulance, and they brought us here."

Several hours passed. Various doctors and nurses entered the room and left. Some brought equipment, while some left the room with vials of blood. Finally, the attending physician came out of the room. He was tall, slender, and spoke with a thick Eastern European accent, He was wearing a black leather jacket where one might expect a lab coat. Phoebe couldn't help but think he reminded her of every Bond villain she'd ever seen in a movie. When he spoke to Karina, his English was fast and muddy, but his words were precise in their effect. "Your husband has dermatomyositis. It's a rare autoimmune disease

usually caused by a virus. We'll need a surgeon to biopsy his legs to confirm my diagnosis." Kitty asked, "What does that mean?"

The doctor went on to explain that the symptoms of the disease begin with a virus. "The body produces an immune response tailored to attack the virus. Once the virus is gone, however, the immune response mistakes healthy tissue for the infection and destroys it. In the case of dermatomyositis, the disease attacks the capillaries, which are small hair-like blood vessels. Every time you take a breath, your lungs add oxygen to your blood. This blood is carried through the arteries along with nutrients to the rest of your body, including your muscles. Your muscles use the oxygen and the nutrients, then expel waste materials that are taken by your veins back through your liver and kidneys. There the waste from the cells is filtered out and blood goes back to the heart, then to your lungs where the process starts over. Dermatomyositis attacks the tiny capillary blood vessels, which cuts off your muscles from the rest of your circulatory system. Basically, your muscles begin to die. When that happens, the body initiates an inflammatory response, which causes a great deal of pain. That's why your father couldn't stop crying." The doctor continued. "Think about your worst sore muscle pain. We can measure that in your blood by testing for a chemical called creatine kinase (CK). A day or two after your hard workout, it would be normal to see your CK levels at 150 units. If you were a Dallas Cowboys football player at training camp, they

might go as high as 300 units. Your father's CK level is at 10,000 units. We've put him on steroids and painkillers to make him comfortable."

"*What is the treatment*?" Phoebe asked. The doctor answered grimly: "*Dermatomyositis has no cure. We can try chemotherapy to reduce the immune response that's hurting him and destroying his muscles.*" Kitty's face dropped and she almost started to cry. This was the problem with having a mind that is always two steps ahead. "*What happens if it reaches his heart or the muscles that he uses to breathe?*" she asked. The doctor's expression was stolid. "*The disease won't affect his heart since it's made from a different kind of muscle tissue. But his diaphragm is a concern. It will take quite a while before that begins to happen. Let's run some tests and try some treatments to slow down muscle degeneration. We need to stall the disease for time. It is always possible the disease will go into remission,*" the doctor replied.

"*You said there isn't a cure!"* Phoebe said angrily. She wasn't mad at the doctor. She was mad because she wasn't used to feeling helpless. "*There isn't," the doctor said. "But sometimes the disease just stops and becomes dormant. He'll have it forever, but if his body can find balance, his muscles might return to normal. I don't want to give you any false hopes. This disease is extremely rare, and all patients react to the treatment differently. Your father's case is extremely aggressive. Yesterday he could walk. Today, he can't. He may never re-*

cover, or he may be totally fine next year. There is no way to tell. For now, we have to get those CK levels under control and stop the damage to your father's muscles." The doctor put a professional but comforting hand on Karina's arm and strode off, disappearing into the throngs of lab coats and scrubs that filled the hallway despite the late hour.

The next month was tough for the whole family. There were constant trips for chemotherapy, physical therapy, and endless rounds of tests. As the disease progressed, the girls' father lost the ability to speak, and one day, he could no longer swallow. The doctors implanted a rubber feeding tube in the wall of his stomach, and the strange new life continued. The four always held on to the doctor's idea of remission. It could happen any day or never. They all chose to hold on to *any day*.

Their mother, Karina, fought with the medical insurance company for weeks. They had denied all their father's medical claims because dermatomyositis is so rare there is no prescribed treatment for it from the United States Food and Drug Administration That meant all life-saving treatments the doctor tried were denied because the treatments were classified as experimental.

Another insurance battle was over a wheelchair. Clearly, their father, who had very little use of his arms and legs, needed an electric chair,

but they didn't have the $10,000 to purchase the one his physical therapy team had recommended. After months of hospital and outpatient treatments, the family was roughly 1 million dollars in debt, having wiped out all their savings, stocks, and retirement accounts. This made buying a chair on credit impossible. Their father was a proud man and refused to let the girls sell Bumble Bikes to cover his illness. In fact, the profits from Bumble Bikes were keeping the family afloat. After their mother went through several rows with the insurance company, Phoebe got that look on her face like the one she had when she was sitting upside down at the factory in front of the TV. Something was about to happen. Something radical.

The door to Phoebe's lab was locked for the next 3 days. Not even pizza boxes or fizzy water came in or out. Strange shadows danced in the eerie glow of half a dozen organic light-emitting diode monitors connected to Phoebe's computer, which was visible beneath the crack under her door. On the third day, she emerged like an angel rolling away a great stone. Tucked under her left arm was a rolled tube of large format printing. It looked like blueprints for something. Phoebe called everyone in the factory together for a big meeting. The factory in Alpine joined via Zoom. Intrigued, Kitty took her usual spot at the head of the table in the conference room. Over the next hour, Phoebe revealed her plan. Bumble Bikes would now be manufacturing wheelchairs! Kitty was excited, but there was one big problem she brought up after the meeting. The two of them were already stretched

thin running the bicycle business and tending to their father. The factories had to stay open to pay the mortgage. They would need help. They needed someone who knew software coding and architecture, preferably someone with experience working with disabled people, who were their target customers. They wanted someone for whom their new mission could become personal.

A physical rehabilitation clinic – Dallas, Texas

"*You have to use the bed to bounce yourself onto your chair,*" said Tom. His speech was muddy and loud, but his new patient understood. The patient had been diagnosed with some disease Tom had never heard of before: "*Dermato something something*." He was trying to teach the patient how to get out of bed and into a wheelchair. The new patient didn't have the muscle strength to stand up and sit down, therefore he couldn't maneuver from bed to chair easily. His doctors were considering letting him go home once he could master movement from the bed to the chair; the alternative was being sent to a nursing home.

The patient was determined not to go to the nursing home, and what he lacked in fine motor control, he could make up in large movements. Tom had the idea to have the patient propel his body upward,

and when he came back down, the springs in the bed would bounce him up just high enough to trampoline him into his wheelchair. The patient threw himself up with everything he had in him. He bounced. He missed and, lacking the ability to catch himself, landed with a hard thud on the cold tile floor. Tom didn't help him back up. The patient knew he had to learn to do this for himself. It was hard to watch but necessary.

"*Help him up NOW!"* a woman yelled from the doorway. Tom was startled at both the loud noise and the unusual sight of two stunningly beautiful women purposefully moving toward the patient. Clearly, logistics had not been a consideration when Phoebe barked the command. It wasn't as though Tom could just scoop up a 200-pound man and put him back in his chair. "*No, don't!*" the girls' father said weakly. He understood. He had to fight his way up into the chair even if it took a minute or several hours. "*Wait outside*," their father said. It was understood by all, including Tom, that they were dismissed.

The small group waited outside the hospital room and introductions were made. The girls had come to tell their father about their plans to make him, and everyone else that needed one, a quality wheelchair, even if they were unable to pay. Bumble Bikes had become a charity overnight. As Tom listened to the plans, his face changed and his body stiffened. He had not known it, but when he left home that morning, it was going to be the first day of his new job. He had always

dreamed of working as a software architect at a hot start-up. He knew his experience designing new software projects using patterns from the onset was going to be valuable. He had found his dream job.

Designing with patterns

Back at the Dallas factory, Kitty and Phoebe spent several days taking Tom through all the ins and outs of Bumble Bikes. He got an intimate tour of the factory (the physical one, not the pattern) and learned everything about its operations and capabilities. He also studied the advanced robotics systems designed and implemented by Phoebe and Kitty. It would be his job to design a similar system to manufacture the best wheelchairs anyone had ever seen.

Tom worked with Phoebe on the wheelchair designs. The designs were for three different models. The first, the flagship model, was Phoebe's idea for the ultimate power chair. It was called the *Texas Tank* and it would be comfortable and capable. The chair would be able to take its owner anywhere by using track designs similar to those on a tank or bulldozer. Stairs, offroad terrain, and even ice would not be a problem. You can see the **computer-aided design** (**CAD**) design here:

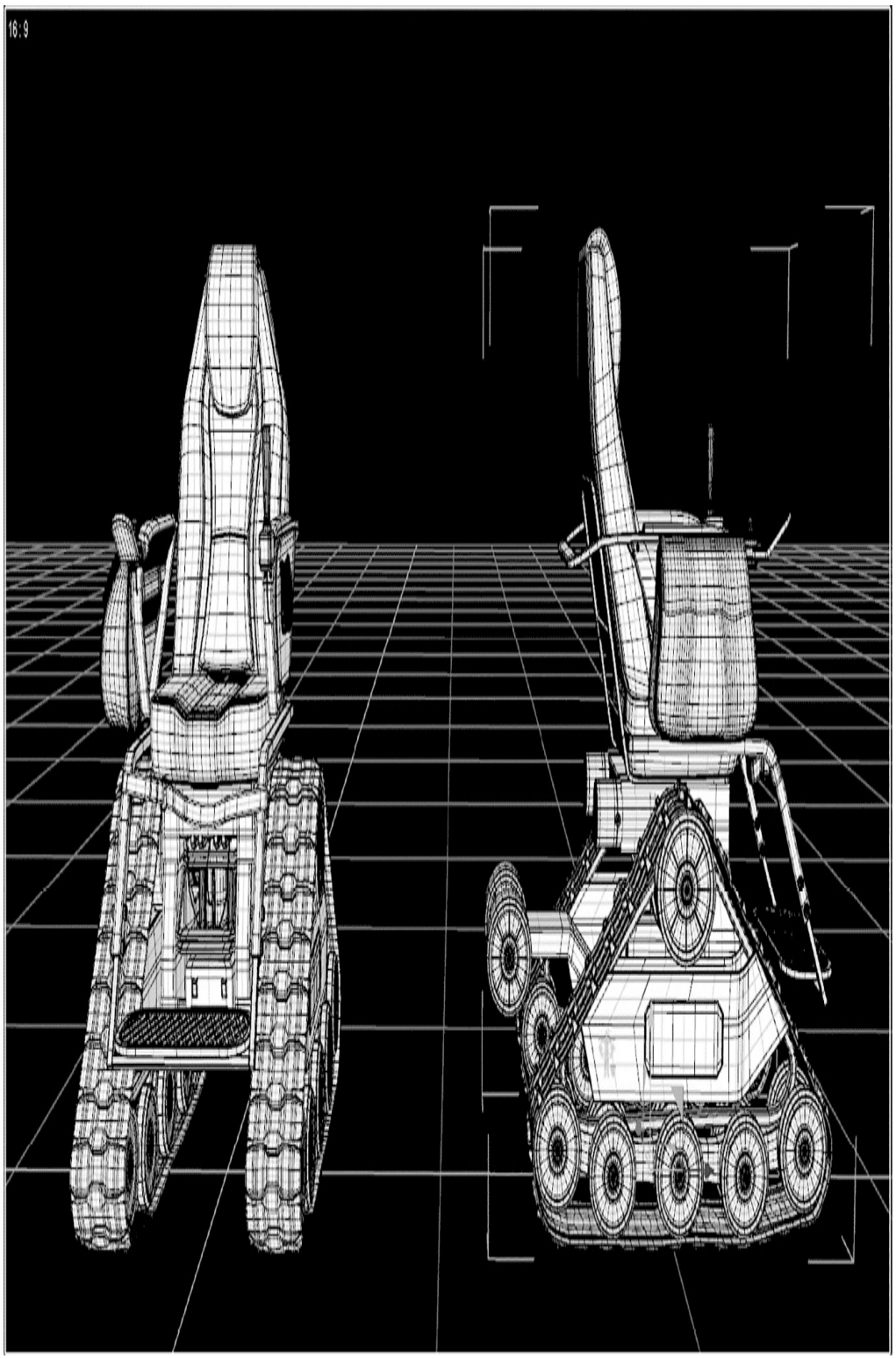


Figure 6.1: The Texas Tank would be the ultimate design for a powered wheelchair

Tom came up with a second idea. He planned to build a chair for people who couldn't use their legs but were otherwise very strong. He planned for something light and fast that might be used for competitive sports, such as basketball. Phoebe, with this goal in mind, devised a wheelchair with cambered wheels and shorter lower-back support to allow a greater range of motion. She called it the *Maverick*, which is a cowboy who plays by his own rules. You can see this design here:

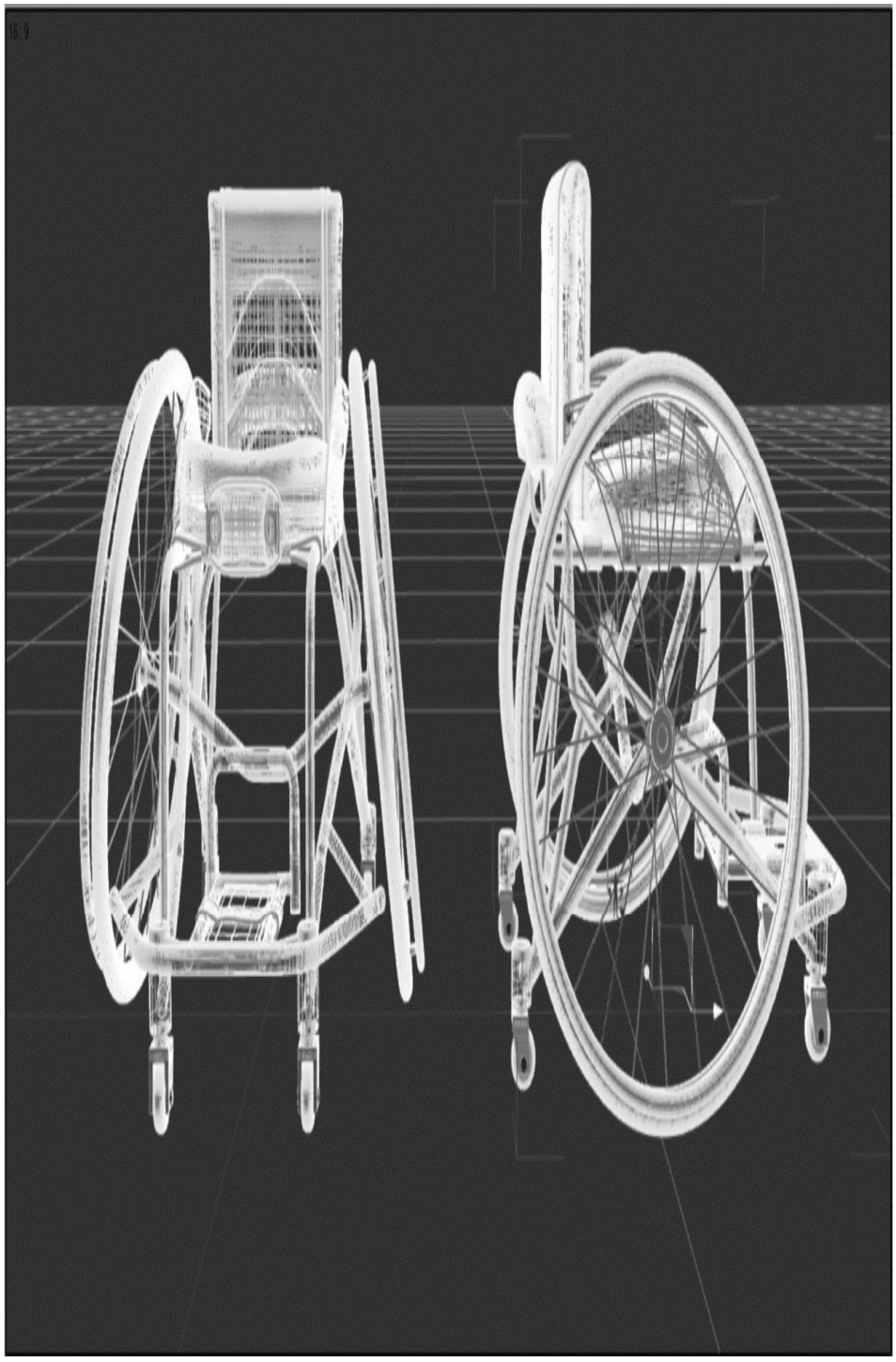


Figure 6.2: The Maverick

The last model is what Phoebe bemusingly called the *Plano Wheelchair*. “It’s, you know, just a plain ol’ wheelchair,” Phoebe said. So far,

all of Bumble Bikes' products were named after something related to Texas. The city of Plano, Texas, lent its name to the pun. It's light, compact, and functional, and it gets you where you need to be. "Maybe it's used temporarily, or with some upgrades to the seat, maybe you use it forever," Phoebe explained. You can see the design for the *Plano Wheelchair* here:

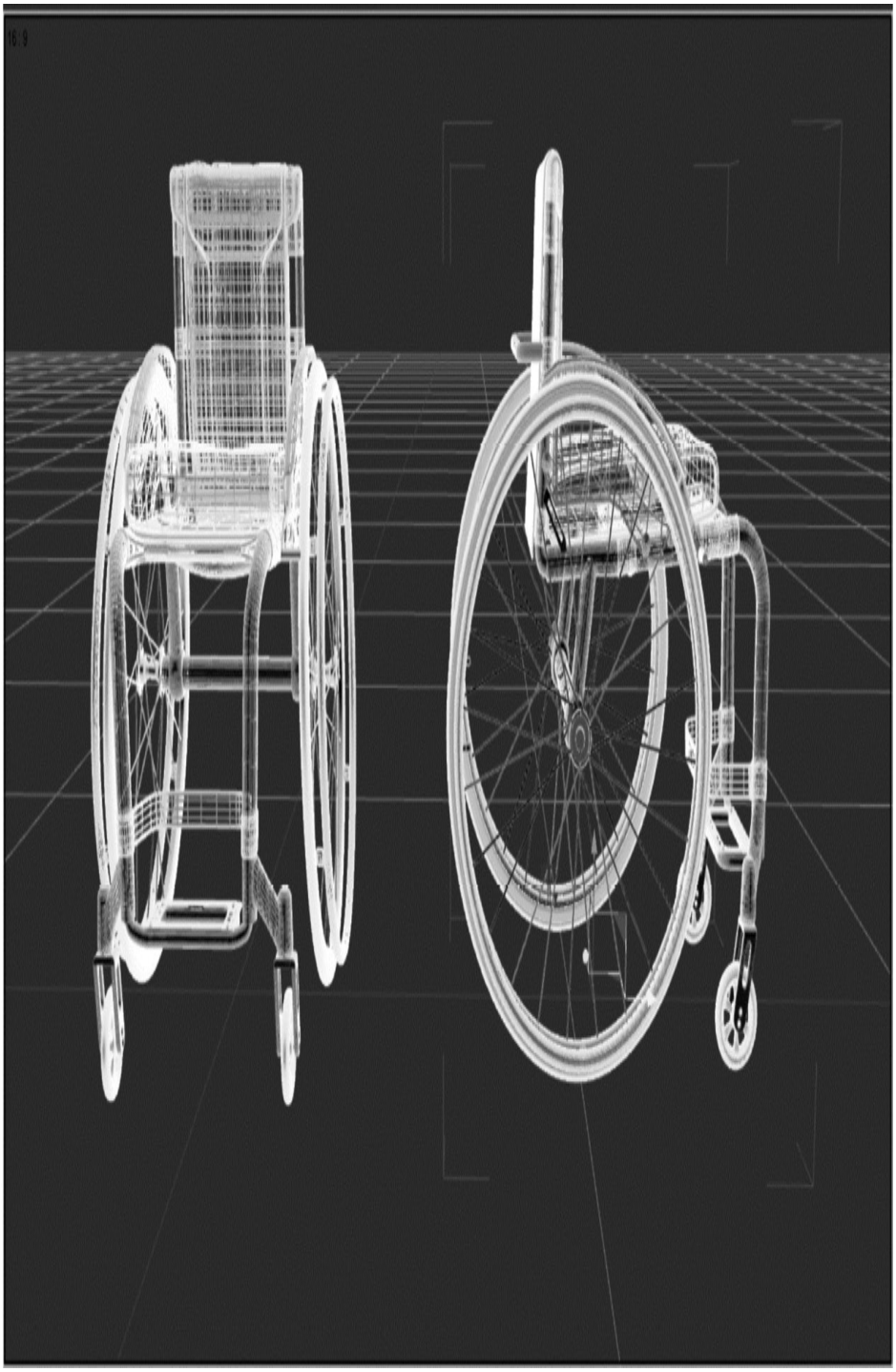


Figure 6.3: The Plano Wheelchair, because it’s just a plain ol’ wheelchair

Phoebe and Tom showed their designs to Kitty. She loved them. Phoebe stated, “When we made the bicycle factories, and the software, we built everything as we went along. We didn’t really know what we were doing. We learned patterns as they were needed. We’re kind of hoping we could leverage our experience along with yours to get this done a little faster than last time.”

“And hopefully with fewer mistakes,” Kitty chimed in. Phoebe rolled her eyes. “It all works, doesn’t it?” Phoebe shot back.

Indeed, it did. It wasn’t perfect, but everyone working at *Bumble Bikes* had learned that “perfect” is the enemy of done. You can’t wait for your software to be perfect. No bicycle would ever be built that couldn’t be improved. Whether it’s software, bicycles, or anything else, designing and coding don’t bring in money, which is the main goal for most companies. These activities are necessary, but only shipping products based on design and code turns a profit. If you code or design for too long, your company goes bankrupt.

“I need to diagram everything in UML before I start coding,” said Tom. Every day the sisters spent with Tom made it easier to understand his stammering speech. Phoebe just stared at him for a minute. Only Kitty understood the nonverbal “Who does that?” expression on her face. Tom explained designing with UML before opening the IDE will help the project move faster. You’ll see your design mistakes sooner,

and it is usually easier to refactor a diagram than it is code. Your diagrams are also where you'll identify patterns in advance, rather than reactively when you are trying to refactor something already in production. Besides, it is impossible to succumb to the real-world temptation to *clean up and ship* a diagram. Effectively, you should be able to replace *throw-away code* with a set of diagrams.

It was important for the sisters to learn to trust this qualified outsider. It was his project, and the girls knew the value of not micro-managing Tom. Phoebe suited up a laptop for Tom with the finest tools money can buy, a condition Tom insisted on as part of his employment. Kitty and Phoebe acquiesced to this condition without a second thought. They showed him to his own lab, which the girls had hastily, but competently, fashioned from studs, insulation, and drywall in a back corner of their Dallas warehouse. They painted the room, and as an office-warming gift bought Tom a bonsai tree from a pop-up vendor stationed on a nearby street corner. The office was furnished using some of the extra furniture and equipment Phoebe had been hoarding in her own lab. "First rule of working in IT…," she said, "Never give ANYTHING back!" This was humorous, given she was half owner of the company and in charge of IT. Humorous, but true.

The first pass

The low-hanging fruit in any object-design exercise is deciding on the basic structures for the objects. Tom knew this was likely the place he would use Creational patterns, though he didn't concern himself too much with which ones he would use. Experience had taught him that analysis-paralysis is a real problem when a developer starts using patterns. You can stare at a blank whiteboard all day trying to decide whether to use an Abstract Factory or a Factory Method. Maybe both together somehow? Maybe we can fit a Command or Singleton in there too?

Tom began by drawing the classes and not worrying about patterns at all. Patterns usually emerge out of chaos, so don't be afraid to create chaos first. Since we're just diagramming, there is no way the pointy-haired boss can tell you to "Clean it up and ship it Monday." Tom's new bosses would never give him that command. Neither Phoebe nor Kitty had pointy hair, nor were they likely to mistake an *Etch A Sketch* for a quality tablet computer; this is the subject of an inside joke the girls had shared while working with the IT executives at MegaBike Corp two summers ago.

Tom reasoned that wheelchairs are like bicycles, but not so similar that there would be any reuse in the base classes. That said, the project could be structured in a similar way. Tom set to work creating a set of interfaces and classes that would represent Bumble Bikes' new line of powered and unpowered wheelchairs.

I'd like you to stop reading and see whether you can create a UML diagram that describes powered and unpowered chairs as they were described by Tom and Phoebe. When you have finished your diagram, see how closely your work matches Tom's.

Tom started by diagramming the `Wheelchair` class, like so:

Figure 6.4: Tom's initial Wheelchair class

At first, the `Wheelchair` class had everything on it. But as he considered further, he decided to split the chairs into two families: **powered** and **unpowered**. See following diagram:

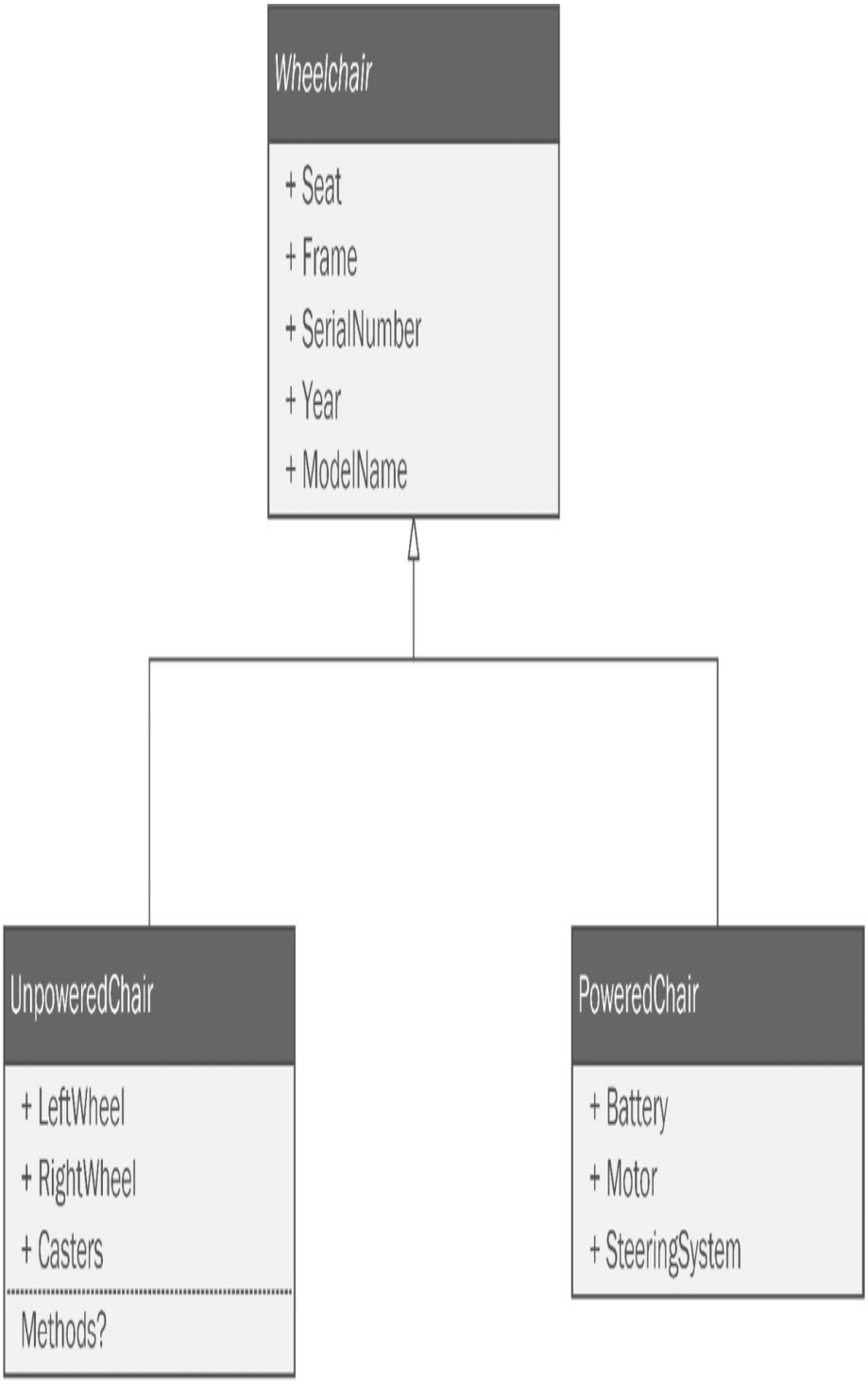


Figure 6.5: Tom split the design into two main types of wheelchairs

He considered which properties these families would have in common, such as **`ModelName`**, **`Year`**, and **`SerialNumber`**. He hoisted these definitions into an interface and an abstract base class. He could probably get by with one or the other, but since his entire design would be structured around this diagram, he decided being inclusive was the way to go.

The unpowered chairs were straightforward with regular wheels and casters. The *Texas Tank*, on the other hand, was an entirely novel design that used tracks instead of wheels. The powered chair would also need a power source, a motor, and a steering system. These could be diagrammed separately, but Tom realized that since the track designs on the powered chair were so strange, it might behoove him to design around this fact. He made an abstract **`PoweredChair`** class that encapsulated the other 99.9% of the powered chairs on the market. It wasn't inconceivable that he might one day be asked to represent such a chair in his code.

The whole diagram wound up looking like this:

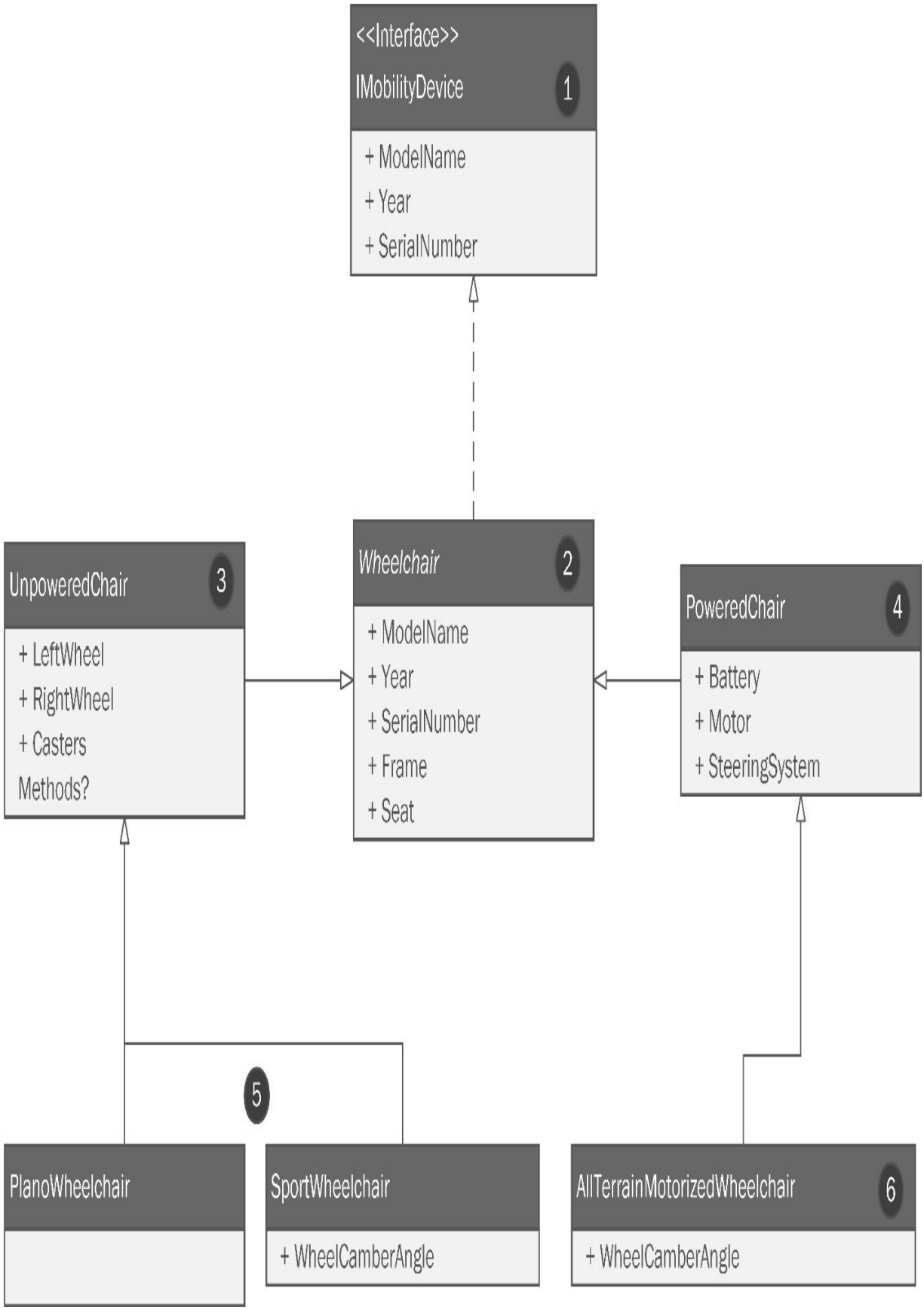

<<Interface>>
IMobilityDevice
1
+ ModelName
+ Year
+ SerialNumber
Wheelchair
2
+ ModelName
+ Year
+ SerialNumber
+ Frame
+ Seat
UnpoweredChair
3
+ LeftWheel
+ RightWheel
+ Casters
Methods?
PoweredChair
4
+ Battery
+ Motor
+ SteeringSystem
5
PlanoWheelchair
SportWheelchair
+ WheelCamberAngle
AllTerrainMotorizedWheelchair
6
+ WheelCamberAngle

Figure 6.6: Tom's initial design

Let's look at it by the numbers, as follows:

1. Tom decided an interface at the top of this structure made sense because, in the future, we need to be able to pass around wheel-chair-like objects. This is an attempt at future-proofing. This kind of design strategy usually works, but really, there is no such thing as *future-proof*.
2. An abstract implementation of the interface is needed when there's some shared method implemented across subclasses. We have something like that in the **`Bicycle`** class with the **`Build()`** method, as shown here:

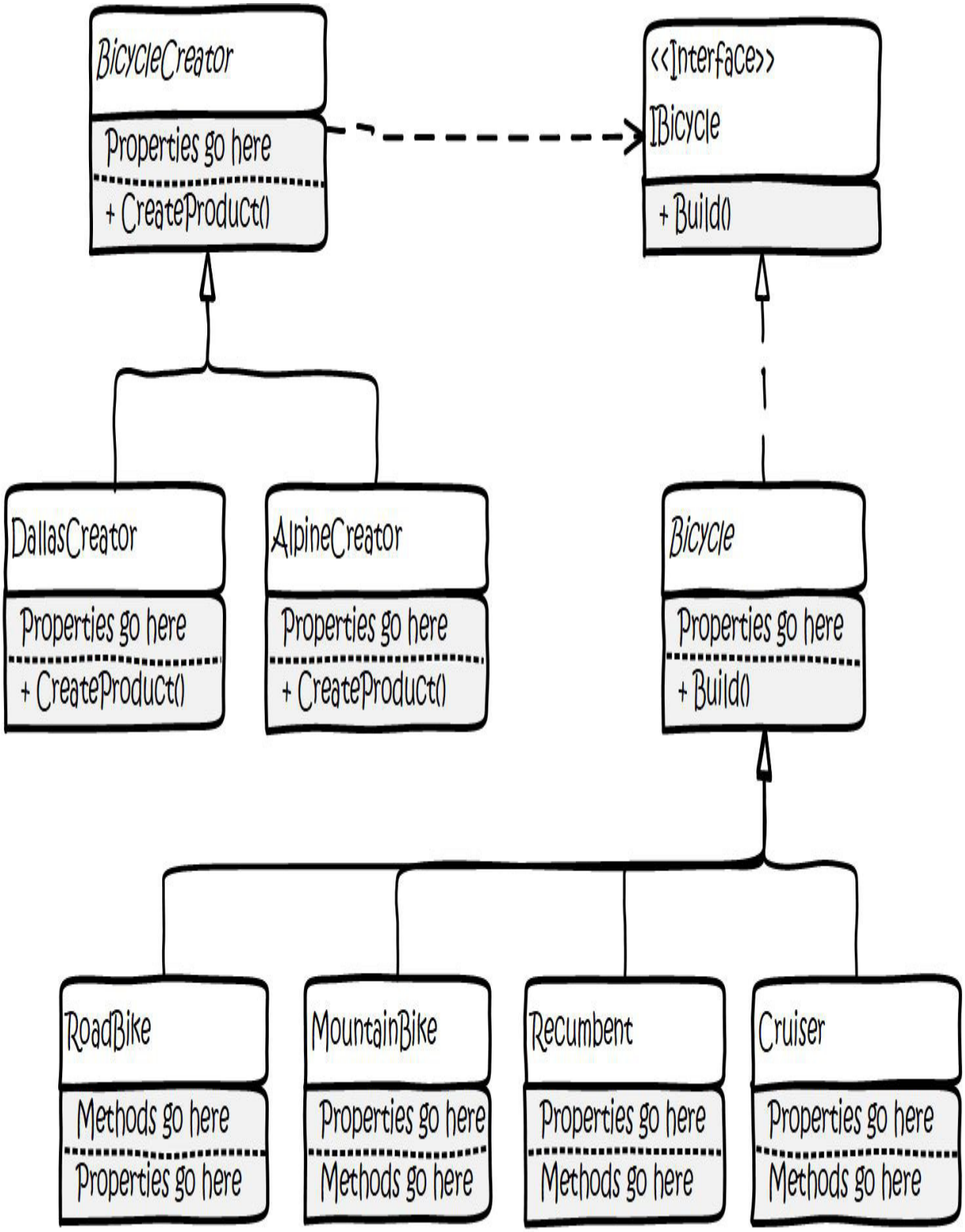


Figure 6.7: The Bicycle Factory method pattern implementation from Chapter3.

3. As with the **`Bicycle`** class, this class has two computed properties: **`Year`** and **`SerialNumber`**. These act like methods. At this stage, we can justify the extra abstract class in the name of being DRY. For an explanation of **don't repeat yourself** (**DRY**), please refer back to *Chapter 2*, *Prepping For Practical Real-World Applications of Patterns in C#*.
4. The **`UnpoweredChair`** class inherits from **`Wheelchair`**. Tom knows the left and right wheels on a wheelchair may look identical. In this first drawing (*Figure 6.4*), he had all the wheels in an array. Wheels on a wheelchair are not identical, and Tom understands he needs to not think so concretely. He isn't modeling wheels as part of the chair. He's modeling slots for wheels that should on some level be interchangeable. Most wheelchairs have two big wheels as the main support, and a set of smaller wheels, or casters, to balance the craft overall.
5. The **`PoweredChair`** class also inherits from **`Wheelchair`**. It is modeled to represent the most common powered chairs on the market. Since Tom has ridden around in one for the entire duration of his 36-year life, he knows the design very well. His class here inherits common properties—some computed—from the base class. The elements unique to the powered chairs, in general, are represented. Every powered chair, including the *Texas Tank*, will need a power supply, steering, and a motor. Again, this is an attempt at future-proofing. It is easy to predict that one day, Bumble Bikes will

make a more common design for a powered chair. While the *Texas Tank* is amazing, Phoebe designed it as something she knew her father would love if remission never becomes a reality. The *Texas Tank* is not for everyone. It is heavy, expensive, and will smell like heavy machinery. Note that Tom has considered there will need to be methods in the design, but in this pass, he hasn't committed anything to the diagram. He continues to work on the obvious needs as quickly as he can to keep the momentum.

6. Two classes inherit from the **`UnpoweredChair`** class. I think these speak for themselves. The only difference between the two is the cambered wheels on **`SportChair`**. In an actual chair, the wheels tilt outward for improved speed and agility. If you didn't notice this detail, refer back to *Figure 6.1* and you can see the camber on the wheels.

Tom is happy with his morning's work. Kitty and Phoebe order lunch from their favorite Mexican restaurant and the three relax. After lunch, Tom is feeling energized and goes to work on the next round of revisions.

Once you get your base classes diagrammed, the next step is usually to think about composition. We have properties and fields in our objects. What information will go inside those properties and fields? When we were beginner coders and things were simpler, it was a combination of primitives. We didn't need composition, but now we

are worldly and wise, and Tom is no exception. He needs to hash out the structures that make up the base classes via composition. For example, what should we do with the `Seat` property in the `Wheelchair` base class? The various chair designs have different seats; therefore, it makes sense to create an interface for maximum flexibility, and then document that requirement in the diagram. Tom is taking a different attitude and approach by diagramming and planning. This is different from the attitude that Phoebe and Kitty had taken while building Bumble Bikes. They skipped any sort of diagramming or planning effort and went directly to writing code. Just look at all the parts where Tom is guessing or making judgment calls. The `Inflate()` and `Deflate()` methods are a great example. He's not sure whether they will be needed or not. The exercise of abstracting a real-world object into class code is mainly about modeling what is important to the application. A `Person` class in an address book app on your phone needs a different level of detail compared to a `Person` class in a medical records application. While you are diagramming, you needn't worry about being DRY, **YAGNI** (which stands for **you aren't gonna need it**), or violating SOLID (if you don't know what these acronyms are, please refer to *Chapter 2*). At this stage of designing your class structure, it is only a diagram. The whole point of diagramming is to get the design in a format that can be discussed, argued over, pondered, socialized, and changed. Sure, you're thinking about SOLID or YAGNI, but it's okay to not be perfect at the point of conception. The

diagram will naturally evolve, and you will make mistakes and have oversights that you can correct and refine. Once the diagram becomes code, all those pesky rules will begin to apply. The code is real. There are rules, and make no mistake about it, if you work on a team, you will be judged by those rules.

Tom diagrams all the components he foresees, including these:

1. The seat
2. The frame
3. Wheels and casters
4. The motor for the powered chair
5. The steering mechanism for the powered chair
6. The battery for the powered chair
7. The track drive system for the *Texas Tank*

Let's watch Tom diagram each set of classes. Each set is going to consist of an abstract class or interface, and appropriate concrete classes inheriting from that abstract class. Don't forget—in UML, the title of an abstract class is written in *italics*.

The seat

The seat is probably the most important part of a wheelchair. Tom knows this first-hand—each chair will need a different kind of seat. Bumble Bikes can get away with the less expensive generic seating

on wheelchairs that are not meant to be sat in all day. The *Plano Wheelchair* is the perfect example of a transporter chair. A basic wheelchair, like the *Plano*, is used for brief transport from one location to another. The chair's user, for example, will transfer from the wheelchair to an office chair or bed. When the need arises to move from one spot to another, the user will transfer back to the transporter chair and set out to the new location. The basic wheelchair does not have a motor. This differs from the other wheelchairs.

The *Maverick* wheelchair too has a special-purpose chair. This chair is designed to be light and maneuverable. Similar to the *Plano Wheelchair*, it isn't designed to be an all-day seat. The *Maverick* wheelchair is not motorized.

The powered chairs, however, cater to different kinds of users. The largest group of users consists of aging senior citizens. This group is not disabled. They can stand and walk short distances but require a chair for longer-distance walking. The second group of wheelchair users are like Tom. They are quadriplegics and have little-to-zero mobility in their arms and legs. These wheelchair users generally lack the ability to transfer on their own. Therefore, they do not need a cheap seat because they are in the chair for long stretches of time. They require a comfortable chair. This group, Tom and Phoebe decided, would be the design inspiration for the *Texas Tank*, the ultimate powered wheelchair.

Tom drafts a design for the seat classes. His ideas are reflected here:

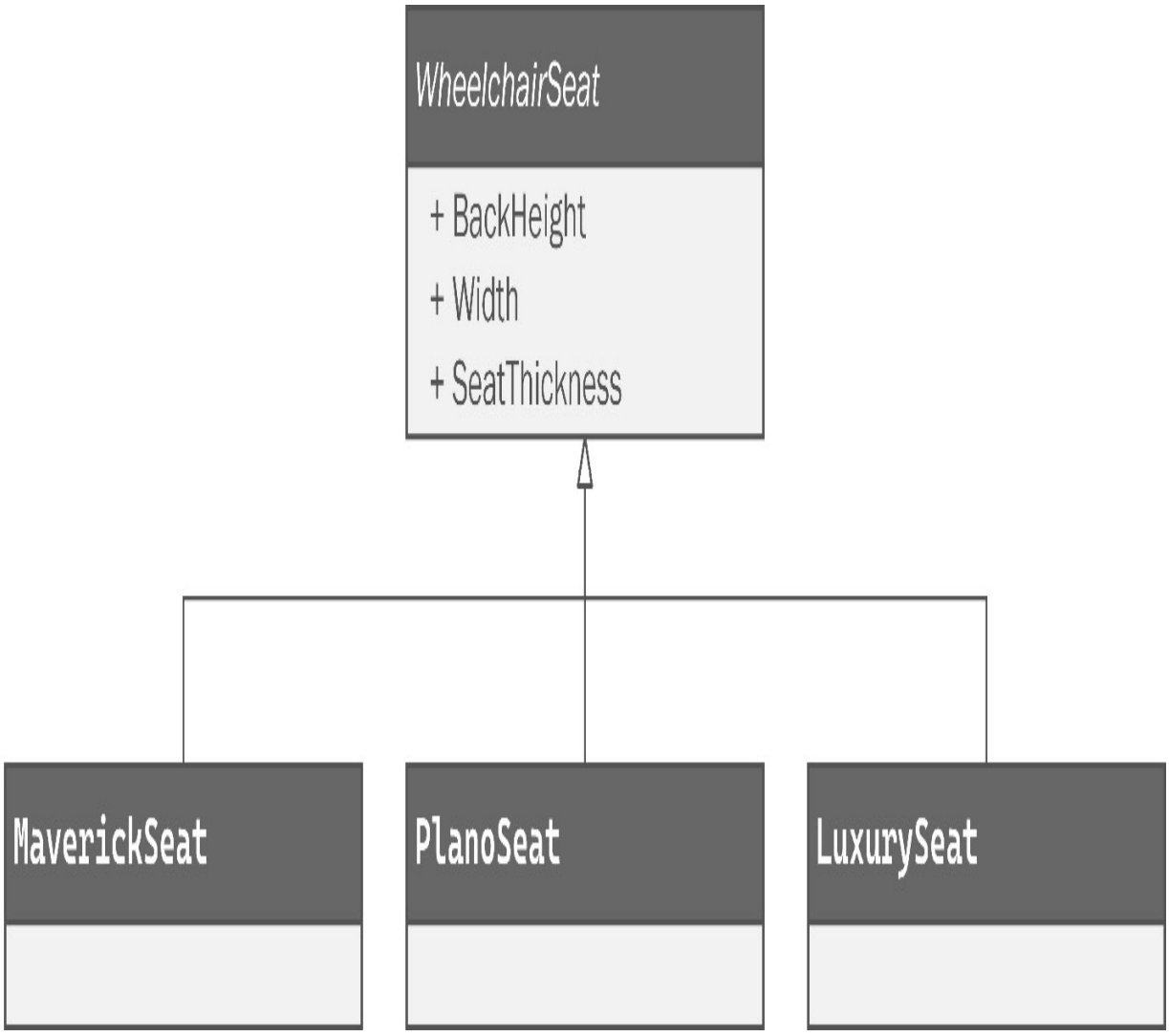


Figure 6.8: Tom's design for the WheelchairSeat class

This design is simple. It consists of an abstract class called **`WheelchairSeat`**. The properties are straightforward. Tom positioned the concrete classes with the unpowered chair seats on the left side of the diagram and the powered chair seat on the right. Later, when

we combine our designs into a larger diagram, you can expect this convention to be used.

The frame

Our three wheelchairs require three very different frame designs. The *Plano Wheelchair* is simple, reasonably light, and a very common design. The *Maverick* wheelchair needs to be very light, but tough enough to weather collisions and crashes, which are inevitable during passionate competition. The *Texas Tank* needs to be, well, a tank. It has heavy tracks, a large battery, and an oversized chair for a seat.

Tom gives this some thought and decides to keep this simple for now. He could try to get down in the weeds and figure out how to represent all the extruded aluminum tubes, bolts, joints, and hinges that make up the chair frame. However, Tom isn't that kind of engineer, so he decides to include a field to hold the file path to the CAD file for the frame. Kitty and Phoebe agree with Tom. You can review Tom's design in *Figure 6.9*. The diagram shows a simple abstract class representing a wheelchair frame. As before, the unpowered components are on the left, while the powered class is drawn to the right:

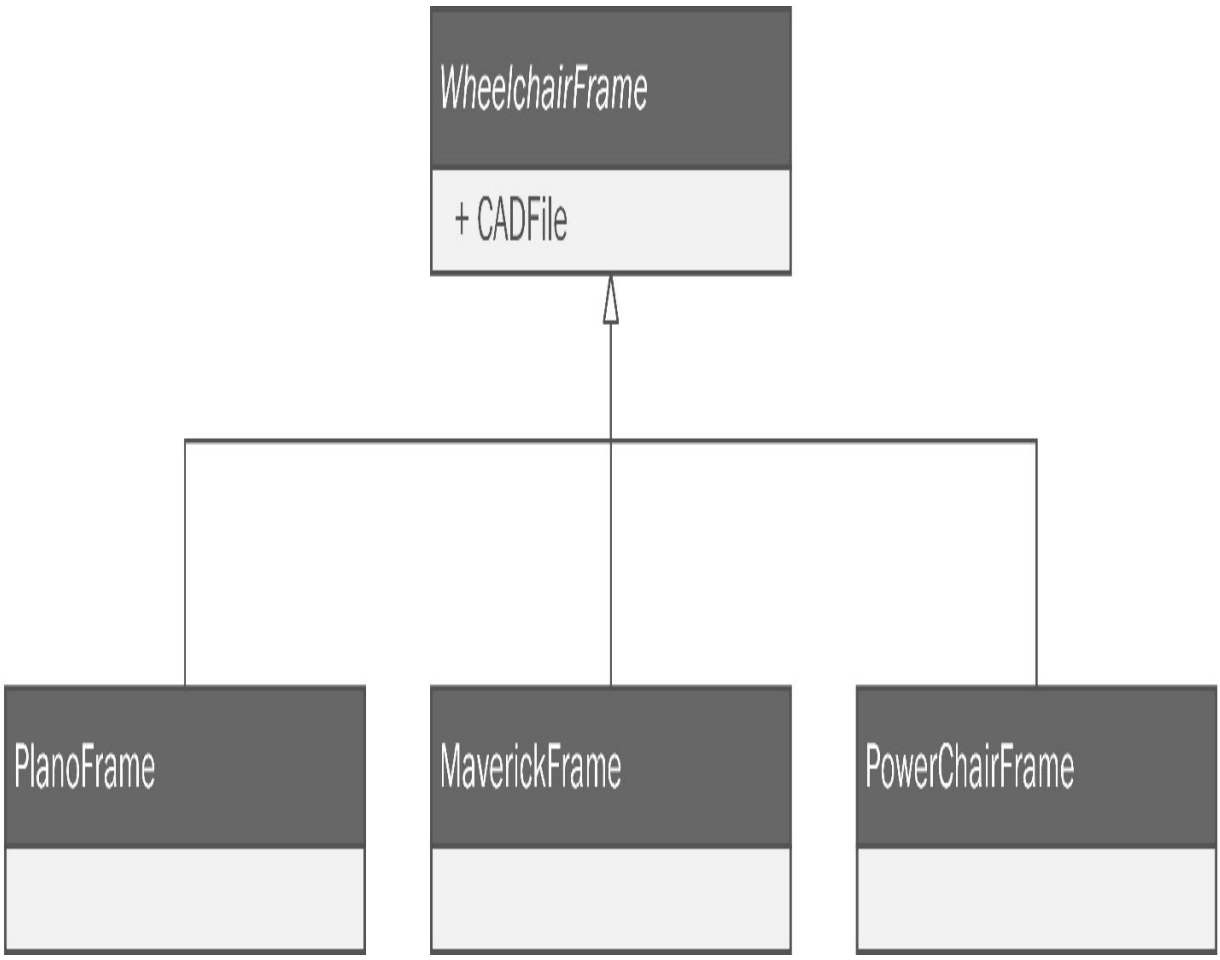


Figure 6.9: Tom's design for classes representing the frames of the wheelchairs

Wheels and casters

You can't really have a wheelchair without wheels, can you? The *Plano Wheelchair* and the *Maverick* wheelchair will use traditional spoked wheels, similar to those used in bicycles. Standard designs for wheelchairs involve two large fixed wheels and a set of smaller wheels configured as casters. The casters can turn in any direction.

In Figure 6.10, Tom calls his abstract class **`MechanicalWheel`**. This class can represent the wheels and casters, but it cannot represent the tank-style treads on the *Texas Tank*. The following diagram shows the design for the **`MechanicalWheel`** class along with the two-wheel subclasses. The **`WheelchairWheel`** class represents the larger wheels on the sides of the chair, while the **`Caster`** class represents smaller wheels on a wheelchair that can rotate, providing maneuverability:

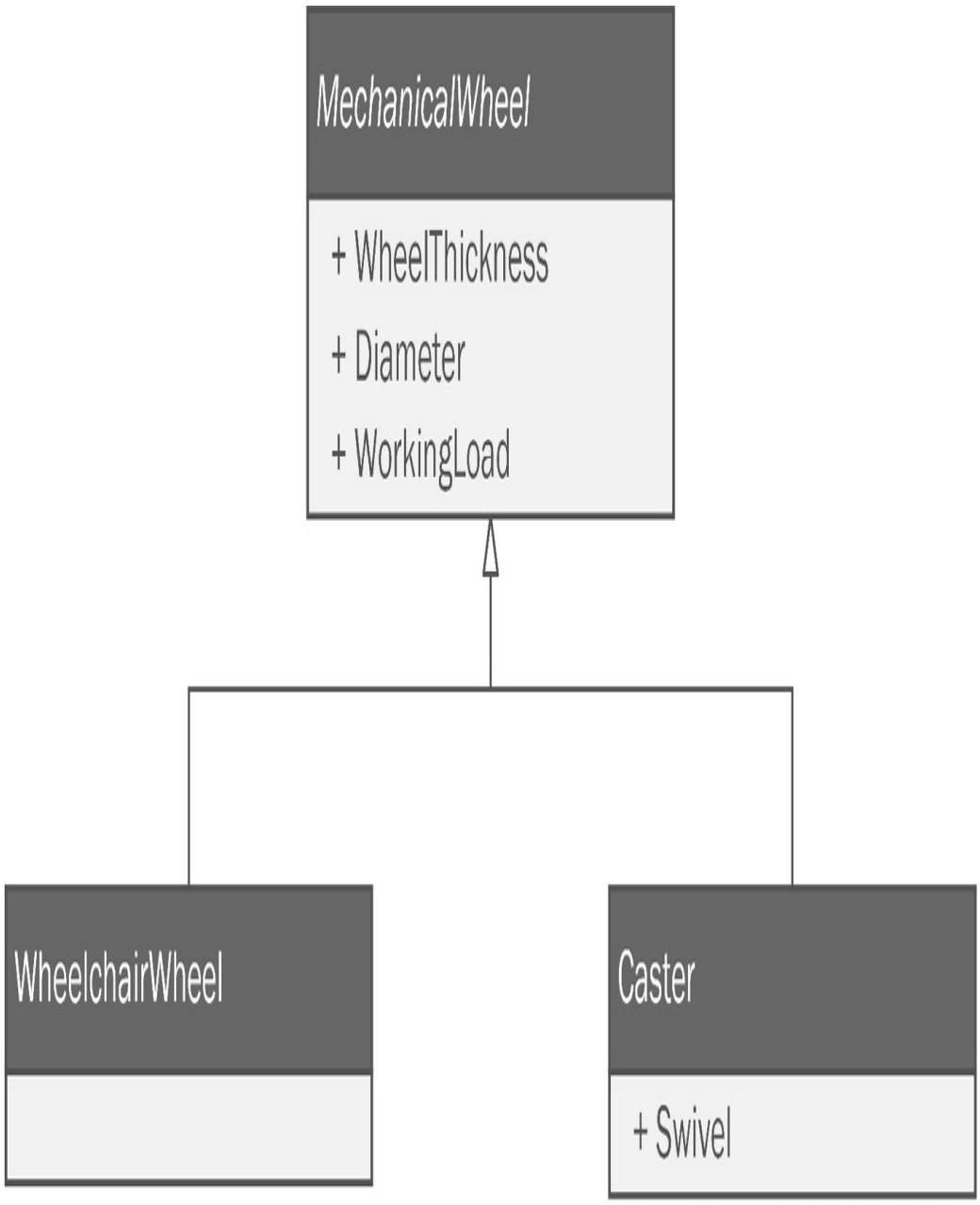


Figure 6.10: The MechanicalWheel abstract class and its subclasses

Tom has modeled everything he needs for an unpowered chair. They are simple machines. The real design challenge, and certainly the most rewarding, lies in creating a class design for the powered chair.

Tom is careful to not specifically model the *Texas Tank*. The *Tank* is a very ambitious project—the kind that can easily be canceled because it's going to be expensive to build. Furthermore, the *Tank*'s unconventional design might not appeal to a wide audience. Only time and a few marketing studies can tell. Tom's strategy then should be to create component classes that are not specifically tied to the *Tank* design, but rather to more common powered chairs. Tom decides to start with the motor.

The motor for the powered chair

Tom's powered chair is still running with its original motor. He bought the chair over ten years ago. These motors are common and Tom is intimately familiar with them. The motor for a powered wheelchair operates off direct current from a battery. The motors are required to have high torque with a relatively show speed, and they need to last a long time. Modeling a such a motor is not a problem. Tom's design is shown in Figure 6.11.

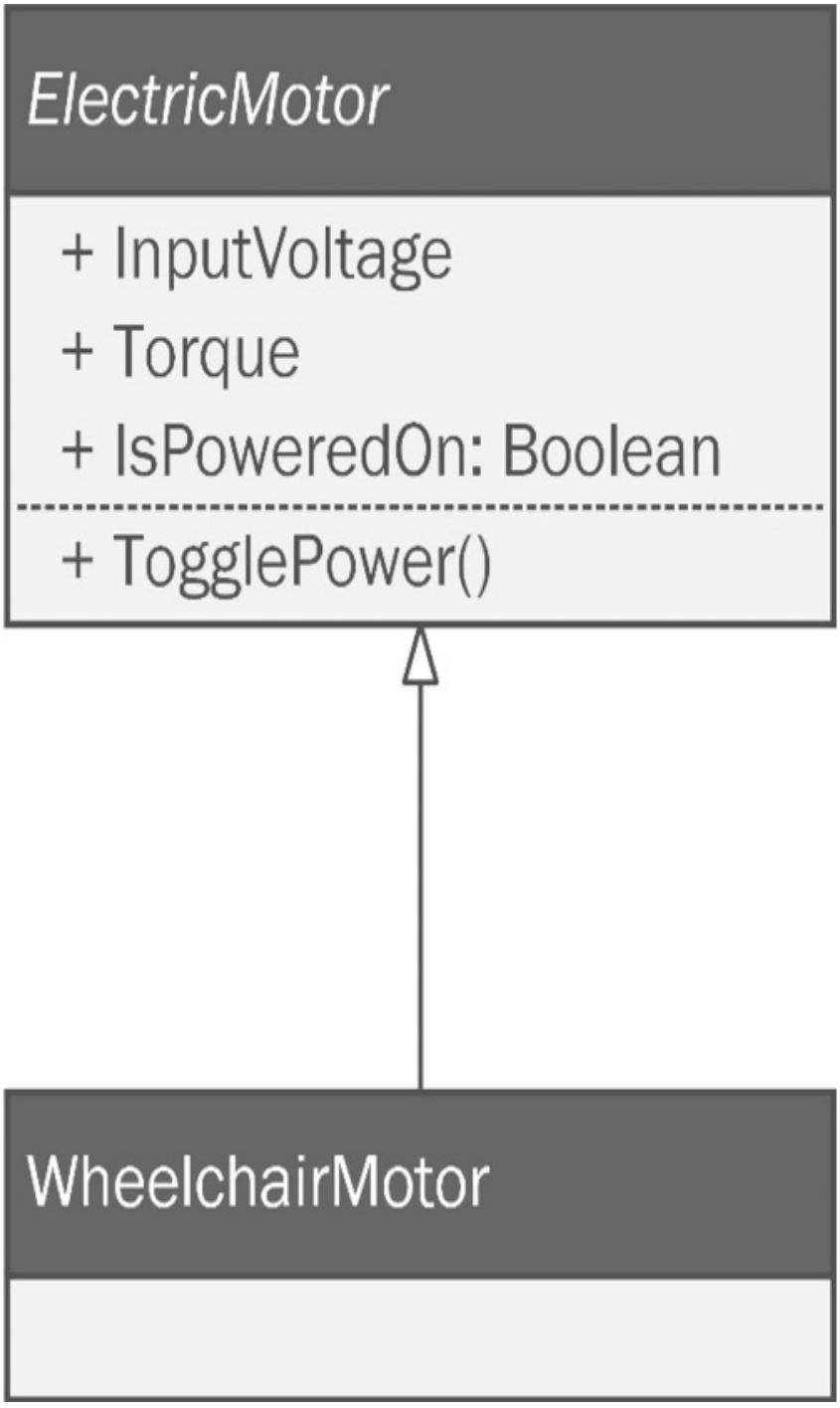


Figure 6.11: The motor class diagram for powered chairs

Since electric motors are common and generic, Tom decides to make them an abstract class. He adds a few properties and a method to

simulate the motor being switched on and off. Then, he adds a subclass to handle the specifics of an electric motor used to power a powered chair.

The steering mechanism for the powered chair

Tom's chair uses a joystick mounted to the left arm of his chair to control it. The joystick is sensitive to how hard it is pushed in any given direction. The harder you push the stick, the faster the chair moves. Tom sees no particular reason to redesign this nearly ubiquitous system. His class design can be seen here:

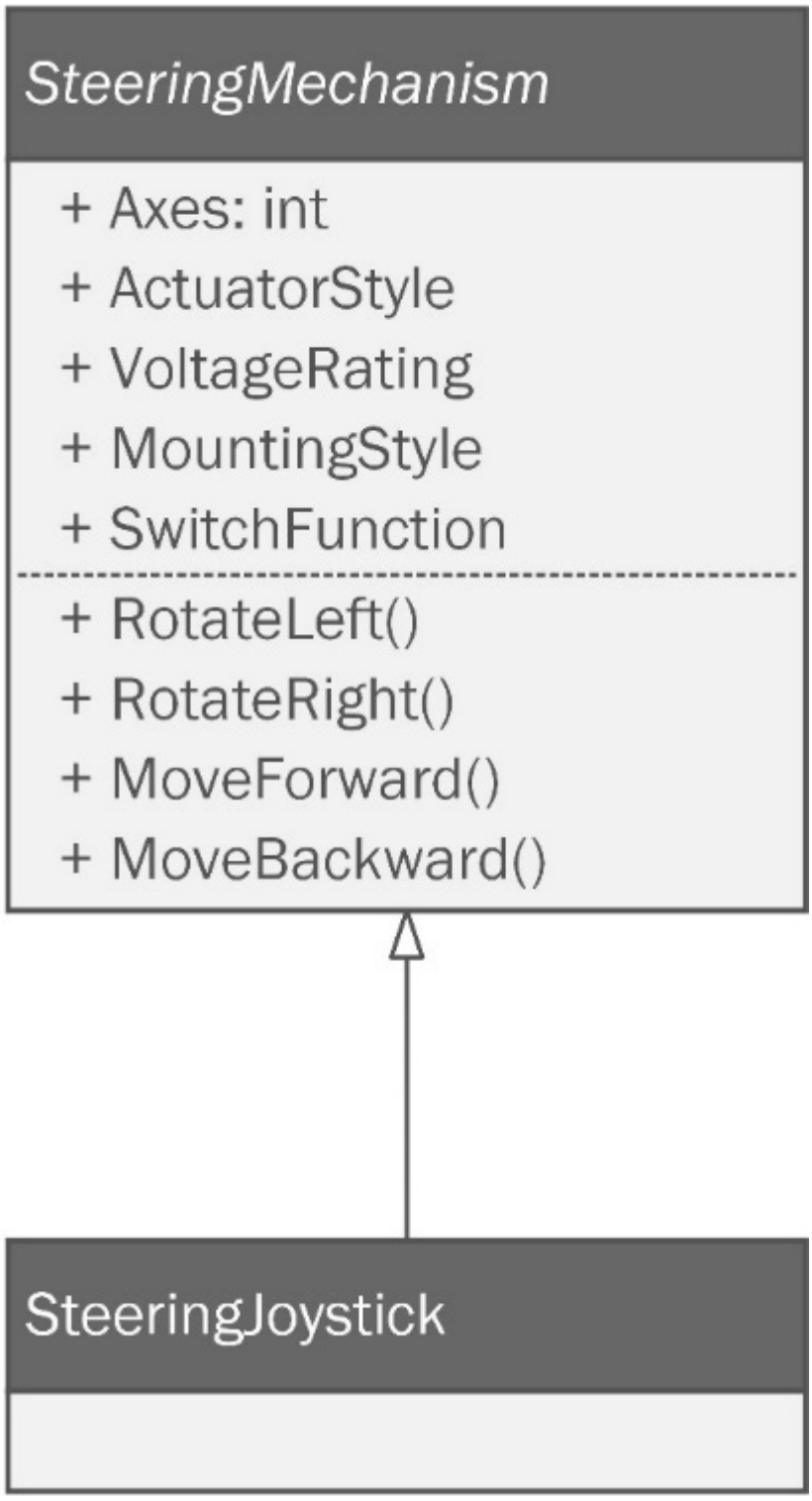


Figure 6.12: The steering mechanism for the powered chair design

The properties Tom chose for the steering mechanism represent a wide range of industrial joystick control systems, used in everything from drone flight controls to automation to—you guessed it: powered wheelchairs.

The battery for the powered chair

The motor for the wheelchair is electric, and we need a battery to power the system. Any industrial-grade battery with the right voltage and current should work. We want a battery that lasts a long time. We need a battery with a high enough **ampere** (**amp**)-hour rating, but not so high that the weight of the battery affects the operation of the chair. Phoebe will definitely want to experiment with different batteries. As you can see in the following diagram, Tom created a base class that might describe any battery and left the implementation details for later on in his `WheelchairBattery` subclass:

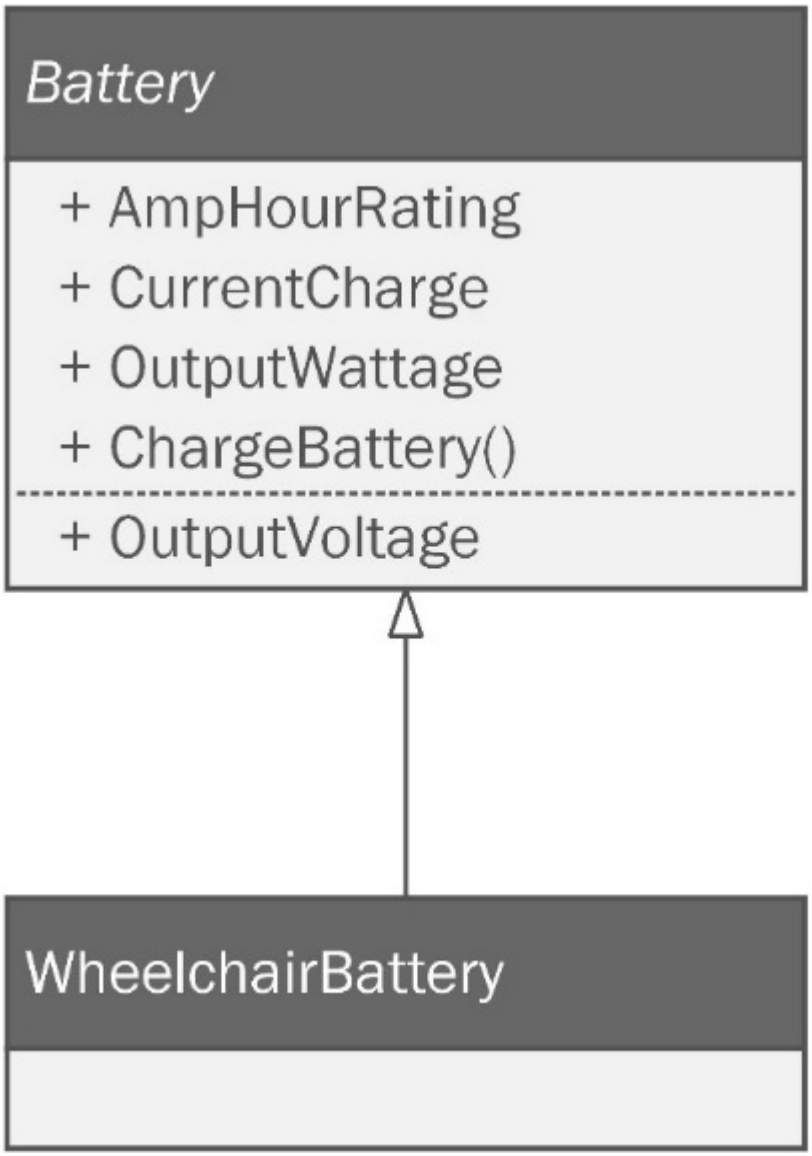


Figure 6.13: A class diagram representing a battery system for the powered wheelchair

The track drive system for the Texas Tank

Tom was out of his element on this one because he had zero experience with track drive systems. Kitty and Phoebe, having spent time at the family cattle ranch when they were young, had a tacit understand-

ing of how track drive systems worked. The three teamed up and sourced a few vendors that might supply their requirements. Since Tom was pressed for time, he simply looked at the properties used to describe the track drive systems on the vendors' websites. From these sources, Tom derived the set of properties you'll see in the following diagram:

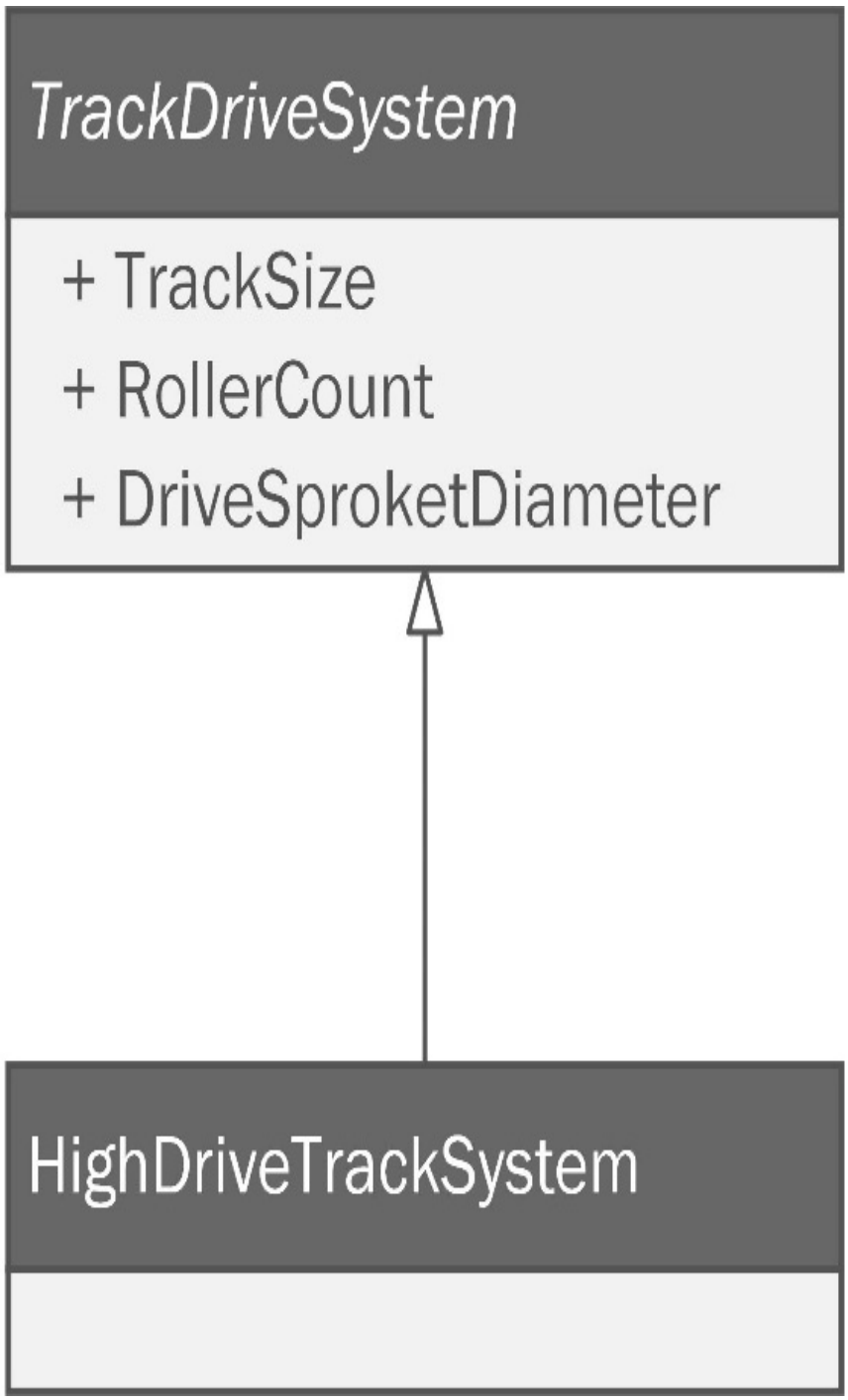


Figure 6.14: The track drive system

Tom was sure that of all the components he modeled, the track drive system would be the one to change and evolve. This is the one where

Phoebe would be working on many prototypes as she built them. He called his subclass **`HighDriveTrackSystem`** because this is what the design is called. In the following diagram, you'll see our *Texas Tank* design with a high-drive system. The track is triangular with a large wheel situated high in the track design. Compare this in Figure 6.15 with a standard continuous track design on a bulldozer, and you'll see the difference:

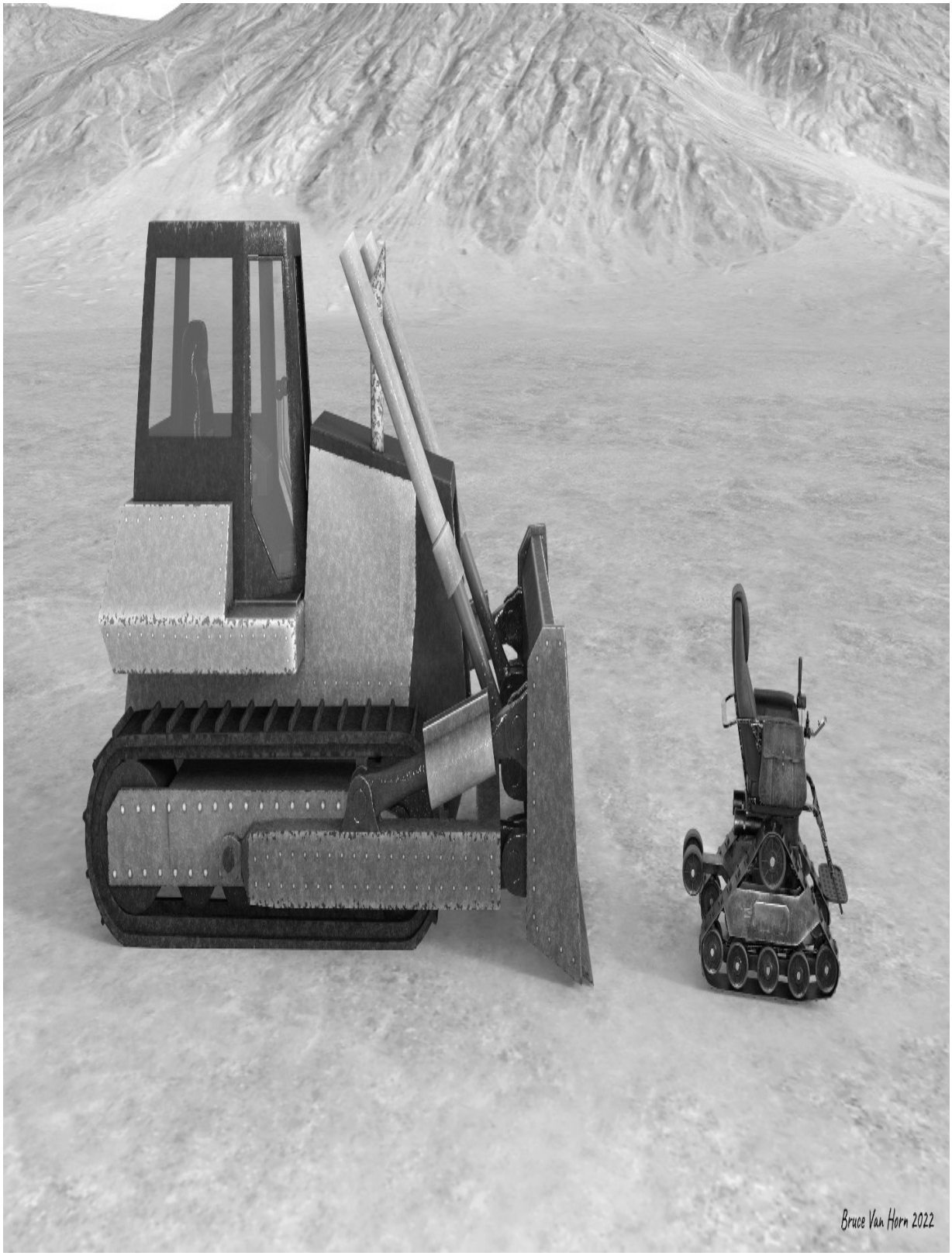

Bruce Van Horn 2022

Figure 6.15: A continuous track drive system on a bulldozer compared with the high-drive design on our powered chair

Phoebe and Kitty designed the chair using the high-drive system because it provides a more absorbent suspension. High-drive systems are also less prone to wear and fail because they isolate the drive sprocket to a flexible section of the track drive. “She’ll handle like a dream!” Phoebe exclaimed with a broad, toothy grin. The engineering prototype didn’t take long to build. The hard part was getting Tom off the *Tank*, as we can see here:

Figure 6.16: Tom monopolized the Texas Tank prototype

Eventually, the *Tank*'s battery was depleted, at which point, Tom got back to work.

Adding patterns

Tom feels pretty good about his first pass. He has enough to spark some serious discussion about how to structure the project. This is the objective of the first pass. Tom makes a quick list of all the patterns Kitty and Phoebe had used in the bicycle project, as follows:

1. **Creational patterns**:
    1. Factory pattern
    2. Abstract Factory pattern
    3. Factory method
    4. Builder
    5. Object pool
    6. Singleton
2. **Structural patterns**:
    1. Decorator
    2. Façade
    3. Composite
    4. Bridge
3. **Behavioral patterns**:
    1. Command

2. Iterator
3. Observer
4. Strategy

Tom begins to focus on the diagram and sets to thinking about the application of each pattern. One by one he looks for opportunities. Some of them are obvious. Some of them may not be needed. Tom's diagram doesn't currently include any changes to the bicycle factory's robotics. For the most part, the robotics classes should not be tightly coupled to making bicycles. After a few hours of personal debate and much consideration, Tom is prepared to discuss patterns with Kitty and Phoebe. He is able to book time on their calendars for that evening.

See whether you can generate your own ideas on how and where creation patterns might be appropriate. Remember—it's a diagram and this is brainstorming. It isn't an exam with definitive right answers. It's normal and okay to be wrong or unsure sometimes. In the next section, Tom will present his diagram containing all the patterns he intends to use for the project. If you'd like to try to figure out which ones are the most appropriate, pause here and make yourself a list. Beware of the Golden Hammer! It is not necessary, nor even desirable to try to use every pattern you know in one project.

The first design meeting

Tom configured his computer's display to share with a large 72" OLED 8K screen in the Bumble Bikes conference room. The lights were dimmed. Phoebe, true to character, made popcorn and passed around fizzy water. With everyone seated comfortably, Tom began his presentation by discussing the base object structure. Once the girls were up to speed on the basics, Tom directed the discussion toward creation patterns. As the sisters listened to Tom, it was clear that they held him in high esteem. They appreciated his abilities and the manner in which he adapted to the life fate had dealt him. The sisters were aware that most people are very uncomfortable interacting with someone in a wheelchair, especially if they also have impaired speech. They knew Tom desired to be seen for himself, and the sisters treated Tom as they did any other colleague. The sisters snapped from their collective reverie as Tom began his presentation.

"The first pattern I want to use is straightforward. The Abstract Factory pattern is used when you want to make related families of objects. You used it when you wanted to manufacture your own parts for different types of bicycles. You have classes for bicycle frames for road and mountain bikes that are not in any way interchangeable in the real world, but their abstract structures are similar enough to follow common interfaces. We can do this same thing with wheelchairs in two places. First, we could have an Abstract Factory creating parts for the wheelchairs just like you do for the bicycles. But I wonder if we couldn't go an extra step and use the Abstract Factory pattern to de-

termine whether we are generating a bicycle or a wheelchair at the highest level of your factory automation system. The clients of Factory patterns don't care about the structure of the object they're getting from the factory. The manufacturing system, therefore, shouldn't care whether it's making a bicycle or a wheelchair."

"That's true, Tom," said Kitty. "But I've found the Builder pattern to be more flexible. While Abstract Factory returns the produced object immediately, the Builder pattern allows you to attach extra steps in the build process."

"Hey!" Tom interjected. "If we use the Builder, we can combine some of the other Creational patterns as part of the build process. You used the Composite pattern to generate a cost model. You also used the Bridge pattern to handle bicycle paint jobs. I think that would be a big deal with wheelchairs. I work with a lot of kids at the hospital, and I know they would rather have crazy colorful paint jobs on their chairs than the standard black and chrome all the other companies make. The Builder could generate a bridged object as well."

"Couldn't you then make the director class for your builder a singleton?" Kitty asked.

Phoebe was sitting at the end of the table eating her popcorn. She leaned back and rolled her eyes. Kitty and Tom were in full-on geek

mode.

“This is going to require another pass at my design,” Tom said. He remembered the age-old phrase, “No battle plan survives first contact with the enemy.” He hardly regarded Kitty and Phoebe as his enemies, but he wasn’t expecting his preliminary design drawings to be considered final or complete. This is the first iteration. Tom said, “*I* need to reorganize all the components so that they’ll fit with the Composite pattern. I think that will probably change a lot of things.” It was late, so the team adjourned for the night.

The second pass

Tom was excited the next morning. He was at the lab early and went straight to work refining the diagram. All totaled, he felt he could use five of the patterns Kitty and Phoebe used in the bicycle project, as follows:

1. **Creational patterns**:
    1. **Builder**: Tom can leverage this pattern to handle the complicated object creation needed for a wheelchair class, including handling the composite and bridge implementations
    2. **Singleton**: Tom considers the builder class might be made into a singleton to save resource.
2. **Structural patterns**:

    1. **Composite**: Tom will create a cost model similar to the one created for the bicycles, but this time it will be integrated directly into the structure of the wheelchair classes instead of being created separately.
    2. **Bridge**: As with the bicycles, Tom uses the bridge to vary complexity in the paint jobs and the wheelchair classes independently.
3. **Behavioral patterns**:
    1. **Command**: The Command pattern will be used to bundle all the details needed to create a wheelchair into a single object that can then be sent to a receiver class

Tom set up the second design meeting with Kitty and Phoebe, which was accepted with less fanfare. The trio huddled around the conference room display and Tom presented his diagram.

The Builder pattern

Tom wasted no time getting into the weeds. “We talked about the possibilities of using the Builder pattern. I think it’s going to work really well,” Tom said.

Deftly moving the pointer around the screen with his big toe, he added the classes needed to implement the Builder pattern. You can see the result here:

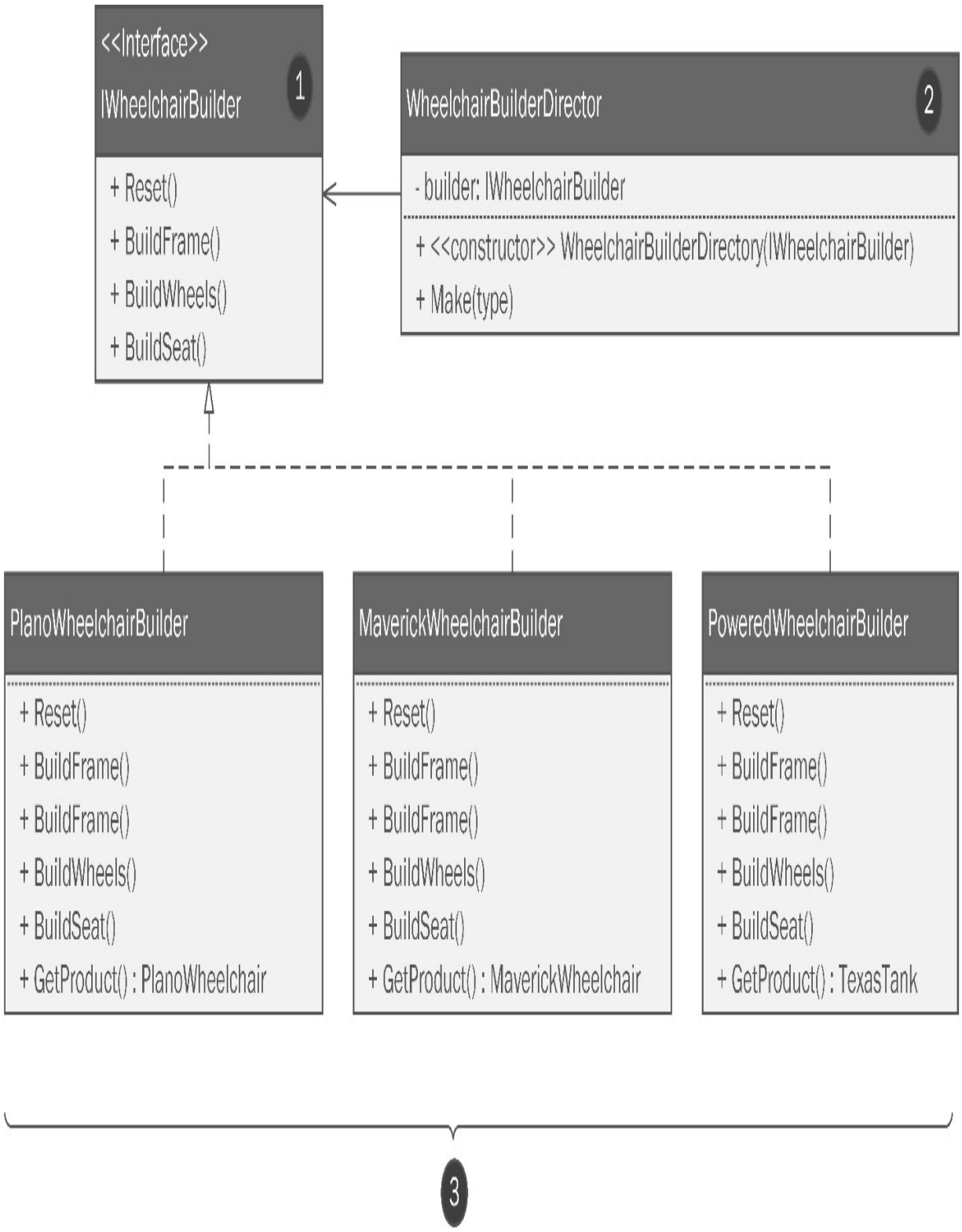


Figure 6.17: Tom has added the structures required for the Builder pattern

We covered the parts of the pattern in detail in *Chapter 3*, *Getting Creative with Creational Patterns*, but let's quickly review this:

1. The builder is defined by an interface or an abstract class. Here, that will be **`IWheelchairBuilder`**.
2. The builder is controlled directly by a **`WheelchairBuilderDirector`** class that contains a private instance of the builder interface. All the work is done by the director.
3. Concrete instances of builders are defined for each product. In this case, each wheelchair model gets its own builder, giving us great flexibility given the inherent differences between the models.

"This is a good start," Kitty said. "We also talked about making the builder a singleton."

"Right!" said Tom.

The Singleton pattern

Tom replied, "There's not much to say about this one." Tom made two changes to the **`WheelchairBuilderDirector`** class in the diagram. You can see the changes here:

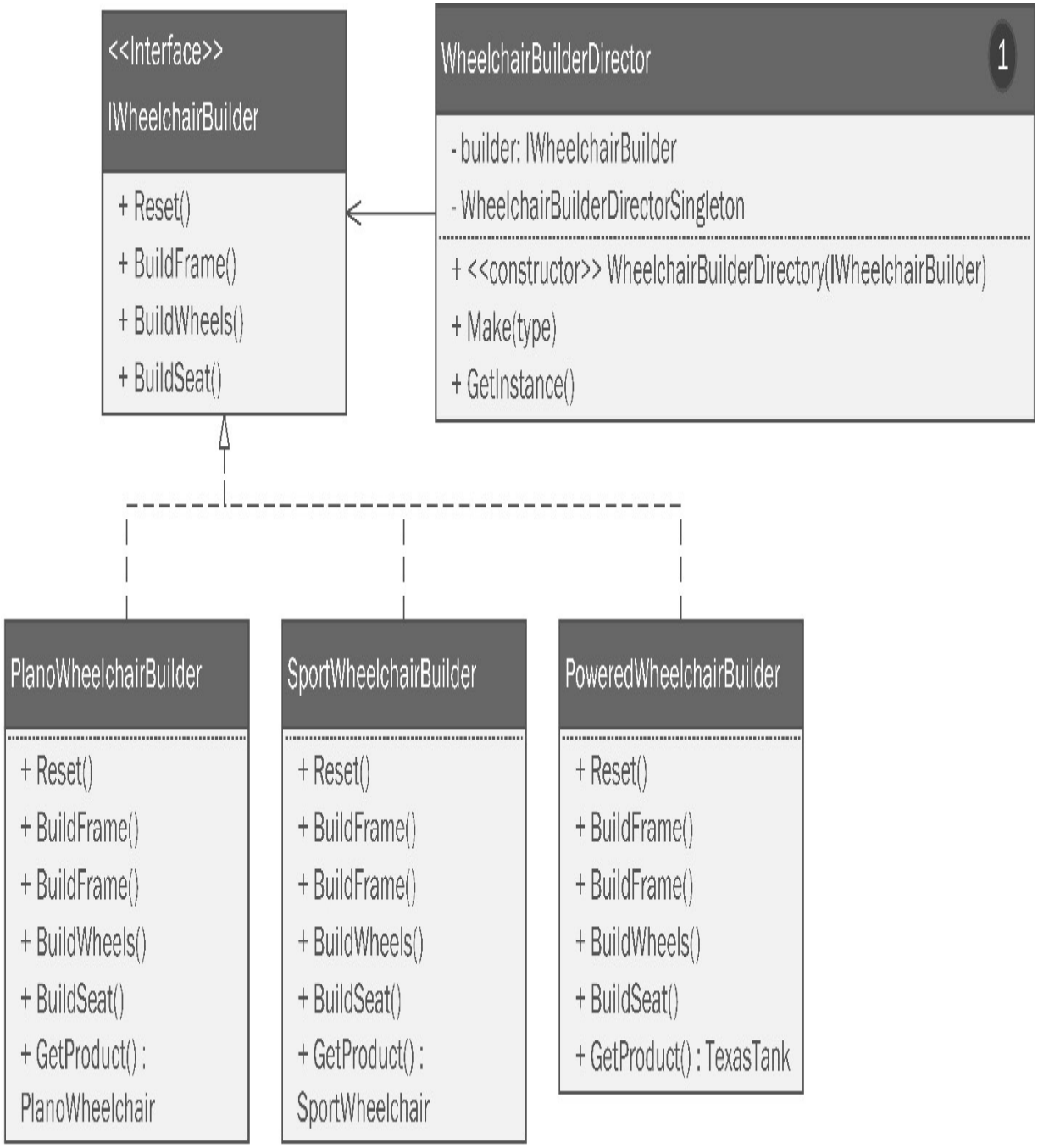


Figure 6.18: The structure for the Builder pattern used for wheelchairs has been made into a Singleton. You can tell since there's only a single number on the diagram

“To make this builder a Singleton, I just added the internal reference **`WheelchairBuilderSingleton`** (1). This will hold an instance of **`WheelchairBuilderDirector`** if the object has already been instantiated. Then, I added the **`GetInstance()`** method. This method will check to see whether there is an instance already in **`WheelchairBuilderSingleton`**. If it’s empty, the method creates an instance, stores it in that private variable, and returns the instance. If there’s already an instance there, it returns that one. I’m not sure we get anything from making it a singleton. Usually, singletons work best in conjunction with an object pool for expensive instantiations such as database connections. I’m not sure we have that here, but we can decide when we write it. Taking it out isn’t a big deal.”

The girls nodded their heads in understanding. They were both waiting anxiously to see what Tom did with the Composite pattern, which was next on the agenda.

The Composite pattern

“I really liked the way you used the Composite pattern with your bicycle models. You were able to leverage the pattern to compute component cost and weight for shipping. But you modeled a special structure just for that purpose,” said Tom.

Kitty fielded the criticism, saying, “A separate structure met our needs at the time we built it. We already had our classes in production code and we didn’t want to violate the open-closed principle of SOLID.” We covered SOLID in *Chapter 2*.

Tom wanted to build a Composite pattern directly into the structure of the wheelchair. This was a little bit challenging because he didn’t necessarily want the structural requirements of the composite to define the structure of the wheelchair. Instead, what he really needed was a normal object structure that acted as a composite. “We can improve on your design by baking the composite structure directly into all our objects. We need to think about the wheelchair assembly as a hierarchy. I see it as looking like this,” Tom said. He wheeled over to a nearby whiteboard. The girls had set one up on a special stand so that Tom could reach it. He sketched out a hierarchical diagram showing how the parts of an unpowered chair might be modeled. Once he had drawn it, Kitty grabbed a marker. “That’s good, but the powered chair would look a little different,” she said while sketching out her own idea for a hierarchical powered chair model. The whiteboard looked like this:

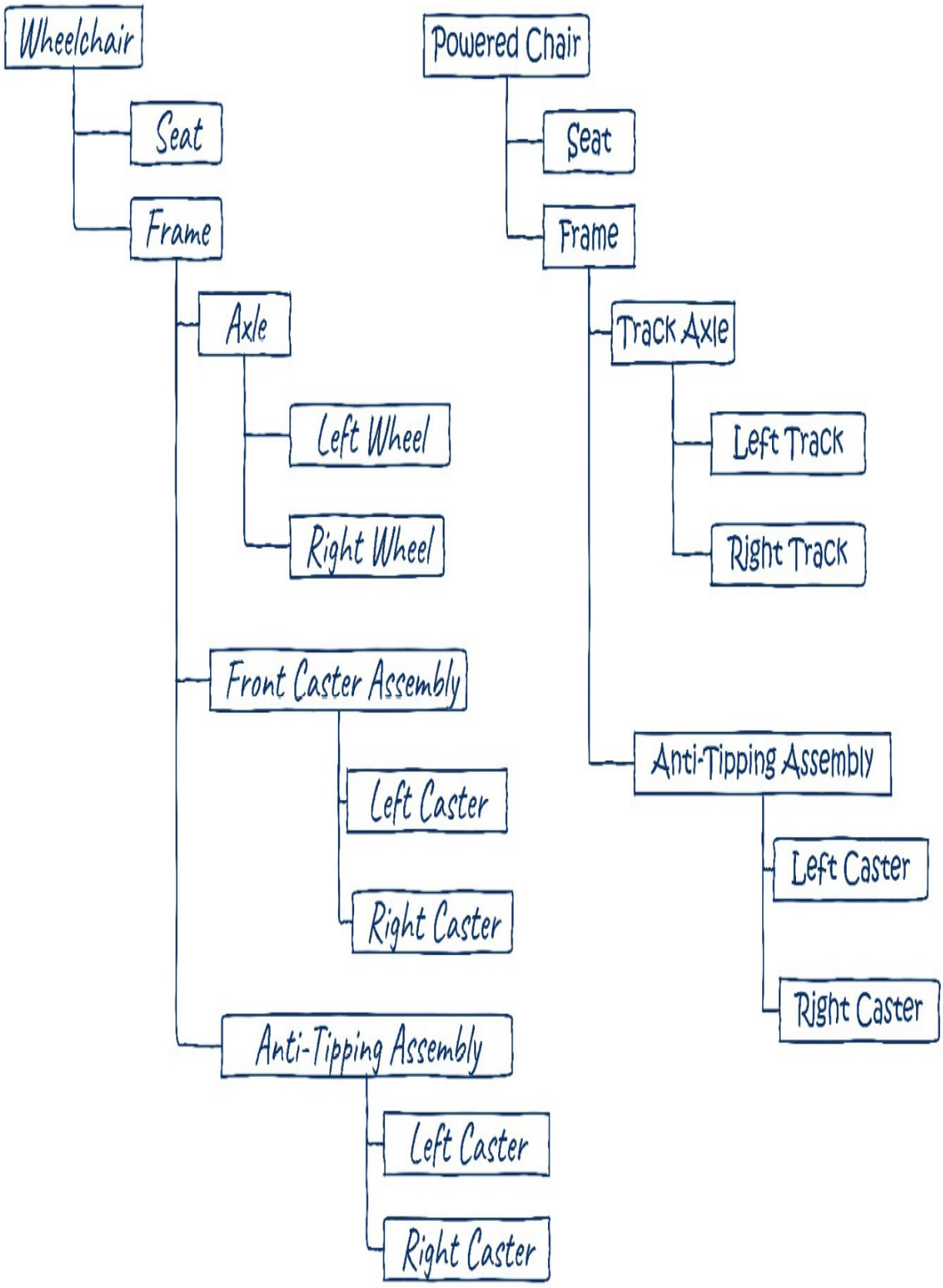

Wheelchair
Seat
Frame
Axle
Left Wheel
Right Wheel
Front Caster Assembly
Left Caster
Right Caster
Anti-Tipping Assembly
Left Caster
Right Caster
Powered Chair
Seat
Frame
Track Axle
Left Track
Right Track
Anti-Tipping Assembly
Left Caster
Right Caster

Figure 6.19: Tom and Kitty were each able to draw a powered and unpowered wheelchair as an object hierarchy to make it compatible with the Composite pattern

Shifting the group's focus back to the computer monitor, Tom continued. "With this structure in mind, we need to overhaul the classes that specify the construction of the frame. Instead of each piece being constructed and assembled as one piece, the robots would need to assemble a collection of parts in order. Structuring this way allows accounting to meticulously track the weight and cost of each wheelchair down to the smallest component. With the ability to track weight and cost, the sales department will be able to price the chairs competitively and work with insurance companies and charities to make sure the chairs can find their way into people's lives who need them without bankrupting the company." Kitty and Phoebe continued staring at the screen, barely blinking.

He continued, "The outsourced logistics team at *ExFed* can use the weight calculations to guarantee accurate shipping quotes. In the bicycle project, you constructed a separate object to house your composite structure."

"What I'll do for this project is hide the composite in the normal structure," Tom said as he zoomed into the appropriate part of the diagram. He dragged a new class to the page and titled it

`WheelchairComponent`. He then italicized it and explained why he added the members needed for his Composite pattern idea. You can see this idea here:

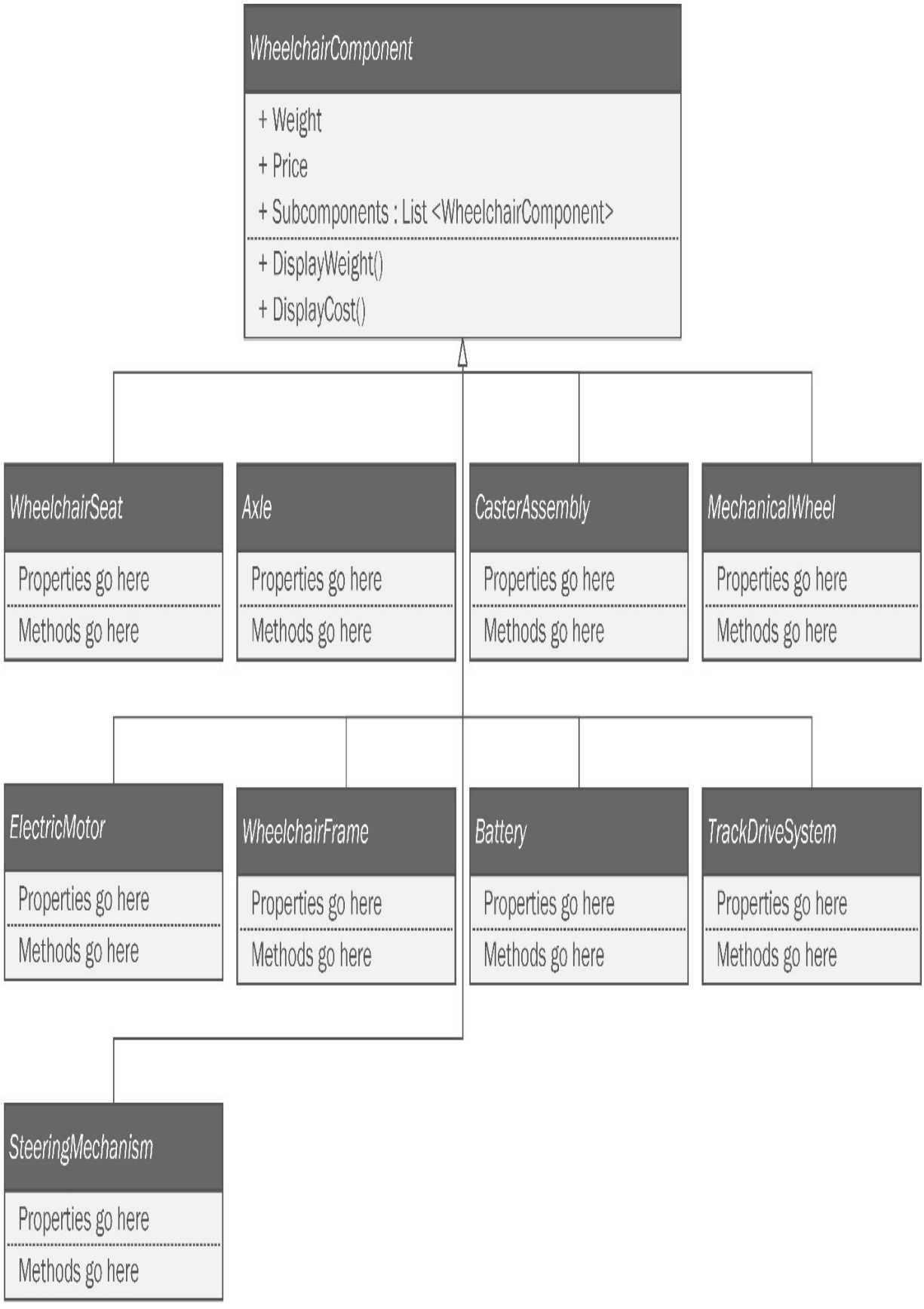

WheelchairComponent
+ Weight
+ Price
+ Subcomponents : List <WheelchairComponent>
+ DisplayWeight()
+ DisplayCost()
WheelchairSeat
Properties go here
Methods go here
Axle
Properties go here
Methods go here
CasterAssembly
Properties go here
Methods go here
MechanicalWheel
Properties go here
Methods go here
ElectricMotor
Properties go here
Methods go here
WheelchairFrame
Properties go here
Methods go here
Battery
Properties go here
Methods go here
TrackDriveSystem
Properties go here
Methods go here
SteeringMechanism
Properties go here
Methods go here

Figure 6.20: Tom had to add and refactor a lot of classes to have it fit the Composite pattern without necessarily looking like a composite

“What I did here was to add a **`WheelchairComponent`** abstract class. I usually do this instead of adding an interface when I need to add methods with implementations in them. In this case, we’ll have the **`DisplayWeight()`** and **`DisplayCost()`** methods, which will recursively process the subcomponents to arrive at totals for weight and cost. Then, I modeled all the components for the wheelchairs as abstract components in case we ever need to swap out concrete versions of those components.” Tom created a separate diagram to show this idea. Experience has taught Tom, who is now teaching this best practice to Kitty and Phoebe, that the best diagrams are small, focused, and concise. Instead of spreading out the classes in the diagram and appending subclasses beneath each abstract class, Tom made a diagram that just shows the inheritance chain for the seating components. You can see this diagram here:

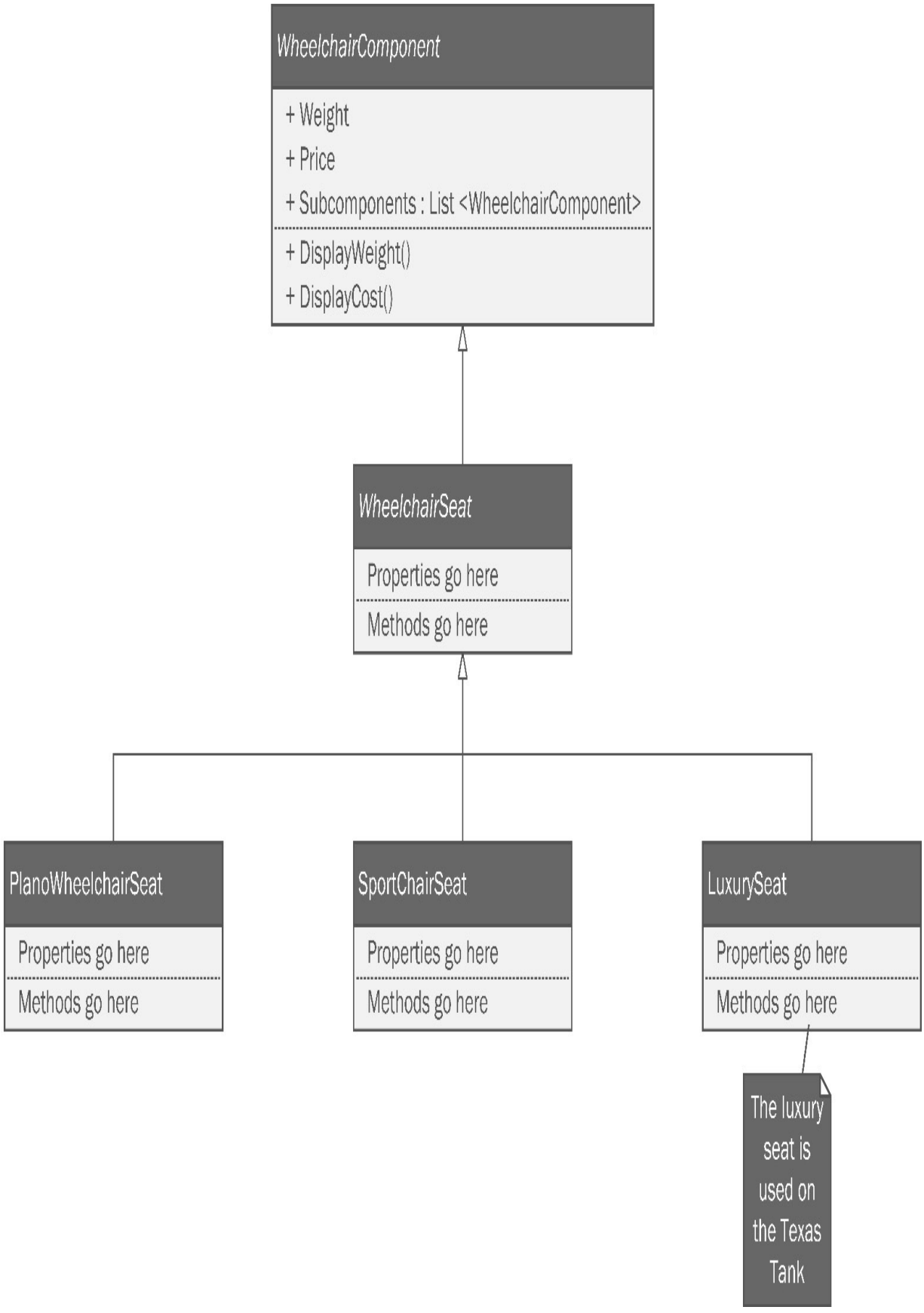

WheelchairComponent
+ Weight
+ Price
+ Subcomponents : List <WheelchairComponent>
+ DisplayWeight()
+ DisplayCost()
WheelchairSeat
Properties go here
Methods go here
PlanoWheelchairSeat
Properties go here
Methods go here
SportChairSeat
Properties go here
Methods go here
LuxurySeat
Properties go here
Methods go here
The luxury seat is used on the Texas Tank

Figure 6.21: The concrete seat objects inherit from the abstract WheelChairSeat class, which inherits from WheelchairComponent

Tom looked over at Phoebe. Clearly, the gears were turning in her mind. She finally spoke, saying, "What we're doing here is have everything inherit from **`WheelchairComponent`**. The **`WheelchairSeat`** abstract class is there to make sure the concrete seat objects can be represented in the composite structure using Liskov substitution. Any seat will have the structure of the parent classes."

"That's right," Tom said. Kitty continued her sister's thought. "We can put any seat on a frame and still be able to have weight and price. That means as long as we assemble the hierarchy in the right order,"—she pointed back to the whiteboard (*Figure 6.18*) as she spoke—"we'll be able to have the benefits of the Composite pattern and we will also have a flexible way to compose our wheelchairs." Tom nodded in agreement. "Think of it this way: it's like basic composition, which might be why the GoF called this the *Composite* pattern, but it is composition done a different way. It is just as flexible, but it is also iterable because the composition uses lists to hold the parts."

Phoebe wrinkled her nose. "Should we worry about the fact this structure might make it possible to add 200 seats to a wheelchair?"

“No,” said Tom. “You can handle that with encapsulation logic. Each class can impose rules on the list of components it contains. Liskov substitution makes the structure flexible, which gives you the power to construct anything. But you need to temper this by added encapsulation methods that control what goes into the composition.” This time it was Kitty’s face that was scrunched up like she’d smelled something awful. “This seems very complicated for the developer that has to use this method. They’ll have to remember to assemble the object in the right order. They’ll have to look back at the diagram (*Figure 6.18*) anytime they want to build a wheelchair.”

“Not so,” said Tom. “Remember the Builder pattern we are using to direct the assembly of the object?” Both the girls’ faces relaxed. They instantly got it. “The patterns can be used together! The builder’s job, its *raison d’être*, is to handle the construction of complex objects using a well-defined set of steps.” “We’ll need to alter our builder class diagram to account for this change,” said Kitty. Phoebe looked alarmed. “Aren’t we violating the open-closed principle? We’ve already designed that class!” Kitty rolled her eyes as Tom grinned. “No, Phoebe. It’s just a diagram. It’s not production code!” At that moment, Phoebe fully understood why Tom had chosen to diagram his work before he committed it to code. At a master’s keyboard, coding becomes a discipline. We need a way to relax the rules so that we can at least have a chance at getting our design right the first time.

“The parent **`WheelchairComponent`** class has the **`Subcomponents`** list in it, but I’m thinking of hiding it. For now, it’s public, but my idea is to have the builder add the appropriate subcomponents to the array at build time. This means I can have a traversable component tree without forcing my whole structure to conform to a common interface, which is probably impossible,” said Tom.

“Hey, that’s pretty neat!” Kitty said enthusiastically. “Can we deal with the changes to the Builder pattern really quickly?” Kitty asked. The trio brought up the diagram they had created in *Figure 6.16* and began to modify it. After a few minutes, it looked like this:

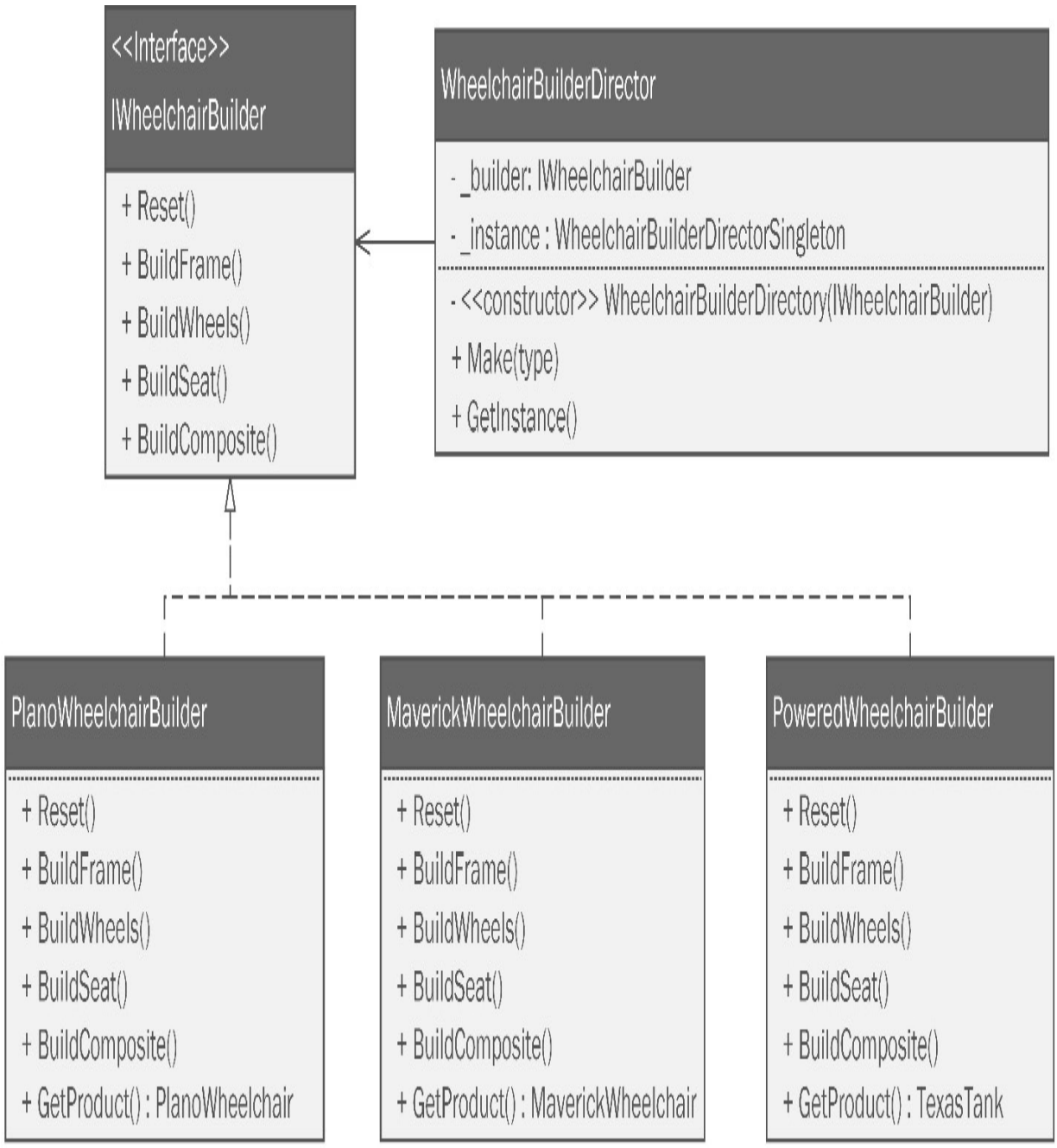


Figure 6.22: The Builder pattern with the addition of methods to build the composite

“That’s it?” asked Phoebe. “What about all that fancy encapsulation logic we talked about? Where is that on the diagram?” Phoebe

pressed. “The logic will be the `Make` method in the `WheelchairBuilderDirector` class. Remember, it is the director class that is responsible for the logic that assembles and returns the object.”

Kitty and Phoebe seemed enthusiastic about Tom’s work. He moved on to the next pattern.

The Bridge pattern

“In our last meeting, we discussed reusing the Bridge pattern work from the bicycle project. I think having paintable frames will be popular because most companies just make black or gray wheelchairs. That’s fine for hospital loaners, but if you’re a lifer like me, after a while you’re ready to upgrade.” Tom smiled. His chair had a little bit of red paint on a few parts of the frame, but it was far from *upgraded*.

Phoebe said, “With our bicycles, one of our Kickstarter backers designed our first custom paint job. Maybe you should come up with one for the wheelchairs.”

Tom flushed. “I’m honored, but really, I’d rather bring in a bunch of the kids from the hospital and see about making some designs they like.” Kitty smiled warmly. Phoebe’s respect for Tom deepened.

Changing the subject, Tom once again focused on his screen. “I didn’t change anything from the bicycle project’s implementation except for adding the concrete painter classes.” Kitty and Phoebe nodded, glad that Tom was going to be able to reuse some of their earlier work. “Of course, I also added a **`FramePainter`** attribute to the abstract **`Wheelchair`** class so that all the subclasses will have it,” Tom concluded. You can see the UML diagram for the Bridge pattern applied to the wheelchair project here:

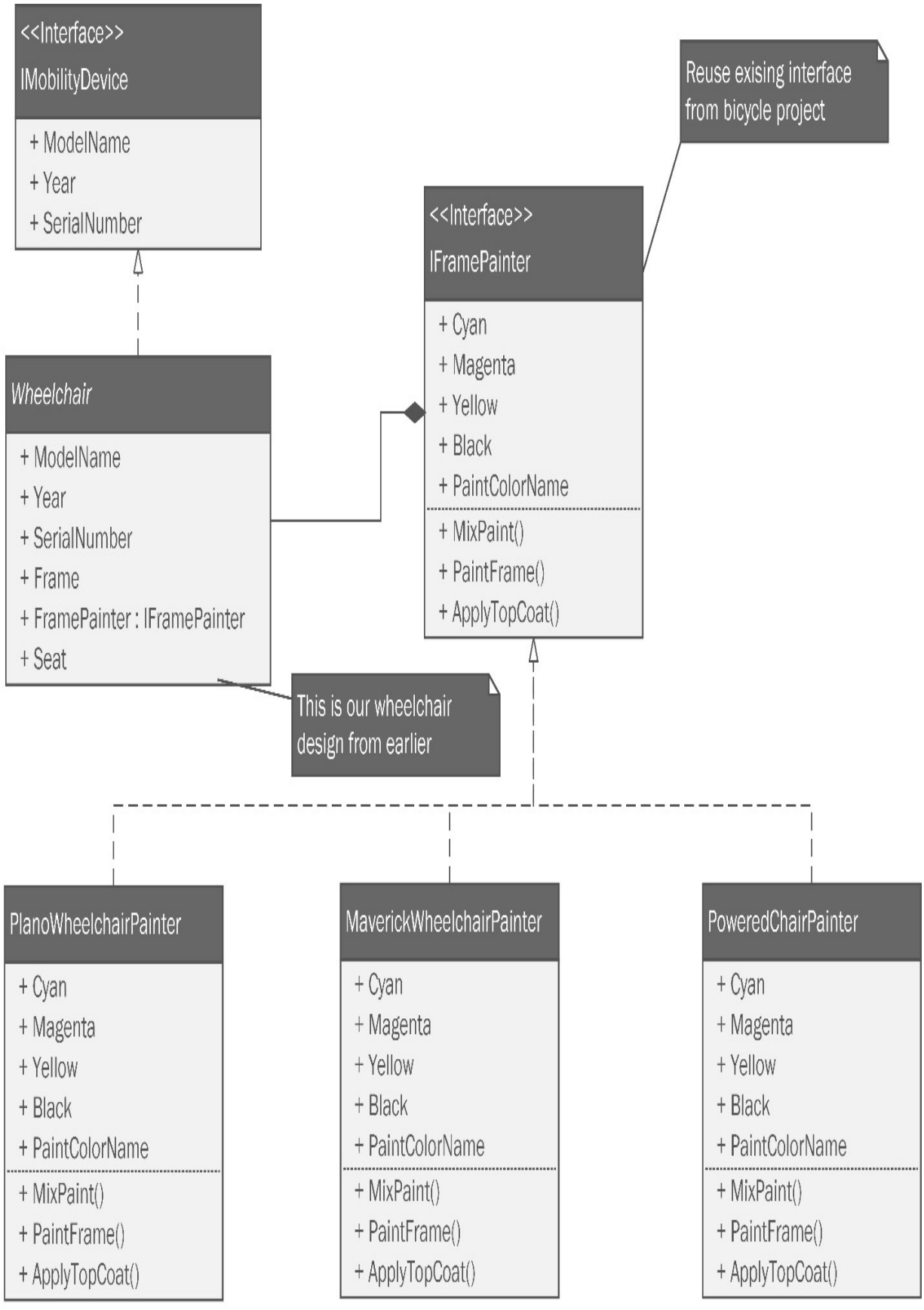

<<Interface>>
IMobilityDevice
+ ModelName
+ Year
+ SerialNumber
Reuse exising interface from bicycle project
<<Interface>>
IFramePainter
+ Cyan
+ Magenta
+ Yellow
+ Black
+ PaintColorName
+ MixPaint()
+ PaintFrame()
+ ApplyTopCoat()
Wheelchair
+ ModelName
+ Year
+ SerialNumber
+ Frame
+ FramePainter : IFramePainter
+ Seat
This is our wheelchair design from earlier
PlanoWheelchairPainter
+ Cyan
+ Magenta
+ Yellow
+ Black
+ PaintColorName
+ MixPaint()
+ PaintFrame()
+ ApplyTopCoat()
MaverickWheelchairPainter
+ Cyan
+ Magenta
+ Yellow
+ Black
+ PaintColorName
+ MixPaint()
+ PaintFrame()
+ ApplyTopCoat()
PoweredChairPainter
+ Cyan
+ Magenta
+ Yellow
+ Black
+ PaintColorName
+ MixPaint()
+ PaintFrame()
+ ApplyTopCoat()

Figure 6.23: The Bridge pattern is generally unchanged save for the concrete classes

“Are the builder classes going to be directly adding the paint to the wheelchair objects?” Kitty asked. Tom said, “I’m glad you reminded me. I would have forgotten to add that to the builder diagrams.” Tom made a quick modification to the builder diagram we last saw in *Figure 6.22*. The new version can be seen in *Figure 6.24*:

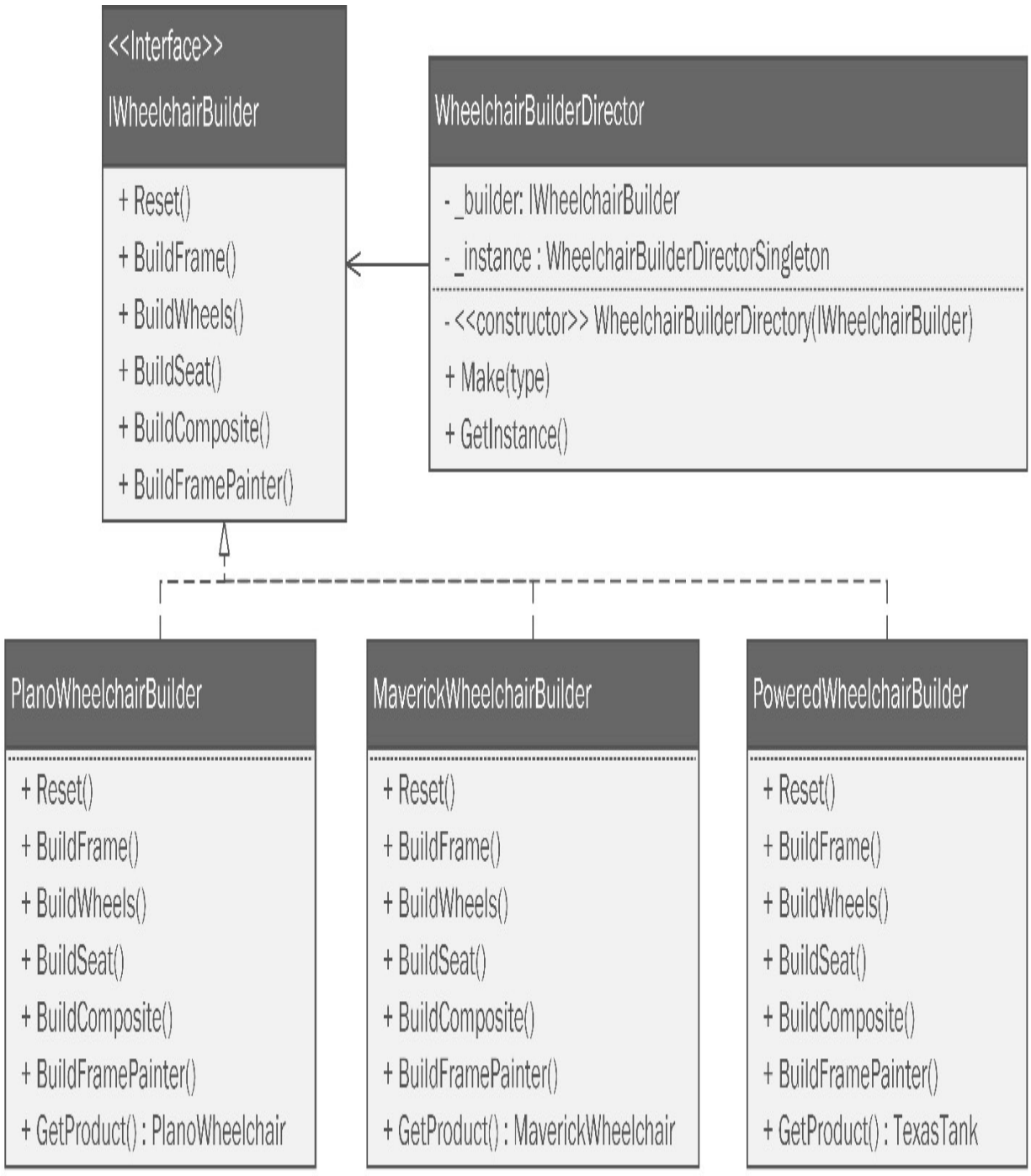


Figure 6.24: Builder pattern diagram revised with the addition of the BuildFramePainter() method

Next on the list was the Command pattern.

The Command pattern

"*There is one last refactor I want to make. You used the Command pattern to direct the manufacturing robotics to build your bicycle. Naturally, I want to use the same manufacturing system. I think the differences can be easily handled,"* Tom said.

"OK," Phoebe said quizzically with one raised eyebrow. "*What are you doing to my babies?*"

Tom dragged some classes to the diagram and arranged them to form the Command pattern. Once he got everything filled in, his diagram looked like this:

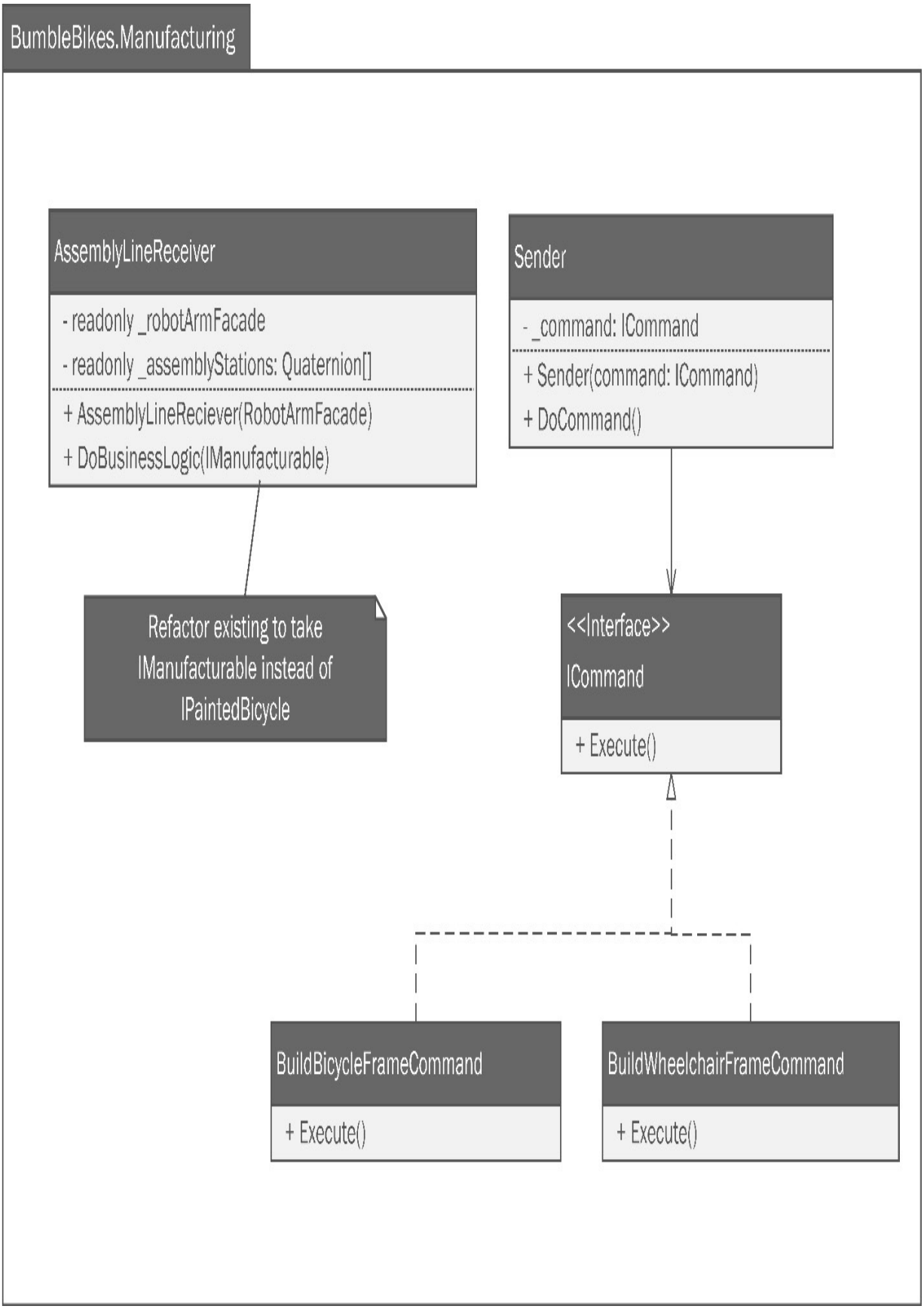

BumbleBikes.Manufacturing
AssemblyLineReceiver
- readonly _robotArmFacade
- readonly _assemblyStations: Quaternion[]
+ AssemblyLineReciever(RobotArmFacade)
+ DoBusinessLogic(IManufacturable)
Sender
- _command: ICommand
+ Sender(command: ICommand)
+ DoCommand()
Refactor existing to take IManufacturable instead of IPaintedBicycle
<<Interface>>
ICommand
+ Execute()
BuildBicycleFrameCommand
+ Execute()
BuildWheelchairFrameCommand
+ Execute()

Figure 6.25: Tom's change to the Command pattern is fairly minor

Tom said, "*We need to make a change to the DoBusinessLogic method in AssemblyLineReceiver. You have it set up to only take one interface: the IPaintableBike interface. I can't force-fit my wheelchair to be a paintable bike. We could change the central interface for my project from IMobilityDevice to IManufacturable.*" He made the change as he was talking. Tom continued, "*This way, the interface has elements common to bicycles and wheelchairs, and probably anything else you'd ever want to make.*" He finished speaking as he added a new member variable to the **`IManufacturable`** interface: **`ManufacturingStatus`**. This was in the **`PaintableBicycle`** class, but to this point, Tom hadn't thought to put it in the diagram. You can review the revisions here:

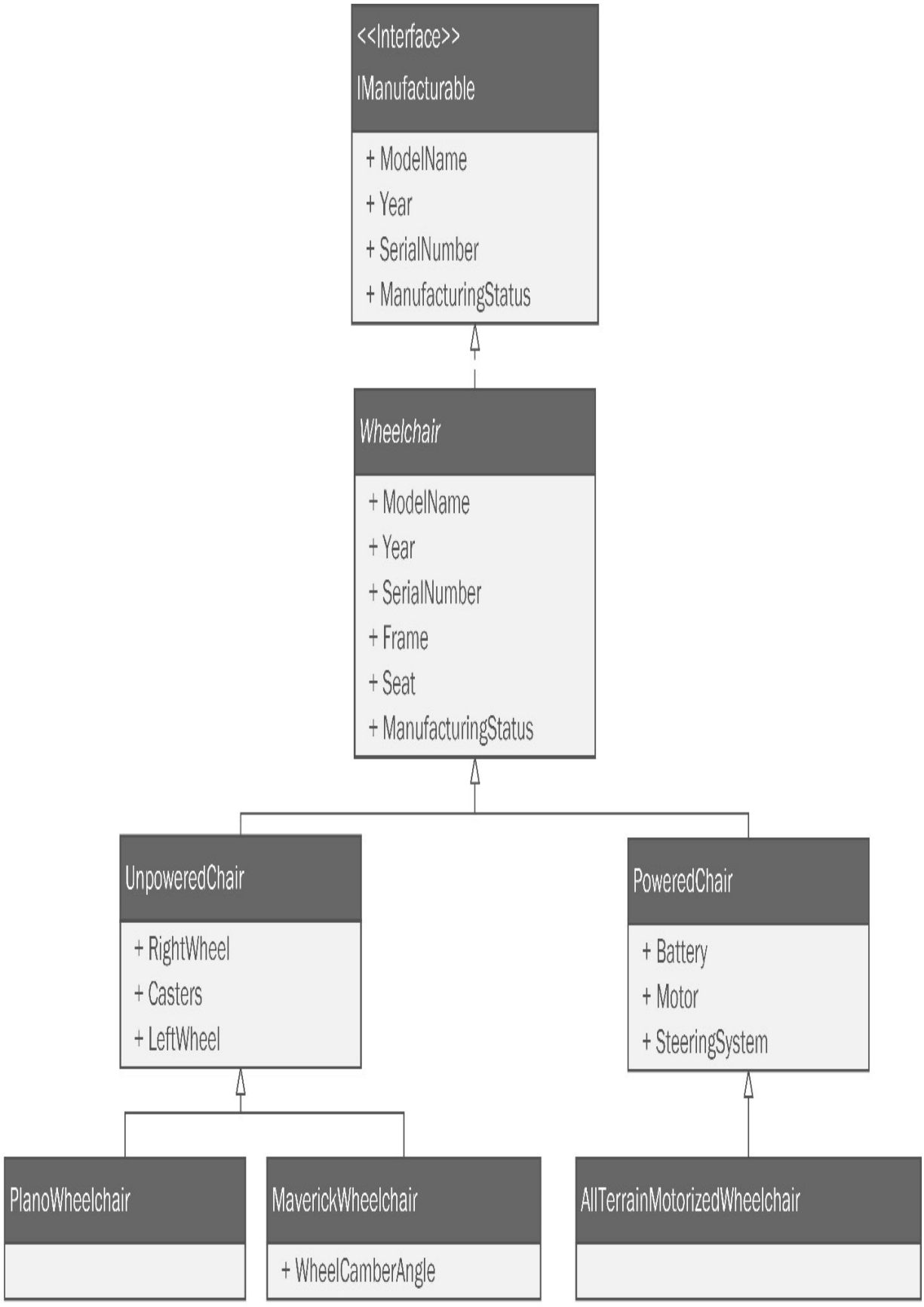

<<Interface>>
IManufacturable
+ ModelName
+ Year
+ SerialNumber
+ ManufacturingStatus
Wheelchair
+ ModelName
+ Year
+ SerialNumber
+ Frame
+ Seat
+ ManufacturingStatus
UnpoweredChair
+ RightWheel
+ Casters
+ LeftWheel
PoweredChair
+ Battery
+ Motor
+ SteeringSystem
PlanoWheelchair
MaverickWheelchair
+ WheelCamberAngle
AllTerrainMotorizedWheelchair

Figure 6.26: The final change renames the central interface IManufacturable.

For the first time during the presentation, Phoebe was standing up. She stared blankly for a moment. “I don’t know about that,” she said. She didn’t say another word for the rest of the meeting.

Kitty picked up the slack. “Tom, this looks great! I can see Phoebe has some trepidation about your last refactor, but we can work through that. Let’s get started on the base classes for your project right away.” Kitty and Tom spent the rest of the afternoon setting up a repository on GitHub and configuring a **continuous integration** (**CI**) build for the new project. Having pushed a trivial class and watched it build, Tom added a simple unit test to the class. It too passed. It was about this time when the three realized it was very late at night, or very early in the morning, depending on your point of view. Though they didn’t really want to, they reluctantly broke for the evening (or morning) and resolved to get started the next day at the crack of noon.

Summary

This chapter presented a design scenario where we saw a drastic change in business requirements for our nascent bicycle company as

they made the brave decision to branch out and make wheelchairs to help those in need. As usually happens with these things, this change was not planned and happened at an inopportune moment. Strapped for time and inspiration, Kitty and Phoebe hired Tom, a wheelchair-bound software engineer who found himself aggressively unemployed following an executive meltdown at his previous employer.

Tom leveraged his expertise in software development, his personal experience, and his experience as a volunteer helping recently injured patients learn to cope with living in a wheelchair. He was able to bring a fresh perspective to the problem and began to design a solution that fits the manufacturing environment as well as the very real cost and resource constraints already extant at Bumble Bikes.

After about a week of analysis and design using UML, Tom was able to plan his development effort leading up to the first release of the new wheelchair project.

Tom did a number of things differently than the girls did on the bicycle project. Mainly, he designed with patterns and UML first, instead of trying to find patterns as he went. Hopefully, this will lead to fewer instances where we need to make a drastic change to the code that might violate our sacrosanct SOLID principles. Tom was adamant that this practice would save a lot of time and frustration on the project.

Tom followed a process for developing his design diagrams as follows:

1. In the first pass, he designed all the base classes upfront. He just drew the classes and the inheritance lines.
2. Once he had the classes, he filled in the inheritance and composition lines.
3. He thought about patterns, but didn't diagram them. Instead, he held a meeting with his development team who had experience and insight with the same set of patterns he was working with. Effectively, he was using iterative development, but with a diagram instead of code. He got input from his stakeholders before trying to finish the whole design project.
4. Tom, Phoebe, and Kitty as a team made a second pass and collaboratively added in the patterns.

The last consideration of Tom's performance was that he didn't feel compelled to use every single pattern available to him. He went out of his way to reuse work wherever he could. Kitty and Phoebe were far more impressed by this than if Tom had suggested a total rewrite, as many developers seem to want to do when they start at a new company.

In the next chapter, we'll follow Tom, Kitty, and Phoebe as they writes the code that implements his diagrams.

Questions

1. What are the advantages of diagramming a project instead of simply modeling everything in code?
2. What is the one time we definitely use an abstract class instead of an interface?
3. How did Tom leverage the Builder pattern to work with other patterns in the second pass of the design session?
4. Can you think of any improvements to Tom's design?

Further reading

Check out this book's companion website at https://csharppatterns.dev.

7

Nothing Left but the Typing – Implementing the Wheelchair Project

In the previous chapter, we learned about the advantages of creating a set of diagrams as a design plan for our next coding project. The whole point is to get the design in a format that can be discussed, argued, pondered, socialized, and changed. After the design is completed, the last step is to implement the diagrams as code. Our trio of software engineers has done just that with an ambitious new project designed to make a difference in the lives of potentially thousands of people who could benefit from access to a high-quality, low-cost wheelchair.

I've often compared UML with sheet music. A good UML design can be handed off to a developer the same way a musical composer can hand off a score to a competent orchestra. In music, the orchestra will often make changes and improvisational improvements to the sheet music. Sometimes they do this to make the music fit the skill of the performers. Other times, they might need to change or adapt the mu-

sic to fit the performance itself. Classical composer Johann Sebastian Bach was prolific. Among his most popular works are *The Brandenburg Concertos*. Bach assembled the collection in 1721. In 1968, a different composer, Wendy Carlos, arranged *The Brandenburg Concertos* to be played entirely on analog synthesizers in her work *Switched on Bach*. The music didn't change as much as the implementation details. It isn't a stretch to think of programming as being a lot like playing music. It requires creativity and improvisation. A talented architect who has created a good diagram can be reasonably sure a competent development team will be able to implement the diagram and, if required, make some improvisations.

In this chapter, we are going to hand off Tom's UML design diagrams for implementation. There is a big difference between designing the classes and implementing them. First, the diagrams leave some areas vague on purpose. This is done to allow the developer to make decisions about details that don't really matter to the diagram. As an example, there are many diagrams in this book that leave out class details. Look at a diagram from the previous chapter in *Figure 7.1*:

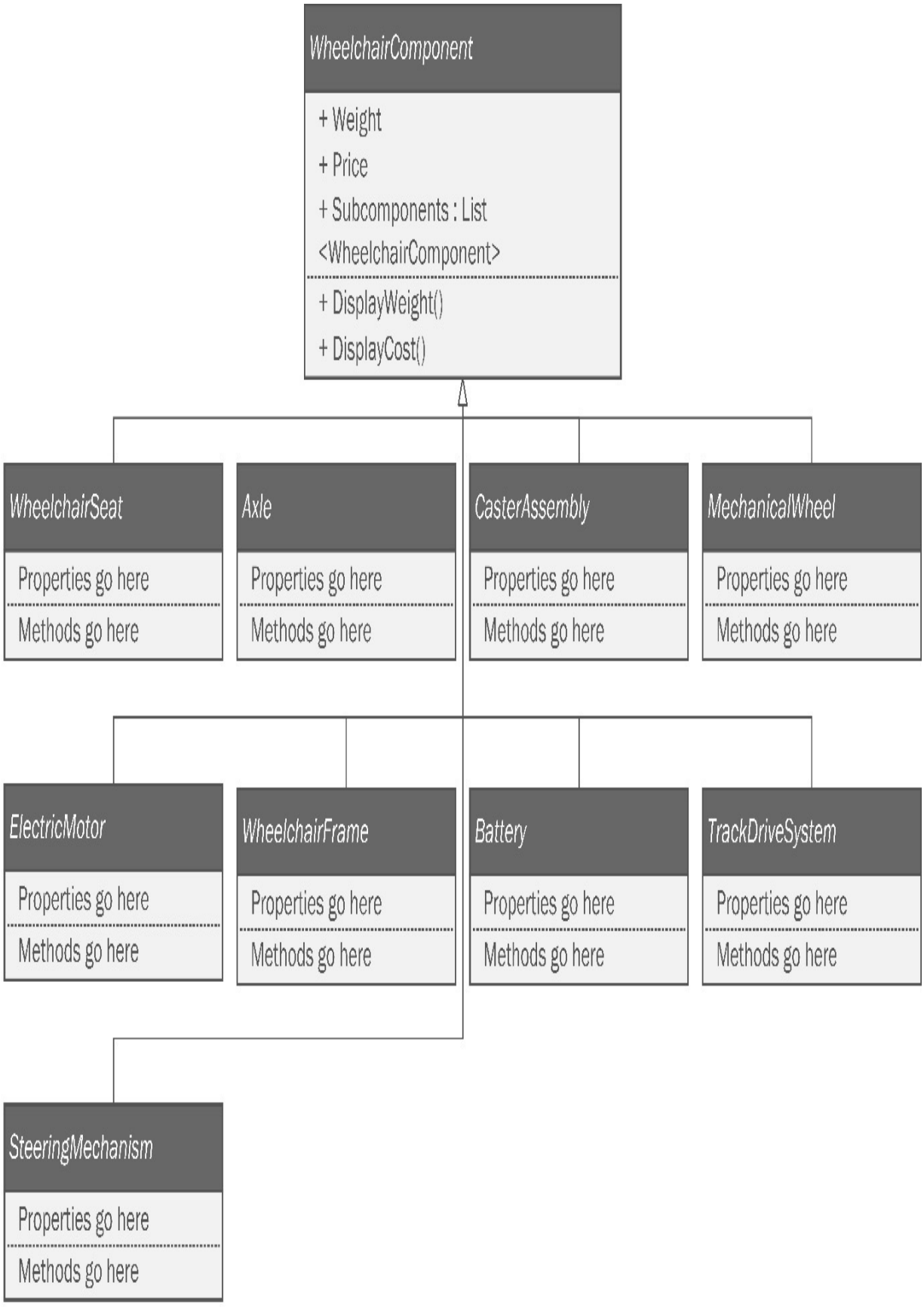

WheelchairComponent
+ Weight
+ Price
+ Subcomponents : List
<WheelchairComponent>
+ DisplayWeight()
+ DisplayCost()
WheelchairSeat
Properties go here
Methods go here
Axle
Properties go here
Methods go here
CasterAssembly
Properties go here
Methods go here
MechanicalWheel
Properties go here
Methods go here
ElectricMotor
Properties go here
Methods go here
WheelchairFrame
Properties go here
Methods go here
Battery
Properties go here
Methods go here
TrackDriveSystem
Properties go here
Methods go here
SteeringMechanism
Properties go here
Methods go here

Figure 7.1: The composite diagram from the last chapter leaves a lot of details up to the developer.

This isn't done because the architect is lazy. It is done because those details don't really matter to the structure of the design. Imagine building a house from a blueprint. The blueprint will probably specify where the walls go. It might even specify the materials to be used, the same way we sometimes dictate a particular type for an instance variable or return type. However, the smaller details, such as what kind of paint to use on the walls or the brand of shingles used on the roof, are all implementation details. The builder can make those decisions independently of the blueprint.

While focusing on the design from *Chapter 6*, *Step Away from the IDE! Designing with Patterns Before You Code*, we'll cover the following:

- Creating a new command -line project. So far, I have stayed away from IDE mechanics. I won't be going into depth here either. It just feels right to start the chapter this way. If you're new to C# development, and you haven't noticed *Appendix 1* at the end of the book, you should check that out because I show you some of these mechanics that might not be second nature to you
- We'll start by adding our base classes just like Tom did, but we don't really need a two-pass system since we did all the organiza-

tion work in the design phase.
- We'll implement the Builder pattern.
- We'll convert the Builder pattern implementation to a Singleton.
- We'll implement the Composite pattern.
- We'll implement the Bridge pattern.
- We'll implement the Command pattern.

Throughout the book, I assume you know how to create new C# projects in your favorite **integrated development environment** (**IDE**). I do not usually spend any time on the mechanics of setting up and running projects. This chapter is a mild exception since the whole chapter really is an example of one big project. If you need more details, or you want to use an IDE other than Rider, and you're not sure how to set up the projects, please see *Appendix 1* of this book. Should you decide to try any of this out, you'll need the following:

- A computer running the Windows operating system. I'm using Windows 10. Since the projects are simple command-line projects, I'm pretty sure everything here would also work on a Mac or Linux, but I haven't tested the projects on those operating systems.
- A supported IDE such as Visual Studio, JetBrains Rider, or Visual Studio Code with C# extensions. I'm using Rider 2021.3.3.
- Any version of the .NET SDK. Again, the projects are simple enough that our code shouldn't be reliant on any particular version. I happen to be using the .NET Core 6 SDK.

You can find the completed project files for this chapter on GitHub at https://github.com/Kpackt/Real-World-Implementation-of-C-Design-Patterns/tree/main/chapter-7.

The crack of noon

Having spent the previous day and well into the night diagramming their project, Tom, Kitty, and Phoebe arrived at Bumble Bikes. At high noon the next day, the energy in the room was palpable.

Kitty walked in with a box under her arm. She had ordered a new keyboard, the kind with the loud blue clicky switches. She loved these keyboards because they reminded her of her father's IBM Model M keyboard. She used to play with it when she was little. Kitty wanted to remember the inspiration for this project.

A short time ago, she and her sister, Phoebe, had started a successful bicycle manufacturing company. Their outlook was optimistic until their father was diagnosed with a rare degenerative muscular disease called dermatomyositis and he was newly confined to a wheelchair. Kitty and Phoebe's mother had fought a long legal battle with their medical insurance provider, who refused to pay for the expensive wheelchair their father needed to cope with his disease. Phoebe took matters into her own hands and decided, nearly singlehandedly, that she could make a wheelchair for their father and everyone else

who needed one. Naturally, Kitty followed her younger sister's lead because she knew Phoebe was right. "*Today is the day,*" Kitty thought, *"we change the world."* The new keyboard would keep her focus on her father and all the people she could help just by doing a little typing.

Tom showed up to the lab wearing his finest nerdy T-shirt; it bore a famous XKCD cartoon featuring two developers sword fighting with the lettering on the shirt stating that Tom was not slacking off; he was merely waiting for his code to compile. Phoebe arrived with a stack of pizzas and a palette of fizzy water. The advantage to starting work at noon is that it's easy to get the supplies needed. As we all know, a programmer is merely a biological machine that converts pizza and caffeine into code.

The trio got set up and Phoebe turned off the harsh overhead fluorescent lights. After a quick scrum session, they decided on a part of the application where they would get started. With any luck, they could have this done in a few days and this code might be the most important code the sisters, with Tom's help, might ever write.

Setting up the project

Kitty, Phoebe, and Tom hunker down in front of Kitty's new keyboard. The three are intent on using pair programming. Pair programming

occurs when two or more developers work together and one keyboard is shared. One person types as the others watch. The developers trade positions every so often. The developers who are not typing are responsible for watching and helping with research. Pair programming negates the need for code reviews and is shown to dramatically increase developer productivity. If you're not familiar with the practice of pair programming, check out the book *Practical Remote Pair Programming* listed in the *Further reading* section of this chapter. Tom is intimately familiar with the design, but Kitty and Phoebe can type faster. Tom's ability to type with his toes is truly amazing, but he accepted a long time ago that typing speed is not a value he brings to a team. Kitty and Phoebe have far less coding experience, but having spent the last few years writing research papers at separate universities, their typing has become quite fast. Kitty decides to take the keyboard first while Phoebe fuels up on pizza and fizzy water. Kitty opens her favorite IDE and creates a new **Console Application** project as shown in *Figure 7.2*.

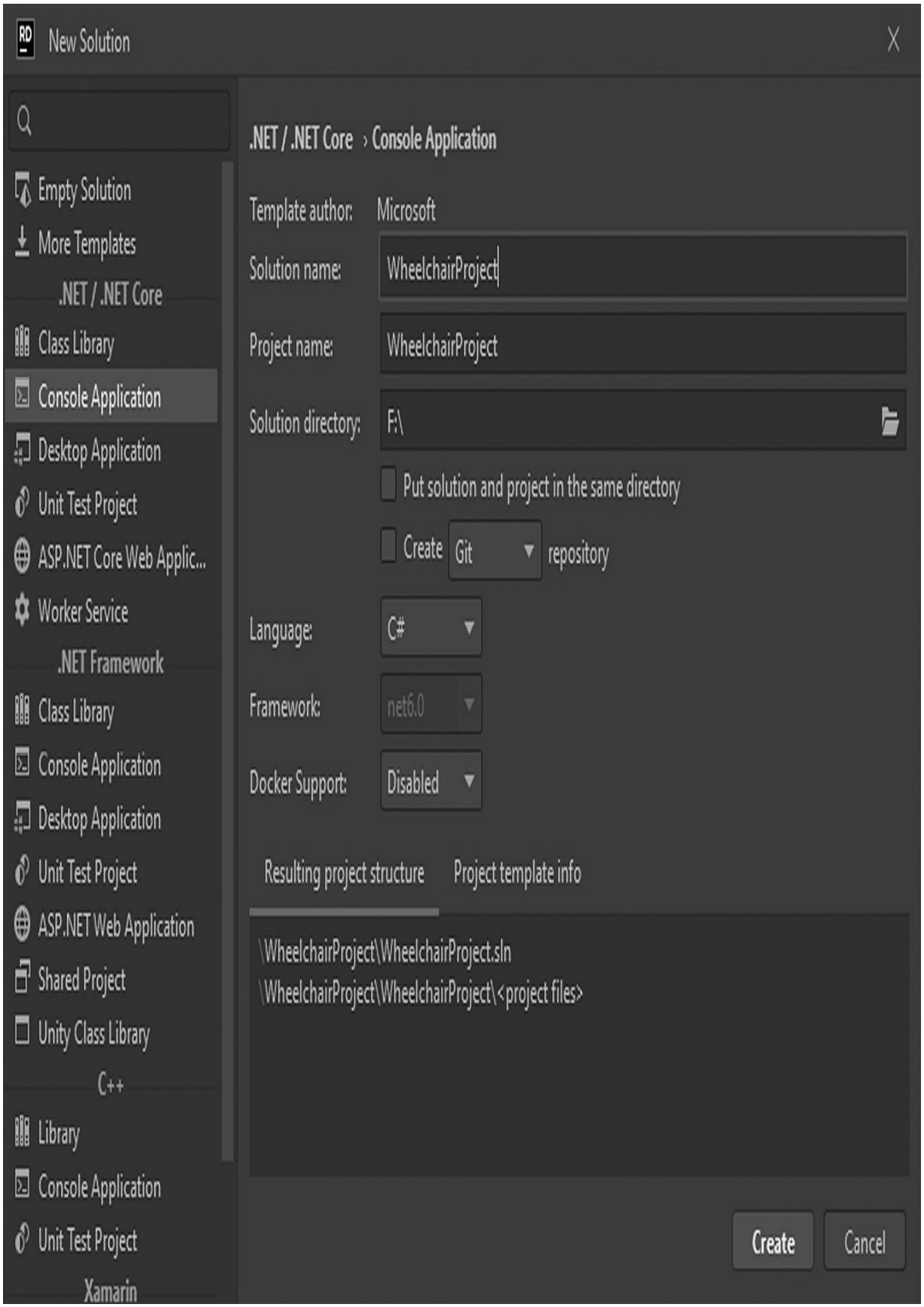

New Solution
Empty Solution
More Templates
.NET / .NET Core
Class Library
Console Application
Desktop Application
Unit Test Project
ASP.NET Core Web Applic...
Worker Service
.NET Framework
Class Library
Console Application
Desktop Application
Unit Test Project
ASP.NET Web Application
Shared Project
Unity Class Library
C++
Library
Console Application
Unit Test Project
Xamarin
.NET / .NET Core › Console Application
Template author: Microsoft
Solution name: WheelchairProject
Project name: WheelchairProject
Solution directory: F:\
Put solution and project in the same directory
Create Git repository
Language: C#
Framework: net6.0
Docker Support: Disabled
Resulting project structure
Project template info
\WheelchairProject\WheelchairProject.sln
\WheelchairProject\WheelchairProject\<project files>
Create
Cancel

Figure 7.2: Creating a new command-line project.

If you're following along with a different IDE, and you're not sure how to create a command-line project, check out *Appendix 1* of this book. In it, I cover project creation mechanics for Visual Studio, Rider, and Visual Studio Code. I used Rider for this book because it has exceptional tools for cleaning and formatting code that come in handy when you're writing a book.

Once Kitty had created the project, she put Visio on one of the other monitors so she could see the UML diagrams for the project. The first diagram she opened was actually the very last diagram they had worked on. You can see it in *Figure 7.3*. Tom had added a change to a central interface called `IManufacturable`. This is a very good place to start since everything else flows from this one interface.

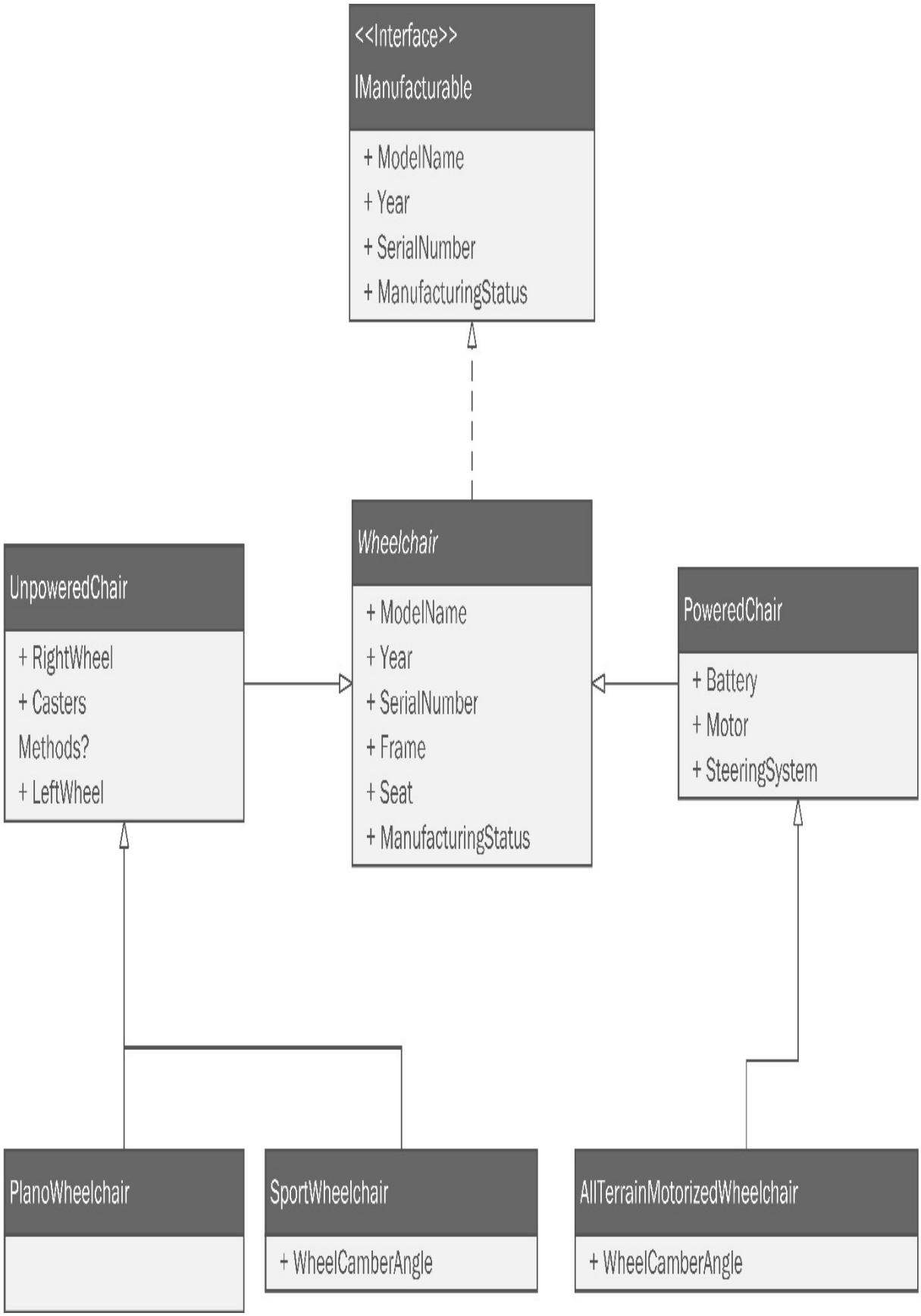

<<Interface>>
IManufacturable
+ ModelName
+ Year
+ SerialNumber
+ ManufacturingStatus
Wheelchair
+ ModelName
+ Year
+ SerialNumber
+ Frame
+ Seat
+ ManufacturingStatus
UnpoweredChair
+ RightWheel
+ Casters
Methods?
+ LeftWheel
PoweredChair
+ Battery
+ Motor
+ SteeringSystem
PlanoWheelchair
SportWheelchair
+ WheelCamberAngle
AllTerrainMotorizedWheelchair
+ WheelCamberAngle

Figure 7.3: The IManufacturable interface and a class structure that implements make the perfect place to start.

Kitty adds a new file called **`IManufacturable`** and types the code indicated by the diagram:

```
public interface IManufacturable
{
    public string ModelName { get; set; }
    public int Year { get; }
    public string SerialNumber { get; }
    public string BuildStatus { get; set; }
}
```

We've seen these properties before in the **`bicycle`** project. The **`Year`** and **`SerialNumber`** properties are going to be set automatically by the constructor in the implementing class, so only a **`get`** method is needed.

Next, Kitty adds the abstract **`Wheelchair`** class:

```
public abstract class Wheelchair : IManufacturabl
{
```

The class is marked as **`abstract`** as per the diagram. Don't forget that in UML, abstract classes display the class title *in italics*. This can be difficult to see with some of the fonts in UML modeling tools. The **`Wheelchair`** class implements the **`IManufacturable`** interface Kitty made a moment ago. Most IDEs can generate the missing members specified by the interface. Kitty generated the following code. She made **`BuildStatus`** nullable by adding the question mark next to the type definition:

```
public string ModelName { get; set; }
public int Year { get; }
public string SerialNumber { get; }
public string? BuildStatus { get; set; }
```

The constructor is almost identical to those used in the **`bicycle`** project. Since the class is abstract, it makes sense to expose the constructor as **`protected`**. Within the constructor, Kitty initializes everything she can:

```
protected Wheelchair()
{
    ModelName = string.Empty;
    SerialNumber = Guid.NewGuid().ToString();
    Year = DateTime.Now.Year;
}
```

```
}
```

The **`SerialNumber`** attribute is a GUID. This is a system-generated string that is guaranteed to be unique. This makes it perfect for a serial number. The **`Year`** attribute is initialized to the current year.

"Let's wait on the frame and seat properties for now. They're not primitives and we haven't created classes for them yet," Tom said.

Tom's design split wheelchairs into two types: unpowered chairs and powered chairs. Kitty adds the **`UnpoweredChair`** class next:

```
public abstract class UnpoweredChair : Wheelchai

{

}
```

She doesn't make it very far. Like most class diagrams, the types needed for the **`RightWheel`**, **`LeftWheel`**, and **`Casters`** properties aren't specified. "*We need the composite diagram for the types,*" Tom reminded her. That particular diagram can be seen in *Figure 7.4*:

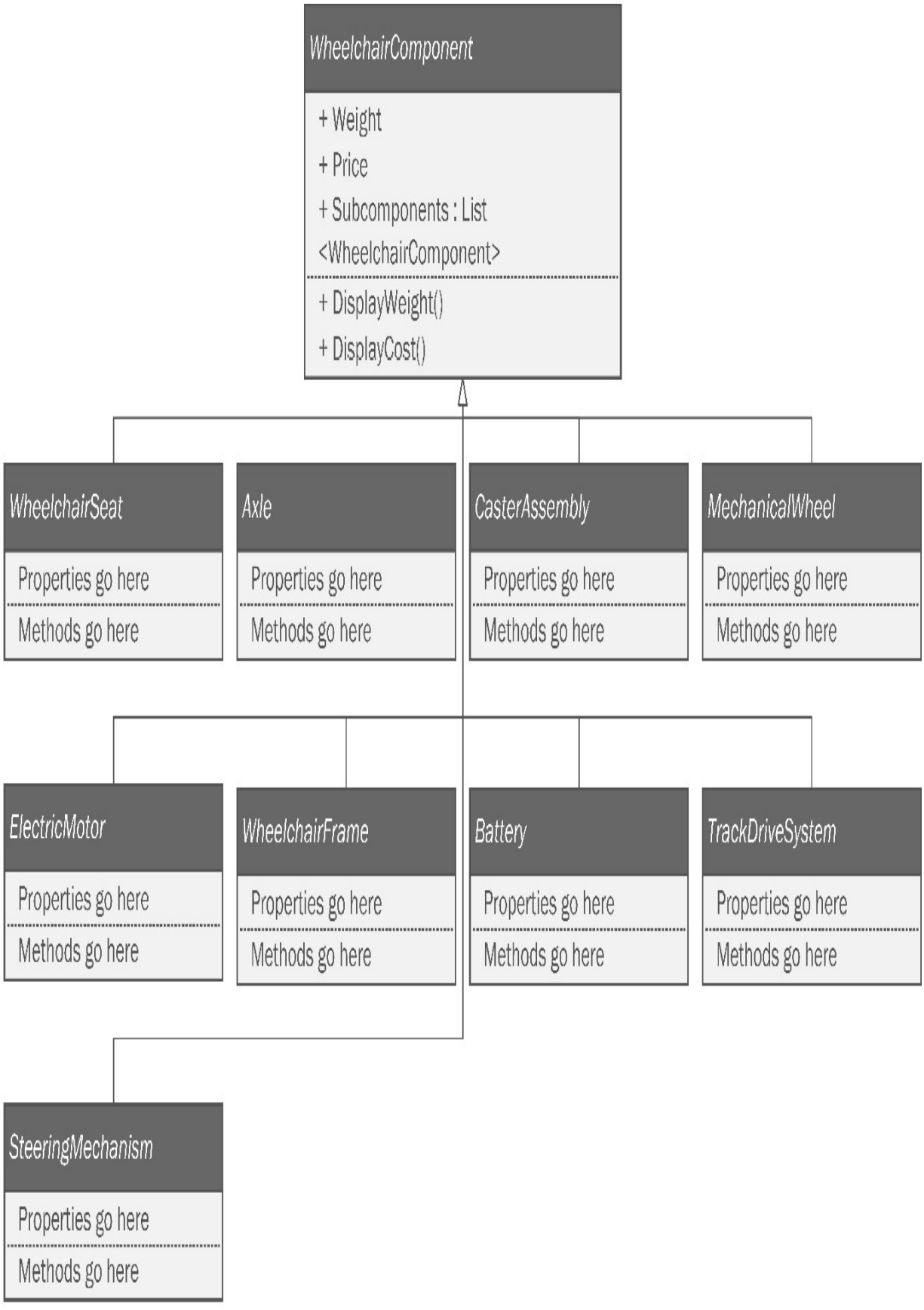

WheelchairComponent
+ Weight
+ Price
+ Subcomponents : List
<WheelchairComponent>
+ DisplayWeight()
+ DisplayCost()
WheelchairSeat
Properties go here
Methods go here
Axle
Properties go here
Methods go here
CasterAssembly
Properties go here
Methods go here
MechanicalWheel
Properties go here
Methods go here
ElectricMotor
Properties go here
Methods go here
WheelchairFrame
Properties go here
Methods go here
Battery
Properties go here
Methods go here
TrackDriveSystem
Properties go here
Methods go here
SteeringMechanism
Properties go here
Methods go here

Figure 7.4: The Composite structure.

Tom's design for the Composite pattern makes each part of the wheelchair out of an abstract class called **`WheelchairComponent`**. The wheels and casters in our current model need this abstract class defined before we can get the types correct. Kitty adds a new class called **`WheelchairComponent`**:

```
public abstract class WheelchairComponent
{
```

In order to use the Composite pattern, we need a few properties. The point of the Composite pattern is to use a tree-like structure to process lists. Kitty and Phoebe want to be able to iterate recursively over their wheelchair model instance to report on the weight and cost for each component. These numbers can be summed at any level to determine the weight and cost of a single part or a subset of the chair. For example, it might be useful to know the weight and cost of the assembled frame. When we are working with bicycles, weight and cost are typical trade-offs. Cheaper components are heavier. Competitive cyclists are willing to pay more to shave grams off their total weight. Doing this can reduce the time of a long ride by a few seconds, which might be enough to win a race.

Lightweight components for a wheelchair have the same concerns as bicycles; however, the motivation is driven by a different purpose. Kitty and Phoebe's dad has a degenerative muscle disease. He specifically needs something lightweight if he's going to be using an unpowered chair. Some wheelchair users have very strong upper bodies; others are not as strong. Lightweight materials are still balanced against cost, and the Composite pattern will help the engineering team analyze and improve the cost-to-weight ratio for the wheelchairs.

Since this class is abstract, its members are going to be **`protected`**, unless there's a good reason for them to not be **`protected`**. Kitty adds properties for **`Weight`** and **`Price`** as floats:

```
protected float Weight { get; set; }
protected float Price { get; set; }
```

Next, Kitty adds the crucial part of the class. The Composite pattern needs a **`Subcomponents`** collection. Remember, components in the composite are either leaves or containers. Containers are nodes in the tree that have other nodes within them. During the design phase, Kitty and Tom drew a diagram of the composite's tree structure on a whiteboard. You can see that again in *Figure 7.5*, with Tom's design written on the left side and Kitty's on the right.

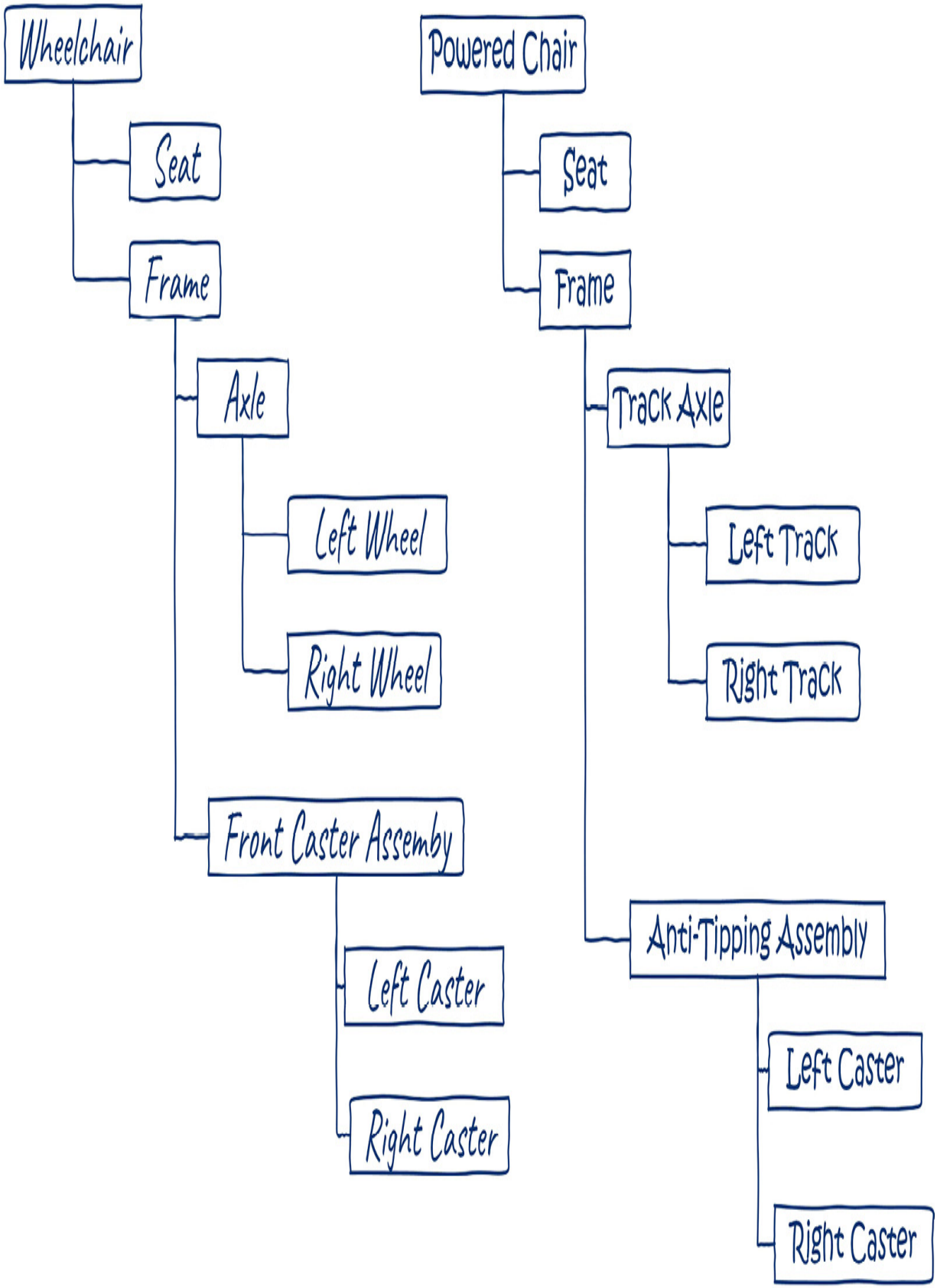

Wheelchair
Seat
Frame
Axle
Left Wheel
Right Wheel
Front Caster Assemby
Left Caster
Right Caster
Powered Chair
Seat
Frame
Track Axle
Left Track
Right Track
Anti-Tipping Assembly
Left Caster
Right Caster

Figure 7.5: Tree-like structure for powered and unpowered chairs.

If you look at the axle on the unpowered chair, you will see that it has components within it. The axle is a container. The right caster, on the other hand, doesn't contain any other components, so it is a leaf.

Regardless of whether they are a container or a leaf, all classes in the structure use the same base class, which contains a collection to hold other components. In effect, everything has the potential to be a container, even if it is merely a leaf. Kitty adds the collection:

```
    protected List<WheelchairComponent> Subcompor
```

Note: the collection type is the very class she's writing. This allows the composite to be iterable. We will rely on Liskov substitution heavily by requiring this base class for our types, but will supply concrete classes when we actually implement the Builder pattern later.

Next, Kitty adds a constructor and initializes all the properties:

```
    protected WheelchairComponent()
```

```
    {
        Subcomponents = new List<WheelchairCompor
        Weight = 0.0f;
        Price = 0.0f;
    }
```

The diagram in *Figure 7.4* indicates a couple of methods. Kitty isn't far enough along yet to worry about the specifics of these methods. For now, she just adds a line that throws **`NotImplementedException`** should anything try to access these methods. This is a very common placeholder:

```
    protected void DisplayWeight()
    {
        throw new NotImplementedException();
    }
    protected void DisplayCost()
    {
        throw new NotImplementedException();
    }

}
```

With this done, Kitty can return to defining the **`UnpoweredChair`** class. She switches to the **`UnpoweredChair.cs`** file in her IDE. This time, she

gets a little further. She adds the three properties required by the class diagram (*Figure 7.3*):

```
public abstract class UnpoweredChair : Wheelchai
{
    protected WheelchairComponent RightWheel { ge
    protected WheelchairComponent LeftWheel { get
    protected WheelchairComponent Casters { get;
}
```

Kitty's IDE indicates a problem with a yellow wavy line beneath each property. She hasn't initialized any of them. This isn't a huge problem because she knows she's going to be using the Builder pattern later to fill in these details based on which chair model is being built.

It was right about then that Kitty was startled by a loud voice behind her. "*PEEE-EW! That's some smelly code you got there!*" Phoebe exclaimed. Apparently, having downed a few slices of pizza and a few cans of caffeinated fizzy water, she was ready to be tagged in.

"*What?*" Kitty asked sheepishly. Tom chimed in, "*I think she's upset because you used the WheelchairComponent base class for your types.*"

"*I thought that's what we were doing*," Kitty replied.

"*Look again, Sis. The composite diagram has classes under the WheelchairComponent class that are more specific,*" Phoebe said. Kitty got up and Phoebe took her place.

Wheelchair components

Phoebe sat down and made a show of stretching her neck and cracking her knuckles. If this were a fight scene in an action movie, this would be her opponent's cue to be intimidated. Instead, the cursor blinked unafraid in the IDE. "*This is a good spot for me to fill in because I know a little bit more about the actual components,*" said Phoebe. She was right. Phoebe had spent many hours sourcing parts and figuring out how to make what she couldn't buy while building bikes. In *Figure 7.4*, we can see there is a set of abstract components that all inherit from **`WheelchairComponent`**. The **`WheelchairComponent`** base class gives us the **`Weight`** and **`Price`** fields used in the composite pattern to help us iteratively compute the weight and price of a set of components. The diagram lists nine such components:

- **`WheelchairSeat`**
- **`Axle`**
- **`CasterAssembly`**
- **`MechanicalWheel`**
- **`ElectricMotor`**
- **`WheelchairFrame`**

- **`Battery`**
- **`TrackDriveSystem`**
- **`SteeringMechanism`**

Phoebe will need to make a class for each item on the list. "*That's a lot of components*," Phoebe said. "*Can we focus on a minimum viable product here?*"

A **minimum viable product** (**MVP**) is a term from agile development. **Agile development** is a project management paradigm Kitty had learned in one of her product development courses. Agile methodologies are popular in software companies because they allow you to get a product to market quickly by focusing on the smallest product the company can sell. It then uses **iterative development** to build that product in a series of small bursts of effort. In this case, we have two wheelchair designs. The *Texas Tank* is seriously cool, but it will also be seriously expensive to make. The trio realizes they can make a huge impact by building the simple unpowered chair: the *Plano Wheelchair*. The faster this chair goes to market, the more people they can help. The team decides to focus on just making that chair. This cuts down her list to just these components:

- **`WheelchairSeat`**
- **`Axle`**
- **`CasterAssembly`**

- **`MechanicalWheel`**
- **`WheelchairFrame`**

"*Why don't you make a folder for those? It will make them easier to find later and our code will be a little less cluttered*," Tom suggested. A focused expression settled over Phoebe's countenance. She added a folder to her project called **`WheelchairComponents`**.

Next, Phoebe added a class within the folder called **`WheelchairSeat`**. She also added the remaining classes from the list, each one being empty for now. At this point, her project resembles *Figure 7.6*:

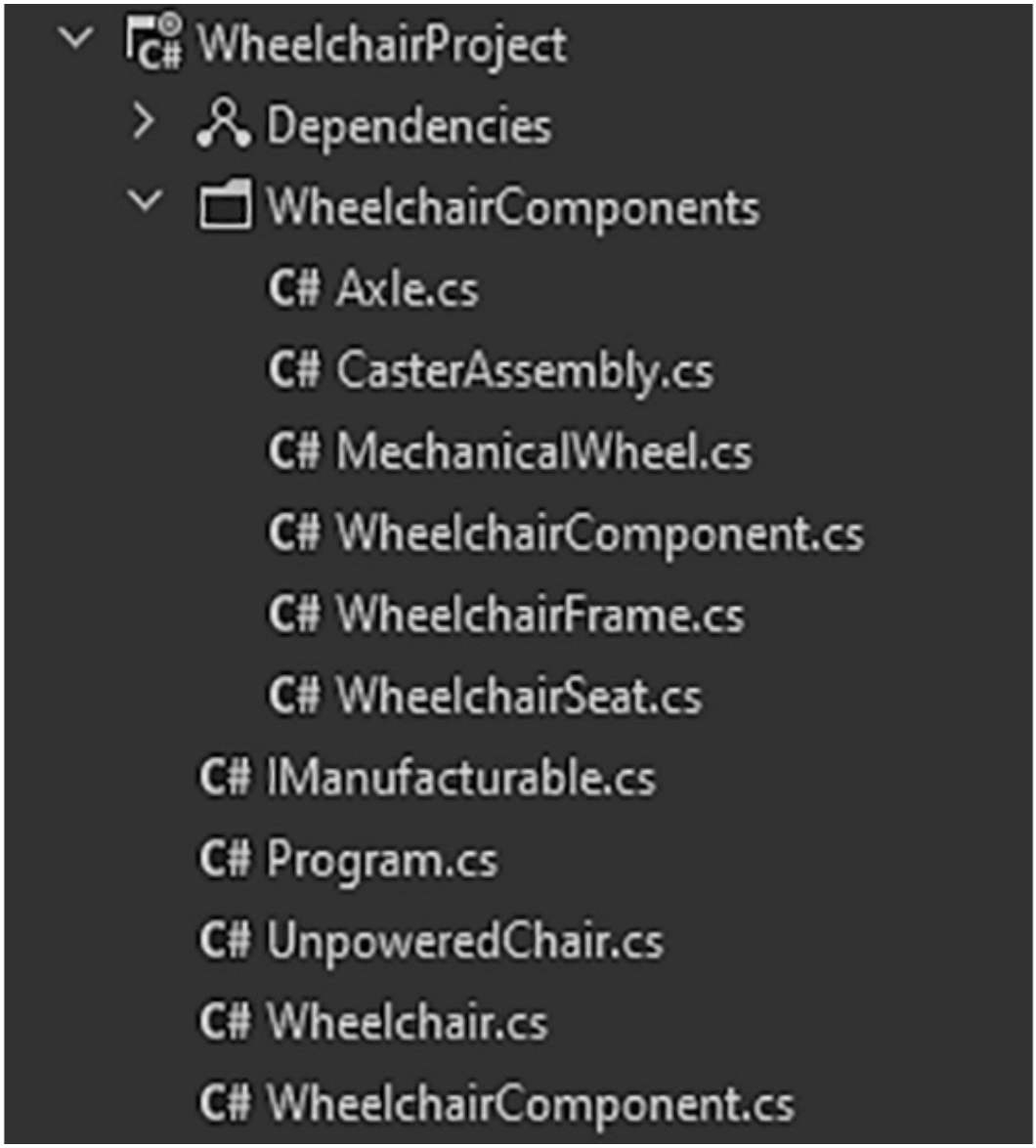


Figure 7.6: The folder structure for the abstract wheelchair components.

"*It probably makes sense to move the WheelchairComponent.cs file into that folder as well*," Tom advised. Phoebe dragged the file from the project's root folder into the subfolder with the other nine class files she had just created.

"*Hey, open that file real quick. Let's make sure the IDE changed the namespace for you,"* said Tom. *"Some IDEs do this and some don't,"* he continued. Phoebe double-clicked the file in Rider. The IDE, in fact, did not change the code. Namespaces in C# are like packages in Java. Their job is to help you keep your code organized and to prevent name collisions in complicated projects. Tom doubted this project would be complicated enough to have such a name collision. A name collision occurs when you want to name two classes with the same name. This is illegal in C# if the classes are in the same namespace. The project's namespace is usually created automatically when the project is created in your IDE.

Changing the namespace in the **`WheelchairComponent.cs`** file isn't crucial to the operation of the program. Since Tom and Phoebe have chosen to organize their code into folders, it is conventional to have the namespace match the folder structure. Phoebe modifies the **`WheelchairComponent`** class code to reflect the new folder location. This is done at the top of the file. The code before the operation reads as follows:

```
namespace WheelchairProject;
public abstract class WheelchairComponent
{
```

Phoebe corrects it to the following:

```
namespace WheelchairProject.WheelchairComponents;
public abstract class WheelchairComponent
{
```

After she did this, her IDE immediately colored her **`UnpoweredChair`** class with red underlines alerting her to a syntax problem. It used **`WheelchairComponent`**, and we just changed the namespace. Fixing this is error is easy. We just need to add a **`using`** statement to the top of the file. However, doing so right now is pointless. Phoebe intends for this class to use the class files she just created instead of the abstract **`WheelchairComponent`** class, which is too generic to represent an actual component.

Phoebe switches her focus back to the **`MechanicalWheel`** component she created a moment ago. Right now, the class merely contains the boilerplate added by the IDE:

```
namespace WheelchairProject.WheelchairComponents;
public class MechanicalWheel
{

}
```

Phoebe goes to work on it:

```
namespace WheelchairProject.WheelchairComponents;
public abstract class MechanicalWheel : Wheelchai
{
    protected float Radius { get; set; }
    protected int SpokeCount { get; set; }
    protected bool IsPneumatic { get; set; }
}
```

Phoebe adds three properties to describe an abstract wheel. "*Why are we making this class abstract?*" Kitty asked. Kitty was sitting off to the side, but the group had set up a monitor so that whoever was sitting out from typing could see all the action. Phoebe answered, "We'll later *define more concrete wheels, but having this abstract is a smart defense against future change. We don't want to tightly couple our UnpoweredChair class to a particular wheel. Instead, just about any type of mechanical wheel you can imagine can be implemented as a subclass to this one. Unless we come up with a radically new type of wheel, all wheels will have a radius. Most wheelchair wheels have spokes, but even if you had one that didn't, you could set the spoke count to zero. Finally, most wheelchairs have some sort of tire. Some are solid and some use air. The ones that use air are called pneumatic tires and are similar to typical bicycle tires.*"

"I get it!" Kitty exclaimed. "*You're going to use this class and the others like it from our list to define the structure of the UnpoweredChair class. Then you're going to define concrete classes for the actual wheels we'll be using. When the builder class constructs the wheelchair object, it can specify the concrete classes. The Liskov substitution allows us to substitute a parent class for a subclass, thus our design remains flexible*."

"*You've got it!"* Tom said. "*Now we need to do the same thing with the rest of the components*."

I need to break the fourth wall here for a minute: I am what you might call a detail junkie. I'm the kind of guy that likes to point out mistakes in TV shows and movies. I think it's a safe assumption, given you are reading this book, that you have likely seen the TV show *Star Trek, The Next Generation*. If that was before your time, you now have a binge-watching assignment after you complete this book. Here are a few things I noticed in that show. In Season 1, Episode 13, the android Data drinks spiked champagne, falling backward, and in the very next scene, he is face down. In Season 1, Episode 25, Riker orders Geordi to increase the speed of the starship Enterprise to Warp 6. Geordi replies, *"Aye sir, full impulse."* In Star Trek, "warp" refers to a logarithmic scale relative to the speed of light, while "impulse" refers to sublight speed. As I said, things like that bother me. I realize, having said that, I will probably be inundated with emails regarding my

own inconsistencies within this book. Here's my rule about that: you're only allowed to do this if you tell all your friends to buy the book so you can have a nice chuckle over my errors. If all your friends buy the book, we'll sell enough to do a second printing, and I can fix all the continuity mistakes. Of course, I'll probably introduce some new ones during the revision process, so the cycle will continue. I'm okay with that if you are.

Here, I'm trying to spare you the tedium of making this so realistic that there are hundreds of classes. Don't forget that our focus is patterns. This isn't an attempt to build a real wheelchair model that would pass muster with a mechanical engineer. I'll be doing more of this as the chapter progresses. Direct your focus to the structure of the objects, not their contents. The parts important to the patterns are always going to be there; the rest just makes for a good story. Now that we've gotten that out of the way, let's fast-forward a bit. All of the classes in the **`WheelchairComponents`** folder are going to be abstract classes that subclass **`WheelchairComponent`**. The other properties don't affect the Composite pattern or any other part of our program.

We now return you to your story already in progress.

Phoebe continues filling out the remaining classes in the components list. We'll go in alphabetical order focusing only on the components needed in an unpowered chair. The first class is the Axle class. Re-

member that in the Component pattern, each element is either a container or a leaf. Containers contain other containers and leaves. A leaf cannot contain anything. According to *Figure 7.5*, the **`Axle`** is a container which holds the left and right wheels.

```
using WheelchairProject.WheelchairComponents.Whee
namespace WheelchairProject.WheelchairComponents.
public abstract class Axle : WheelchairComponent
{
```

I'll add the two wheels as private fields.

```
    private MechanicalWheel _leftWheel;
    private MechanicalWheel _rightWheel;
```

These next two properties define the axle, which is just a cylinder.

```
    protected float Radius { get; set; }
    protected float Length { get; set; }
```

Next, Phoebe adds accessor methods for the **`_leftWheel`** and **`_rightWheel`** fields. First comes the left wheel. Note the **`FixComposite()`** method. This implements Tom's original idea of having the Composite pattern baked directly into the object structure, in contrast

with Kitty and Phoebe's implementation in the bicycle project which used a separate object graph. We'll create the `FixComposite()` method in just a moment.

```
public MechanicalWheel LeftWheel
{
  get => _leftWheel;
  set
  {
    _leftWheel = value;
    FixComposite();
  }
}
```

Then comes the right wheel.

```
public MechanicalWheel RightWheel
{
  get => _rightWheel;
  set
  {
    _rightWheel = value;
    FixComposite();
  }
}
```

Next, with Tom's guidance, Phoebe creates a method called **`FixComposite()`**. *"You need to put this on the abstract components."*, Tom said. Phoebe added the following code:

```
    private void FixComposite()
    {
      Subcomponents.Clear();
      Subcomponents.Add(_leftWheel);
      Subcomponents.Add(_rightWheel);
    }
  }
```

"I see what you're doing!", Kitty said from her chair behind Phoebe. *"Each component manages the subcomponents inside it. Instead of making a separate object graph, you're having the accessor methods rebuild the composite every time the left and right wheels change."*

"Why aren't we putting it in the **`WheelchairComponent`** *base class?"*, asked Phoebe. *"There are two reasons."*, Tom said. *"First, this method is specific to the Axle. It contains these two specific components"* Phoebe interrupted, *"But we could make it abstract, and override it in the subclasses."*

"Let me finish, Phoebe", said Tom. He continued, *"The second reason is the interface segregation principle from SOLID. Only containers*

need this method. The leaves don't have any sub-components, and so have no need to fix the list of subcomponents on the Wheelchair-Component base class. The interface segregation principle holds that no class should be forced to implement an interface it doesn't use, nor should it depend on a method it doesn't need. If we make this an abstract method in the base class, the leaf classes will have to implement it. Any implementation we add with be contrived, and it will introduce a code smell to those classes."

"Who is writing smelly code, now Phoebe?", Kitty chided as she enjoyed sweet revenge on her sister.

Phoebe shrugged off her sisters jab and continued working. Based on *Figure 7.5*, she knows these classes need a **`FixComponent()`** method because they are containers:

- **`Axle`** (already written)
- **`CasterAssembly`**
- **`Wheelchair`**
- **`WheelchairFrame`**

`Wheelchair` is the base class for all wheelchairs. It needs to inherit from **`WheelchairComponent`** because we need a top-level container. Phoebe makes extensive modifications to the Wheelchair class. When she's finished, it looks like this:

```
using WheelchairProject.WheelchairComponents.Fra
using WheelchairProject.WheelchairComponents.Seat
```

Phoebe has added the frame and the seat namespaces so we can specify the two components that go into the wheelchair. She continues by adding the **`WheelchairComponents`** reference right below the namespace.

```
namespace WheelchairProject;
using WheelchairComponents;
```

Next, she adds the **`WheelchairComponent`** as the base class:

```
public abstract class Wheelchair : WheelchairComp
{
```

Next, she adds the private fields for the seat and frame:

```
    private WheelchairSeat _seat;
    private WheelchairFrame _frame;
  public string ModelName { get; set; }
```

```
    public int Year { get; }
    public string SerialNumber { get; }
```

Then she adds the accessor methods just like before with the **`Axle`** class:

```
    public WheelchairSeat Seat
    {
      get => _seat;
      set
      {
        _seat = value;
        FixComposite();
      }
    }
    public WheelchairFrame Frame
    {
      get => _frame;
      set
      {
        _frame = value;
        FixComposite();
      }
    }
```

Next, Phoebe adds the magical **`FixComposite()`** method which clears out the subcomponent list and adds the _seat and _frame fields. This

is called from the accessor methods, so anytime these get changed, the composite is updated. Truthfully, since we'll be using the Builder pattern to create these objects, there likely won't be much call for changing the object once it gets created, but its nice to know you can.

```
    private void FixComposite()
    {
      Subcomponents.Clear();
      Subcomponents.Add(_frame);
      Subcomponents.Add(_seat);
    }
```

The rest of the class remains unchanged. Rather than cover every class Phoebe created, I encourage you to review her finished code in the **`chapter-7`** project in the book's sample code. You'll find most of this in the **`WheelchairComponents`** folder.

Finishing the wheelchair base classes

"Nice work, Phoebe. All that's left is to clean up Kitty's mess," Tom chided. Kitty stuck out her tongue. Phoebe giggled and opened the **`UnpoweredChair`** class. She now had all the types she needed to model the abstract classes. She updated the class code like so:

```
using WheelchairComponents;
```

Here, the **`using`** statement is needed because we moved the **`component`** classes into the **`WheelchairComponents`** namespace. The next few lines remain unchanged:

```
public abstract class UnpoweredChair : Wheelchai
{
```

Phoebe's next change is to set the correct types for the **`RightWheel`**, **`LeftWheel`**, and **`CasterAssembly`** properties:

```
    protected MechanicalWheel RightWheel { get; s
    protected MechanicalWheel LeftWheel { get; se
    protected CasterAssembly Casters { get; set;
```

Finishing up the composite

"We should take a break soon," Phoebe said. "*We've got a lot done. We've got most, if not all, of our abstract base classes written. Those classes are the most important parts of our patterns. We've also almost finished the composite pattern. We just need to add the meth-*

ods that recursively compute the weight and cost of each component."

"*To do that,"* Tom said, "w*e just need to make two small adjustments. Open the WheelchairComponent class."*

Phoebe complied. She remembered that Kitty had left the **`DisplayCost`** and **`DisplayWeight`** methods for later. They presently read as follows:

```
protected void DisplayWeight()
    {
        throw new NotImplementedException();
    }
    protected void DisplayCost()
    {
        throw new NotImplementedException();
    }
```

Phoebe adds the implementation:

```
protected void DisplayWeight()
{
```

As with the **`BicycleComponent`** implementation, first, we check to see whether there are any subcomponents. If not, we simply return:

```
        if (!Subcomponents.Any()) return;
```

If there are, we print the name and weight:

```
        foreach (var component in Subcomponents)
        {
            Console.WriteLine(component.GetType().Nar
```

Then, we call the **`DisplayWeight`** method recursively:

```
            component.DisplayWeight();
        }
    }
```

We do the same thing with **`Price`**:

```
    protected void DisplayCost()
    {
        if (!Subcomponents.Any()) return;
        foreach (var component in Subcomponents)
        {
```

```
            Console.WriteLine(component.GetType().Nar
            component.DisplayCost();
        }
    }
```

Phoebe slumped in her chair. "*Phew! That was a lot of work,*" said Phoebe. "*I know. We have the structure for the Composite pattern, but we still need a builder to populate it*," said Tom. "*Yeah,*" said Kitty. "*This structure is complicated. We can't just make a Wheelchair object and add the chair, the frame, and some wheels like we can with simple composition. We need a way to assemble the hierarchy we drew on the board earlier (Figure 7.5).* The three took a break and rested up. They knew the next pattern they needed to make was the Builder pattern.

Implementing the Builder pattern

Kitty had called Karina, their mother, and asked her to stop by, in order to give her a break from the non-stop hospital vigil. Phoebe shared some pizza and the four played a little Mario Kart on the Nintendo Switch to get their minds off their problems. After a few races, Tom, Kitty, and Phoebe were ready to get back to work. Karina returned to the hospital

"*This is going to be the hard part*," Tom said. "*The Builder pattern implementation is going to be doing a lot of work. It needs to assemble the wheelchair object, assemble the composite, and ultimately handle the wheelchair's paint job with the Bridge pattern.*"

"We should review our diagram," said Kitty. It was her turn to type. She brought up the diagram they had drawn of the builder pattern. You can review it in *Figure 7.7*:

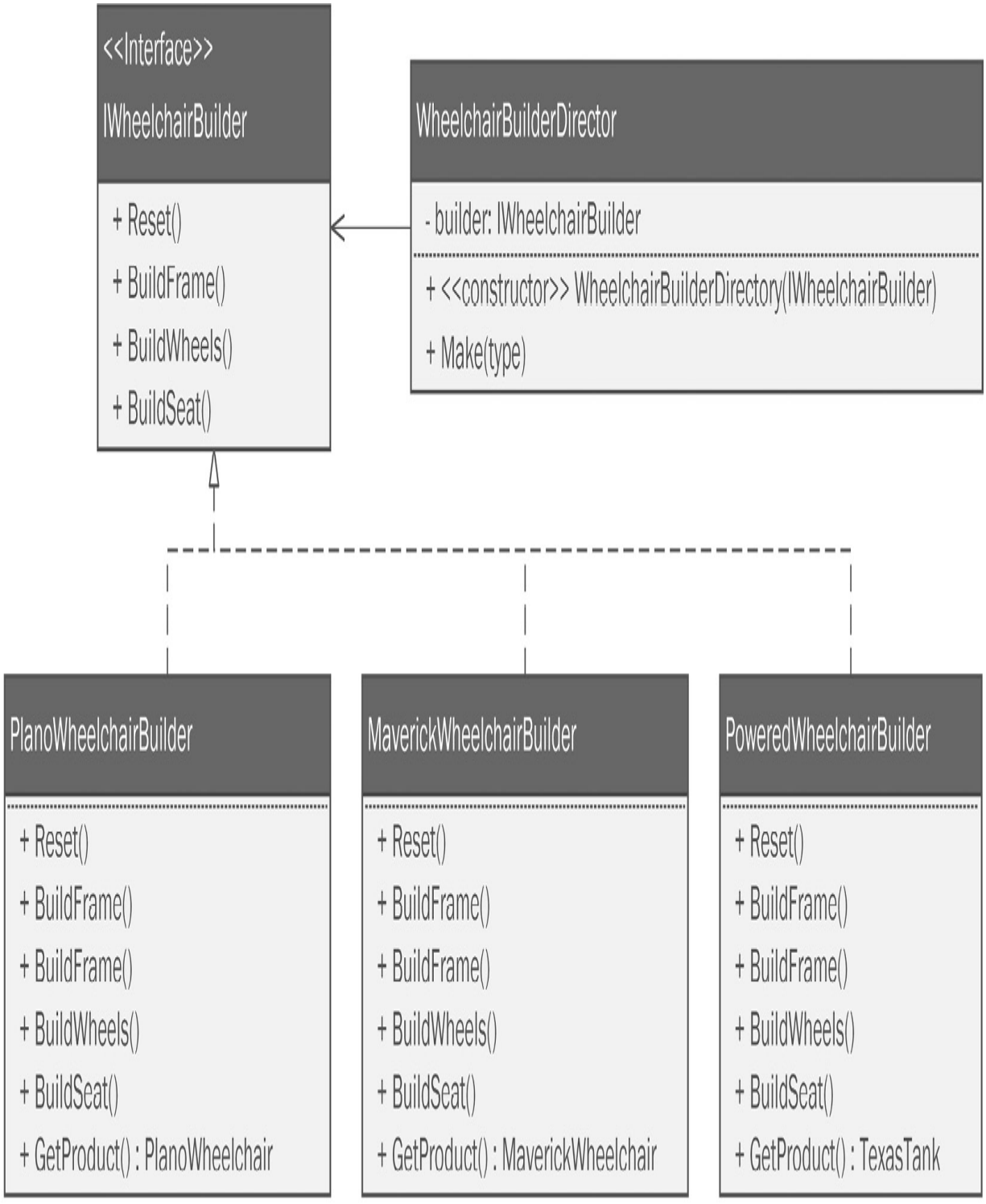


Figure 7.7: The Builder pattern design with everything on it, including the Composite and Bridge pattern elements.

"I think I'll make a folder to hold all the builder classes and interfaces. It seems like there are going to be a few of them," Kitty said. "*Good idea,"* said Tom. Kitty created a folder called **`Builders`**. Next, Kitty added the **`IWheelchairBuilder`** interface to the **`Builders`** folder:

```
namespace WheelchairProject.Builders;
```

Kitty implemented the code specified in the UML design:

```
public interface IWheelchairBuilder
{
    public void Reset();
    public void BuildFrame();
    public void BuildWheels();
    public void BuildSeat();
    public Wheelchair GetProduct();
}
```

Aside from the interface, the Builder pattern needs a **`director`** class. That is, it needs an abstract builder that implements the interface, and one or more concrete **`builder`** subclasses to build the concrete products. You can see each of these in *Figure 7.4*.

Kitty adds the director class next. She decides to call it **`WheelchairBuilderDirector`**:

```
namespace WheelchairProject.Builders;
public class WheelchairBuilderDirector
{
```

The director holds a private instance of any class implementing **`IWheelchairBuilder`**:

```
    private IWheelchairBuilder _builder;
```

Kitty adds a constructor that specifies the builder she wants to use on instantiation:

```
    public WheelchairBuilderDirector(IWheelchairB
    {
        _builder = builder;
    }
```

The **`Build`** method specified in the diagram is designed to call the various methods on the **`builder`** class specified in **`IWheelchairBuilder`**. Remember, the director's job is to call these in order. The complex logic behind building the object is controlled here in the director class:

```
        public Wheelchair Build()
        {
            _builder.BuildSeat();
            _builder.BuildFrame();
            _builder.BuildAxleAssembly();
            _builder.BuildCasterAssembly();
```

Finally, the director returns the built object using the **`GetProduct`** method:

```
            return _builder.GetProduct();
        }
    }
```

“So far, so good!” said Tom. “*Let’s make a concrete builder for the Plano Wheelchair*.”

Kitty added a class called **`PlanoWheelchairBuilder`**. It looked like this:

```
    namespace WheelchairProject.Builders;
    public class PlanoWheelchairBuilder : IWheelchai
    {
```

The pattern requires us to have a **`private`** field to hold the **`Wheelchair`** object as it is built:

```
private PlanoWheelchair _wheelchair;
```

The pattern also requires a **`Reset`** method, which essentially re-initializes the **`_wheelchair`** field. In order to keep the class **DRY** (**Don't Repeat Yourself**), you can create the **`Reset`** method, then call it from the constructor, which you also need. Kitty prefers to make sure the constructor is always the first method in the class, so it comes first:

```
Public PlanoWheelchairBuilder()
{
    Reset();
}
```

Then, she adds the **`Reset`** method:

```
public void Reset()
{
    _wheelchair = new PlanoWheelchair();

}
```

Since this class implements the **`IWheelchairBuilder`** interface, we need the remainder of the required methods. Most IDEs will generate this for you. Kitty is using Rider, so she clicks her cursor into the class name line near the top of the class. It has some angry-looking red squiggly lines beneath it because she hasn't implemented the methods required by the interface. She presses *Ctrl* + *.* (control-period) and she sees the option to generate the missing members with placeholder code. You can see what this looks like in *Figure 7.8*:

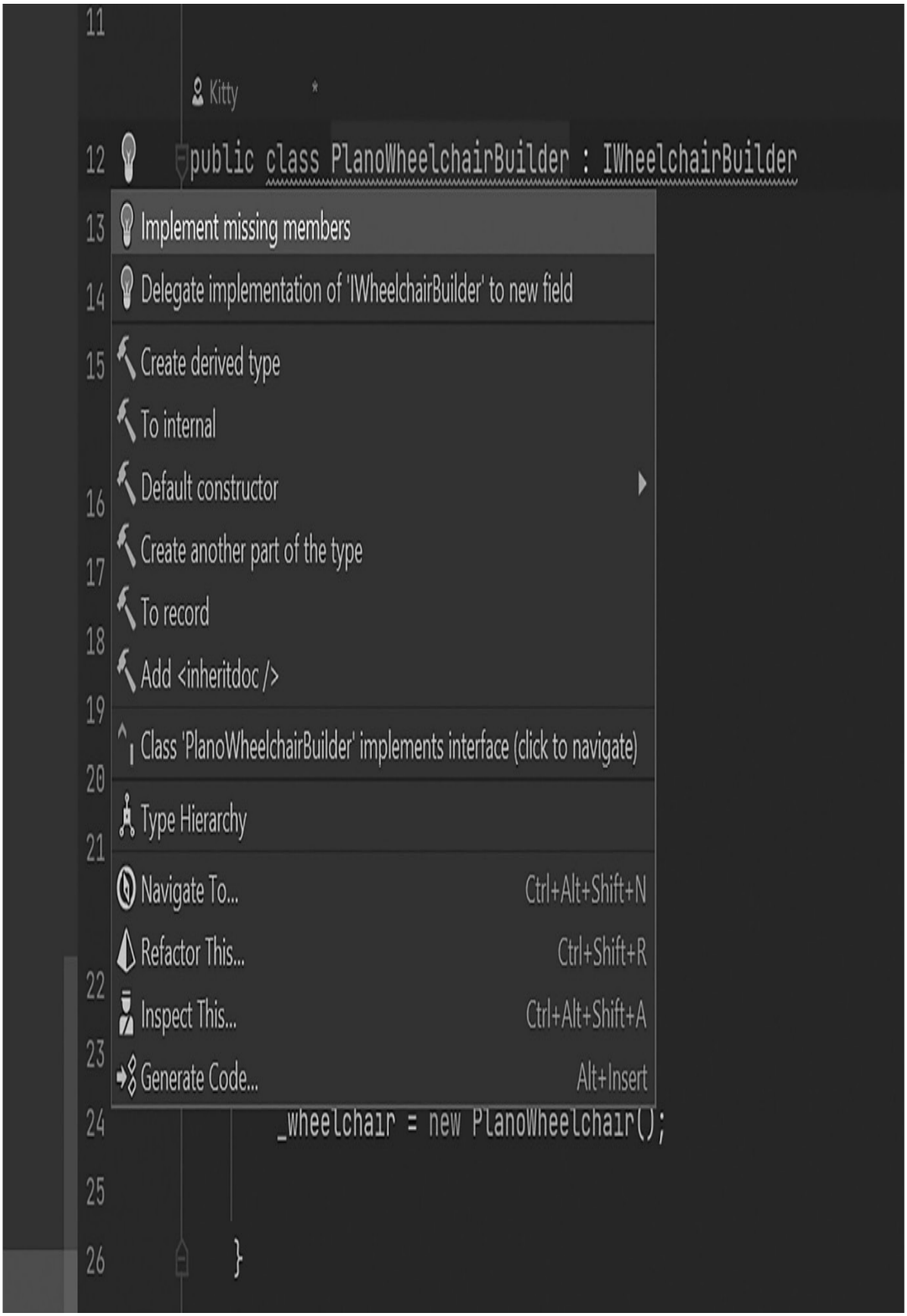

Kitty *
public class PlanoWheelchairBuilder : IWheelchairBuilder
Implement missing members
Delegate implementation of 'IWheelchairBuilder' to new field
Create derived type
To internal
Default constructor
Create another part of the type
To record
Add <inheritdoc />
Class 'PlanoWheelchairBuilder' implements interface (click to navigate)
Type Hierarchy
Navigate To... Ctrl+Alt+Shift+N
Refactor This... Ctrl+Shift+R
Inspect This... Ctrl+Alt+Shift+A
Generate Code... Alt+Insert
_wheelchair = new PlanoWheelchair();
}

Figure 7.8: Rider, like most IDEs, will autogenerate placeholder code for any missing members required by the interface.

The code generated looks like this:

```
    public void BuildFrame()
    {
       throw new NotImplementedException();
    }
    public void BuildWheels()
    {
        throw new NotImplementedException();
    }
public void BuildAxleAssembly()
    {
        throw new NotImplementedException();
    }
    public void BuildCasterAssembly()
    {
        throw new NotImplementedException();
    }
    public void BuildSeat()
    {

        throw new NotImplementedException();
    }
```

```
        public void BuildComposite()
        {
            throw new NotImplementedException();
        }
        public void BuildFramePainter()
        {
            throw new NotImplementedException();
        }
        public Wheelchair GetProduct()
        {
            return _wheelchair;
        }
    }
```

This code is long and uneventful, but it saves a lot of typing! Kitty just needs to fill in the implementation for each method. She can't really do that yet. She needs concrete implementations of the various abstract wheelchair component classes, and those have yet to be created.

Another refactor

"That's a whole lot of placeholder code," Tom said. *"Maybe it's time to add some concrete classes for the wheelchair components so we build a real wheelchair?"*

“OK,” Kitty agreed. She started to think about structuring the concrete classes. There is going to be at least one concrete component for each of the abstract component classes. Assuming the wheelchair project is wildly successful, the list of concrete implementations is likely to grow, making the **`WheelchairComponents`** folder very crowded.

“Tom, why don’t we split up the WheelchairComponents folder a little more? As we add concrete classes, I think it will be easier to find everything if we do it that way,” Kitty said. *“Good idea,*” Tom replied.

Kitty added a series of folders beneath the **`WheelchairComponents`** folder:

- **`Axles`**
- **`Casters`**
- **`Frames`**
- **`Seats`**
- **`Wheels`**

Next, she moved the abstract classes for each component into their respective folder. In each case, she corrected the namespace for the class she moved.

The **`Axle`** class goes in the **`Axles`** folder. The namespace code should be adjusted to this:

```
namespace WheelchairProject.WheelchairComponents.
```

The **`Caster Assembly`** class goes in the **`Casters`** folder. The namespace code should be adjusted to this:

```
namespace WheelchairProject.WheelchairComponents.
```

The **`WheelchairFrame`** class goes in the **`Frames`** folder. The namespace code should be adjusted to this:

```
namespace WheelchairProject.WheelchairComponents.
```

The **`WheelchairSeat`** class goes in the **`Seats`** folder. The namespace code should be adjusted to this:

```
namespace WheelchairProject.WheelchairComponents.
```

Finally, the **`MechanicalWheel`** class is moved to the **`Wheels`** folder and its namespace is adjusted like so:

```
namespace WheelchairProject.WheelchairComponents.
```

When this refactor is over, the code's folder structure looks like *Figure 7.6*.

Adding concrete component classes

Kitty's next job is going to be adding component classes for each component type. We've established a base class and structure for each component. Next, we simply need to add the concrete classes that extend the base classes.

Axles

She starts in the **`Axles`** folder by adding a class called **`StandardAxle`**. The parts for the *Plano Wheelchair* are mechanically simple and common. Phoebe has provided a **bill of materials** (**BOM**) listing the components, along with the data required by the Composite pattern implementation. In the world of manufacturing, this is a list of parts normally exported to an Excel spreadsheet. Kitty and Tom can simply reference the spreadsheet to get the values needed in the concrete classes.

The **`StandardAxle`** class looks like this:

```
using WheelchairProject.WheelchairComponents.Whee
namespace WheelchairProject.WheelchairComponents.
public class StandardAxle : Axle
{
    public StandardAxle(MechanicalWheel leftWheel
    MechanicalWheel rightWheel)
    {
        Price = 4.33f;
        Weight = 0.335f;
        Radius = 0.24f;
        Length = 28.5f;
        LeftWheel = leftWheel;
        RightWheel = rightWheel;
    }
}
```

Casters

Next, Kitty adds a concrete class for the caster assembly to be used on the *Plano Wheelchair*. Unsurprisingly, she calls it **`PlanoCasterAssembly`**. Its contents are as follows:

```
using WheelchairProject.WheelchairComponents.Whee
namespace WheelchairProject.WheelchairComponents.
public class PlanoCasterAssembly : CasterAssembly
```

```
{
    public PlanoCasterAssembly(MechanicalWheel wh
    {
        LoadCapacity = 300.0f;

        MountingType = "STEM";
        Weight = 0.443f;
        Price = 4.32f;
        Wheel = wheel;
    }
}
```

Frames

It's time to add a concrete class for the *Plano Wheelchair's* frame. It's called **`PlanoWheelchairFrame`**:

```
namespace WheelchairProject.WheelchairComponents.
public class PlanoWheelchairFrame : WheelchairFra
{
    public PlanoWheelchairFrame()
    {
        Price = 75.92f;
        Weight = 16.34f;
    }
}
```

Seats

A wheelchair wouldn't be very useful without a place to sit. Kitty adds a class she calls **`PlanoSeat`**:

```
namespace WheelchairProject.WheelchairComponents.
public class PlanoSeat : WheelchairSeat
{
    public PlanoSeat()
    {
        Price = 27.48f;
        Weight = 3.22f;
        Width = 22;
        BackHeight = 30;
        SeatThickness = 2.4f;
    }
}
```

Wheels

Equally important to the construction of a wheelchair, beyond the seat, are the wheels. There are two of these. The large wheels on the side of the wheelchair are specified by a class called **`StandardWheel`**:

```
namespace WheelchairProject.WheelchairComponents.
```

```
public class StandardWheel : MechanicalWheel
{

    public StandardWheel()
    {
        Price = 11.34f;
        Weight = 1.3f;
        Radius = 16f;
        IsPneumatic = true;
        SpokeCount = 48;
    }
}
```

The second type of wheel we need is the smaller set that attaches to the swiveling casters on the front of the chair. Kitty called these **`CasterWheel`**:

```
namespace WheelchairProject.WheelchairComponents.
public class CasterWheel : MechanicalWheel
{
    public CasterWheel()
    {
        Price = 5.21f;
        Weight = 0.753f;
        Radius = 6f;
        IsPneumatic = true;
        SpokeCount = 24;
```

```
        }
    }
```

Now, Kitty's folder structure resembles *Figure 7.9*:

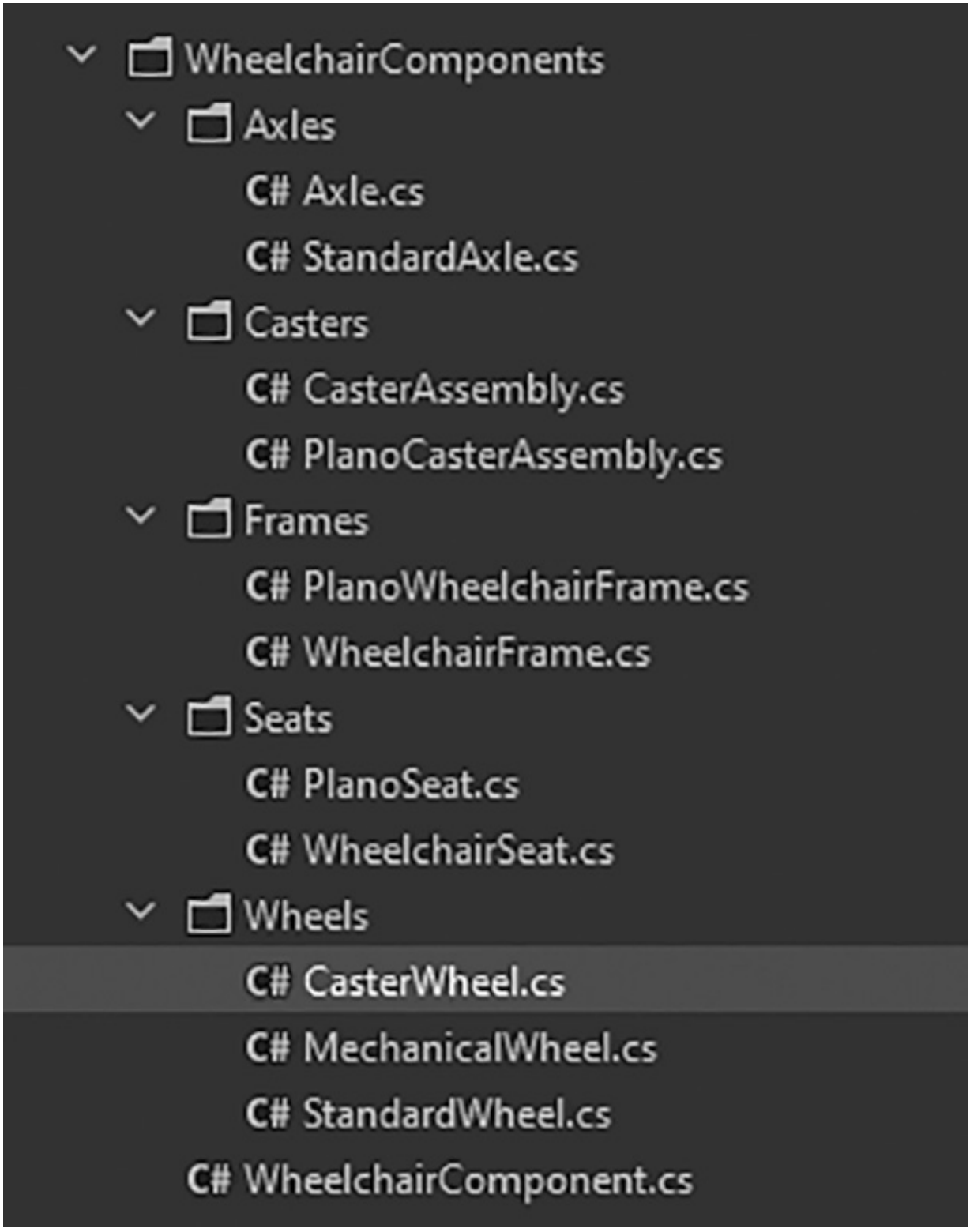


Figure 7 .9: The WheelchairComponents folder after Kitty added all the concrete classes.

Now that all the concrete classes for the *Plano Wheelchair* are in place, we can set up the Builder pattern code to build a full **`Wheelchair`** object.

Wrapping up the Builder pattern

At this point, Kitty needs to finish the **`PlanoWheelchairBuilder`** class. If you recall, this was the class where she generated all that placeholder code earlier. She needs to replace the placeholder code with code using her new concrete classes.

Kitty starts with the frame because the frame is the foundation for all the other parts of the wheelchair. She changes the code in the **`BuildFrame`** method generated by the IDE:

```
public void BuildFrame()
{
    wheelchair.Frame = new PlanoWheelchairFrame()
}
```

There's not much to the **`BuildFrame`** method. All it does is instantiate **`PlanoWheelchairFrame`** and set the **`_wheelchair`** frame property.

Next, Kitty replaced the **`BuildAxleAssembly`** method. An **`Axle`** object, such as the **`StandardAxle`** class, per the Composite pattern, is a con-

tainer. The team specified this in *Figure 7.5*. The **`Axle`** contains the left and right wheels, which are the **`StandardWheel`** type. As such, the code looks like this:

```
public void BuildAxleAssembly()
{
    var leftWheel = new StandardWheel();
    var rightWheel = new StandardWheel();
    var axle = new StandardAxle(leftWheel, rightW
    _wheelchair.Frame.Axle = axle;
}
```

Here, we've instantiated two **`StandardWheel`** objects and set them into **`Axle`** using the **`StandardAxle`** constructor. Lastly, we set the **`_wheelchair`**'s **`Axle`**, which is a property of the **`Frame`** property. This mirrors real life because an axle would be attached to the wheelchair's frame, and the wheels, in turn, would be affixed to the axle.

In her mind's eye, all Kitty sees is a wheelchair with two wheels awkwardly rocking back and forth. With the main wheels attached, Kitty decides to do the casters next. She alters the **`BuildCasterAssembly`** method like this:

```
public void BuildCasterAssembly()
{
```

```
{
    var planoCasterWheel = new CasterWheel();
    var casterAssembly = new     PlanoCasterAsser
    _wheelchair.Frame.LeftCaster = casterAssembly
    _wheelchair.Frame.RightCaster = casterAssembl
}
```

PlanoCasterAssembly consists of **CasterWheel**, which is passed into the **PlanoCasterAssembly** constructor. The assembly is then mounted to the frame on the left and right sides. As I review Kitty's code, I can't tell whether she's cutting a corner here or not. She used the same instance of **PlanoCasterAssembly** on both sides. I'm sure it's probably right. If it isn't, I'm sure Phoebe would have let her know.

We have four wheels and a frame. We're missing a seat. Kitty updates the **BuildSeat** method:

```
public void BuildSeat()
{
    _wheelchair.Seat = new PlanoSeat();
}
```

Like the frame, this one is straightforward. Just instantiate the **PlanoSeat** object and attach it to the frame. Kitty needs one more easy change. **GetProduct** should not return an instantiation of **PlanoWheelchair**. Instead, it should return the one we've built:

```
public Wheelchair GetProduct()
{
    return _wheelchair;
}
```

"*Wow, that's really neat!"* Phoebe said. Phoebe had been wandering in and out of the lab, but all of a sudden, showed a renewed interest. *"I see what you mean by hiding the Composite pattern complexity inside the builder. Anyone working with the object can use normal composition, but we also get the benefit of the recursive price and weight functions,"* Phoebe said.

"*Are we still going to make it a singleton?"* Tom asked.

"I think we should try," Kitty said. "*It would be interesting to write some tests to see whether we gain any benefit from making this a singleton. But for now, let's go ahead and turn the builder into a singleton*."

Adding the Singleton pattern

"*Tag! I'm in!"* Phoebe exclaimed, as she nudged Kitty out of the typing chair and sat down at the keyboard. Phoebe made a show of cracking her knuckles and popping her neck. They were in the home stretch and Phoebe knew it.

"*The Singleton is pretty easy*," Tom said. *"I remember* it," said Phoebe. She continued, *"It seems like all I need to do is change the director class in the Builder pattern.*" Phoebe located the director class in the **`Builders`** folder and opened it in her IDE:

```
public class WheelchairBuilderDirector
{
    private IWheelchairBuilder _builder;
```

Phoebe adds this line to create a field to hold the current instance. She makes it nullable because if the field is **`null`**, we need to create a new instance of this class and place it in the **`_instance`** field. The field is marked **`static`** to ensure it is unique in memory:

```
    private static WheelchairBuilderDirector? _ir
```

Next, Phoebe changes the constructor's accessor to **`private`**. It should not be possible to instantiate the director directly. To use the class, you have to use the hallmark **`GetInstance`** method, which she'll write next:

```
    private WheelchairBuilderDirector(IWheelchair
    {
```

```
            _builder = builder;
        }
```

The last step to converting the builder's director class is to add a public static method called **`GetInstance`**. We still need the **`IWheelchairBuilder`** parameter to be passed in just as we did on the original constructor. This method checks to see whether the **`_instance`** field is null. If it is, then this method will invoke the private constructor and set the **`_instance`** field to the result. If the **`_instance`** field is not null, the **`GetInstance`** method will simply return the instance it already has:

```
    public static WheelchairBuilderDirector? GetI

    {
        if (_instance == null)
        {
            _instance = new WheelchairBuilderDire
        }
        return _instance;
    }
```

The rest of the class remains unchanged.

"*That wasn't so tough,"* Phoebe said. "*I can imagine this object being expensive in terms of memory once the factories are producing*

wheelchairs as well as bicycles," Tom said. "*I'm on the fence*," Phoebe replied. "*I'm not totally sure we need this, but I don't think it will hurt anything. There's just one thing left, and it's my favorite!*" said Kitty, watching from behind Phoebe's chair. "*You're proud of your paint system, aren't you Sis?"* asked Phoebe. Kitty just grinned and sat down behind her sister.

Painting the chairs with the Bridge pattern

The last pattern we need to complete our wheelchair project is the Bridge pattern. Remember, the Bridge pattern is used when you have two related systems of complex classes. The bridge allows you to join the classes using composition. It gives you the benefit of being able to vary the complexity and maintain these complicated classes independently. Kitty and Phoebe used this system to add the ability to create custom paint jobs for their bicycles. They did this late in the game though, and the changes they had to make to accommodate those changes were problematic. They had to violate the open-closed principle.

This time, with experience behind them, they can integrate the Bridge pattern for painting wheelchairs early in the implementation. The ability to specify colors for wheelchairs will be a big differentiator for Bum-

ble Bikes since most manufacturers only sell black and gray chairs. This is fine for the loaner chair at your local hospital, but for people who use wheelchairs every day to get around in life, it's nice to have a little fashion in the mix.

Kitty created the paint system for the bicycles to work in a similar fashion as an inkjet printer works. She devised a system that could mix any color using cyan, magenta, yellow, and black paint. This is called the **CMYK** color model, and it's a standard in the printing industry.

Tom, who works with disabled children at a local pediatric hospital, had taken an informal survey on what the most popular colors might be for a wheelchair. Then he asked the kids to pick just one color for the initial product release. After some animated discussion, the kids decided on a shade of green. There was a woman who worked with the kids, named Judy. Everyone loved her, and she had sparkling green eyes. They decided to honor her by naming the color *Green Eyed Judy*.

Phoebe got to work on implementing the Bridge pattern. In real life, it might be possible to reuse the bridge interfaces and classes from the bicycle package. The only downside is that you'd create a dependency between two products that don't necessarily have anything to do with each other besides being two products made by the same com-

pany. For the purposes of this book, we're just going to make a new set of classes in order to keep the wheelchair project self-contained.

Phoebe starts by creating a new directory called **`Painters`**. Next, she adds an interface called **`IFramePainter`**. This is the key to the Bridge pattern. This interface defines a complex system of classes that specify a color painting system. This is one side of the bridge.

On the other side of the bridge is the wheelchair class. We can freely expand and modify either side of the bridge without affecting the other side.

The **`IFramePainter`** interface looks like this:

```
namespace WheelchairProject.Painters;
public interface IFramePainter
{
```

The interface requires five properties. **`PaintColorName`** allows you to give a name to the color combination. The remaining four are the values for **`Cyan`**, **`Magenta`**, **`Yellow`**, and **`Black`**:

```
    public string PaintColorName { get; set; }
    public int Cyan { get; set; }
    public int Magenta { get; set; }
```

```
        public int Yellow { get; set; }
        public int Black { get; set; }
```

Next, Phoebe adds the requirements for two methods. This is for the benefit of the robotic machinery that will assemble and paint the wheelchair. The system needs to first mix the paint color, then apply it:

```
        public void MixPaint();
        public void PaintFrame();
    }
```

The next thing we need is a concrete class to implement the interface. Phoebe remains focused on the *Plano Wheelchair*, and so makes a class called **`PlanoWheelchairPainter`**:

```
namespace WheelchairProject.Painters;
public class PlanoWheelchairPainter : IFramePaint
{
```

Phoebe leaves the five properties mentioned earlier as auto-properties:

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

```
        public int Cyan { get; set; }
        public int Magenta { get; set; }

        public int Yellow { get; set; }
        public int Black { get; set; }
```

The **`MixPaint`** method is complicated and highly proprietary. Kitty and Phoebe's lawyer won't let me show you the real code. So, we'll have to settle for some placeholder code:

```
        public void MixPaint()
        {
            Console.WriteLine("Mixing in Cyan: "
            Console.WriteLine("Mixing in Magenta: " +
            Console.WriteLine("Mixing in Yellow: " +
            Console.WriteLine("Mixing in Black: " + B


            Console.WriteLine("Mixing complete!  The
        }
```

Likewise, the **`PaintFrame`** method refers to some proprietary robotics APIs, so once again, we'll keep it simple by way of an example:

```
        public void PaintFrame()
        {
```

```
            Console.WriteLine("Applying " + PaintColo
        }
    }
```

The next change we need to make is to add the interface through composition to the **`Wheelchair`** class. Phoebe opens the **`Wheelchair`** class and adds a property:

```
public abstract class Wheelchair : WheelchairComp
{
    public IFramePainter FramePainter { get; set;
```

There's just one thing left to do. We need to add the bridge implementation to the builder so that when the builder builds a wheelchair, it gets painted in the same process. The builder really ties everything together!

The first change will be to the **`IWheelchairBuilder`** interface. Phoebe just adds one new method definition:

```
public interface IWheelchairBuilder
{
    public void Reset();
    public void BuildFrame();
    public void BuildAxleAssembly();
```

```
        public void BuildCasterAssembly();
        public void BuildSeat();
```

She adds the definition here:

```
        public void BuildFramePainter();
```

The rest of the code hasn't changed:

```
        public Wheelchair GetProduct();
    }
```

The interface updates, and suddenly Phoebe sees some red squiggly lines indicating a problem. Since she changed the interface, the **`PlanoWheelchairBuilder`** class is wrong because it doesn't have the new method. Phoebe opens up the **`PlanoWheelchairBuilder`** class and adds the missing method:

```
        public void BuildFramePainter()
        {
```

Phoebe instantiates a concrete **`PlanoWheelchairPainter`** class and sets it up with the new wheelchair paint color:

```
```

```
        var painter = new PlanoWheelchairPainter

        {
            PaintColorName = "Green-Eyed Judy",
            Cyan = 79,
            Magenta = 22,
            Yellow = 100,
            Black = 8
        };
```

Next, she sets the property on the **`_wheelchair`** instance inside the builder. After that, she calls the methods to mix the paint color and paint the frame:

```
        _wheelchair.FramePainter = painter;
        _wheelchair.FramePainter.MixPaint();
        _wheelchair.FramePainter.PaintFrame();
    }
```

SLAP!

A loud sound echoed through the lab as Kitty and Phoebe high-fived each other. Tom let out a good old-fashioned Texan "*Yee haw!*"

Over the next week, the team would test, debug, and refactor the code. If everything went to plan, Bumble Bikes could be shipping their high-quality, low-cost wheelchairs to rehabilitation centers and pediatric hospitals all over the world.

Summary

Kitty, Phoebe, and Tom were delighted at their progress. There is no doubt they would continue to refine their software over the coming weeks to make the system product-ready. They managed to get a lot done in a short time because they planned their software by designing first and implementing second.

You might have noticed a huge gap between the project proposed and what was delivered. We only worked on the *Plano Wheelchair* because the team decided this chair represented the minimum viable product. Neither the *Maverick* nor the flagship product, the *Texas Tank*, were built in this chapter. I left this as a challenge for you. Practice what you've learned and try implementing the *Maverick* and powered chair diagrams on your own.

We also saw a lot of interplay between patterns. The Builder was leveraged in conjunction with the Bridge, Singleton, and Composite patterns. This resulted in all the complexity being handled in one place. When these patterns were introduced, they were presented

one at a time. Now we see them fitting together like interlocking pieces of a puzzle.

Questions

1. What was the point of making the Builder pattern's director class a Singleton? Do you agree with the design decision? Why, or why not?
2. What is the advantage of embedding the composite structure into the normal object graph of the wheelchair versus making a separate object graph as we did in *Chapter 4* with the bicycle project?
3. What is the process in your favorite IDE for generating missing members for an interface?

Further reading

Practical Remote Pair Programming by Adrian Bolboacă: https://www.packtpub.com/product/practical-remote-pair-programming/9781800561366

Be sure to check out this book's companion website at https://csharp-patterns.dev.

8

Now You Know Some Patterns, What Next?

Over the course of writing this book, I asked a few friends, colleagues, and at least one of my many archenemies for input. Invariably, they would ask about a pattern they had studied in school or used on a project wondering why it wasn't included in this wonderful tome. The short answer to their question: the goal of the book is to focus on patterns that you can add quickly to your coding arsenal. Patterns that have a quick return on your investment of time and money.

Many of the patterns I chose to omit are very similar to the ones I included in this book. The patterns that made the cut were strictly my own preferences. These patterns have proven the most useful to me during my 25 years of experience as a celebrated and award-winning software engineer using C#.

By the end of this chapter, we will have covered the following topics:

- The GoF patterns we didn't cover will be discussed very briefly.

- Patterns outside of **object-oriented programming** (**OOP**). There are patterns that are not applicable to OOP—for example, patterns designed to describe database or network structure.
- How to create your own patterns. The GoF book documents a format for creating your own patterns.

Please note that I am only using diagrams. There is no code for this chapter. Likewise, there aren't any technical requirements.

Patterns we didn't discuss

I didn't cover all 23 patterns in the GoF book. I only covered about half. A number of factors went into deciding what to include and what to leave for a tacit discussion in this chapter. Some of the patterns are more troublesome than they are worth. The **Memento pattern** solves a problem that can easily be solved with a few .NET features. Some patterns were not included because they are very similar to another pattern we covered. Some are patterns you are never likely to need. The **Interpreter pattern** is only useful if you are inventing a new programming language. This is rarely done anymore owing to the popularity of **domain-specific languages** (**DSLs**). Tools exist for the construction of DSLs that preclude the need for the Interpreter pattern.

Here are the patterns from the original GoF book we didn't cover in this book:

- Prototype
- Adapter
- Flyweight
- Chain of Responsibility
- Proxy
- Interpreter
- Mediator
- Memento
- State
- Template Method
- Visitor

I'll discuss each of these patterns at a high level in the following sections.

Prototype

Making copies of objects can be tricky but if your object is flat, with no composition and just a few fields, it's no problem at all. However, making a deep copy of a complicated object that was built using composition and inheritance—and it's got a few layers of each—is harder. *Deep copying* refers to making an exact copy of an object from the highest level to the lowest level. The first step is to instantiate a new object from the same class. Then, you need to copy the values from all the properties and fields into the new object. You can only do this if

every field or property in the object you want to copy is `public`. This includes all the objects used to compose the object you're copying. Even if you manage to do this, the copy you create is subsequently of the dependent class used to make the copy.

Let's look at it another way. You're beginning a project with the *Transylvanian Historical Society*. Your job is to copy, brick by brick, a castle owned by one Vlad von Dracula. The castle's copy will be a museum in another town. There's a catch: the castle's drawbridge is up and you're not allowed inside.

It isn't hard to make a copy of the outside parts of the castle you can see, but it is impossible to duplicate the interior—especially that creepy crypt in the basement. That is, unless you had inside help. A man named Renfield offers his help. He has full knowledge of the castle's interior because he never leaves its walls. If you can get Renfield to help you from the inside, making a copy of the castle isn't going to be difficult.

Prototype is a Creational pattern that enables you to copy an object brick by brick. It works by delegating the copy job to the object itself. In short, it's an inside job. The copy operation itself is called **cloning**. An object capable of cloning itself is called a prototype.

Have a look at the following diagram:

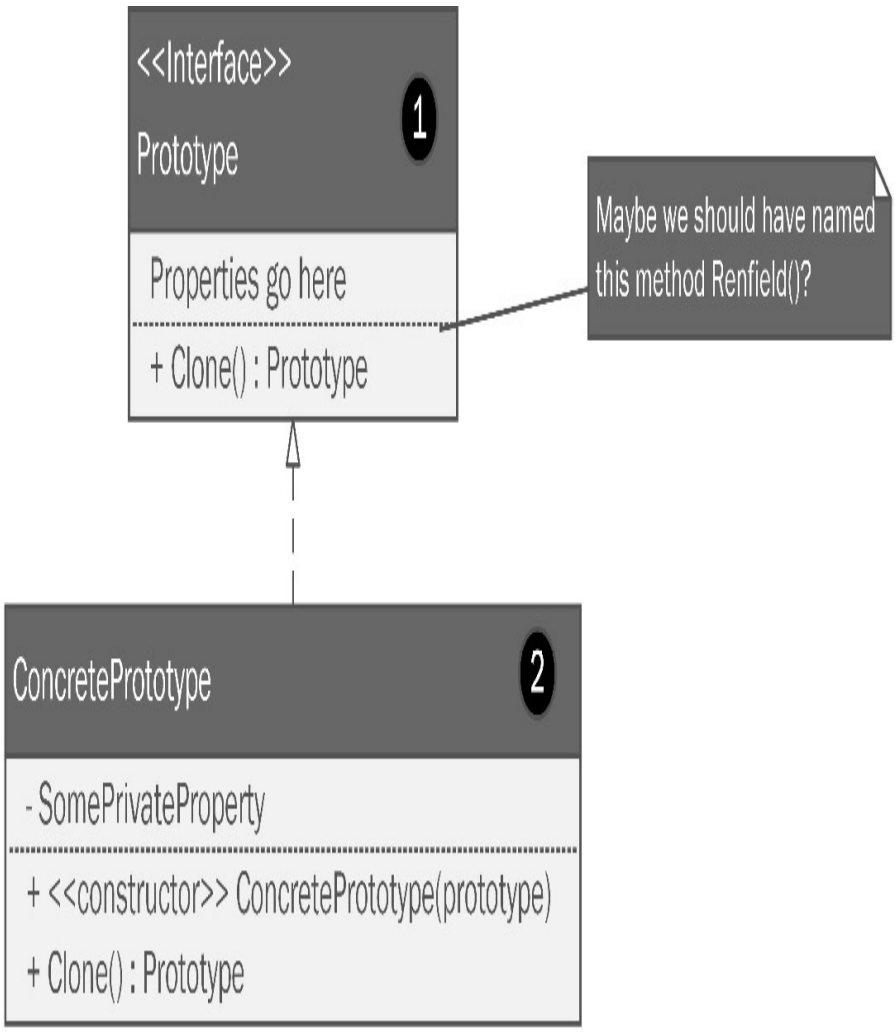


Figure 8.1: The Prototype pattern is used when you want to be able to make deep copies of an object.

Let's review the Prototype pattern diagram by the numbers displayed on it, as follows:

1. The **`Prototype`** interface defines the method that will perform the cloning operation.
2. The **`ConcretePrototype`** implements the Prototype interface and provides the implementation of the **`Clone`** method. If your object

structure is complicated, this method is positioned to be able to see everything inside the object where it lives. This method might be refactored so that its name is *Renfield*. He's the guy inside the castle who can help you clone it. The **`Clone`** method likewise can provide you with details needed to copy the object without losing any blood.

The Prototype pattern can be very useful if you need a bunch of classes copied from a small set of objects defined by concrete subclasses. Think back to our bicycle factory. Let's say there are half a dozen bicycle configurations that are very popular. Let's further presume that Phoebe's robots need a new instance of the **`bicycle`** object in order to manufacture a physical bicycle. In this case, a **`bicycle`** object is instantiated, the bicycle is manufactured, and the object is destroyed.

It would make sense in these circumstances to make a collection of "master copies" of the popular bicycle models and configurations. These masters could be cloned instead of the software needing to run an expensive builder method to generate a new bicycle conforming to a commonly ordered configuration.

Adapter

I keep two sets of wrenches in the back of my Jeep. One set is a common 3/8" mechanical ratchet set. One time I was driving home through Oklahoma from vacation in the Ozark mountain range. We had opted to take a scenic route by driving the back roads through a forested area of the state. We were cruising along, and I ran over a board in the middle of the road. It was unavoidable. I immediately switched my car's display to the tires, and over the course of a few miles, I could see the pressure in one tire gradually decreasing. I had a nail in one of my tires. I pulled over at the first opportunity and found some level ground so that I could safely jack my car and change the tire.

My wife and two girls were with me. We had to take all the luggage out of the car to get to the jack. If you've ever tried to change a tire with factory-supplied tools, then you understand I was hot, frustrated, and stuck on the side of the road. A pickup truck pulled up behind us, and the driver, wearing a cowboy hat, jeans, and a t-shirt, offered to help us out. I could see an array of expensive power tools in the back of his truck. I accepted his offer. We loosened the tire's lug nuts with his power wrench, and in minutes, we were back on the road. I vowed the first thing I'd do when I got home was to get me the same kit he had.

The minute I brought my new power wrench home, I wanted to use it on more than just changing a tire. After all, flat tires don't happen all

the time. I wanted to use it with my other ratchet set. Unfortunately, the 3/8" sockets from my other set don't work with the ½" power tool. If you're not familiar with these tools, check out the following screenshot:

VAN HORN
TOOL CO.
12V 5AH

Figure 8.2: My power impact wrench can only use the socket wrench sockets if I use an adapter.

The power wrench is like a normal wrench, except that the drive square is bigger on the power tool. The drive square is the part of the tool where you attach the sockets, which have a square hole in them. There's no way a 3/8" socket can fit onto a ½" drive square. That is, it didn't fit until I found an adapter. An adapter lets me use one interface —such as a 3/8" socket—with a different interface, such as the drive square of a ½" power socket tool.

The Adapter pattern does the same thing for your classes. An adapter implementation allows two classes with different interfaces to be used together. If my wrench problem were expressed in **Unified Modeling Language** (**UML**), it might look like this:

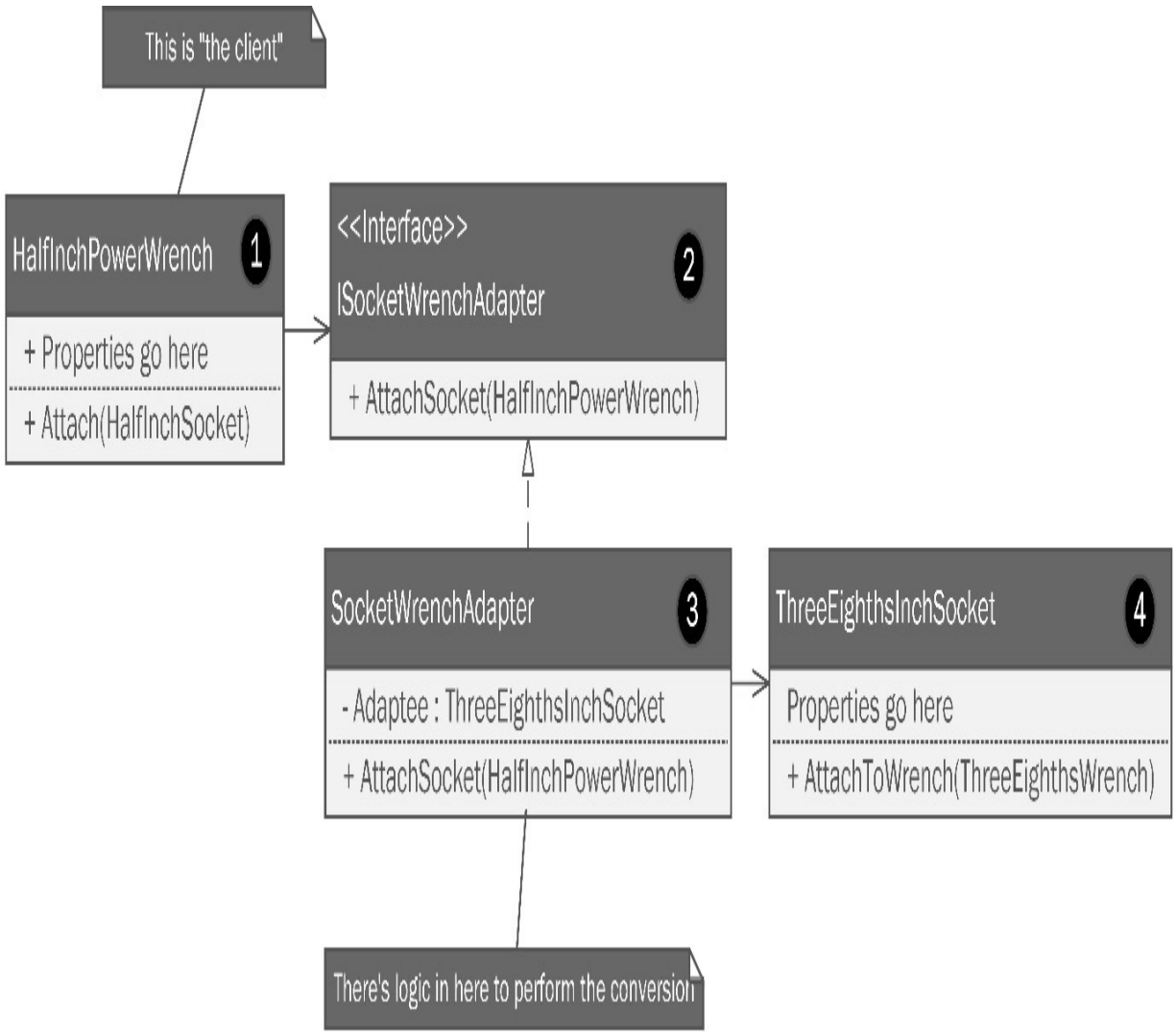


Figure 8.3: A class structure following the Adapter pattern is used to allow a class following one interface to work seamlessly with a different interface.

Let's break this down, as follows:

1. The power wrench is the client and we need a way to attach the smaller 3/8" sockets to the ½" drive square. To do this, we need an adapter.

2. The adapter should be described with an interface to prevent tight coupling. This interface calls for an **`AttachSocket`** method, which is implemented in the concrete adapter class.
3. The concrete adapter class implements the interface and contains a method that accepts something from the client that can be adapted.
4. The **`ThreeEighthsInchSocket`** class represents the incompatible interface you'd like to connect to your client.

In short, the adapter implements the client interface—**`ISocketWrenchAdapter`** in this example. It also wraps the class with which we are interfacing—in this case, **`ThreeEighthsInchSocket`**. The adapter receives the drive square from the power tool when it calls the **`AttachSocket`** method in the adapter class. The internals of this method perform the logical operations needed to convert that input into something the adaptee can use. In software parlance, the client would be making a call to the adapter class. The method called would provide conversion logic that converts what was passed into the method so as to be compatible with the adaptee.

This can be a very useful pattern when you need to leverage third-party or legacy with new work.

Flyweight

Have you ever had a thought that keeps you awake at night? Or maybe you have a thought that wakes you up? You're sound asleep and then you awake with a jolt. Your half-slumbering brain just figured out what's wrong with line 37.

Late one night in a fit of heuristic frenzy, Kitty wonders what would happen if one day, demand for Bumble Bikes exploded. What if the small company was inundated with thousands of orders? The robotic manufacturing system Phoebe built instantiates a bicycle object each time it builds a bicycle. Each bicycle object takes up space in the server's **random-access memory** (**RAM**). Kitty decides to try a simulation using a development server. After a few load tests, she determines she can load 1,000 instances of a bicycle object complete with the bridged painter system. Once the object count goes above 1,000 concurrent objects, the server starts to slow down. Once she reaches 2,000 objects in memory, the system nearly grinds to a halt and is unusable.

One obvious solution to this problem is to order more RAM for the server. Of course, this makes it a hardware problem. We are software developers. Maybe there is a way to solve this problem by strictly using software patterns. Maybe a small adjustment will prevent us from having to ask the pointy-haired boss for several thousand dollars in upgrades.

The Flyweight pattern is used to move some of the shared elements of each object's state into one shared object. Sometimes, you can shift a large amount of data out of memory and into one shared object. You see this anywhere you have a high object count. For example, if you use C# with Unity to develop game software, your game might have hundreds, or even thousands, of enemies. Maybe you're leveraging a homemade particle system with thousands of shiny moving particles, or maybe you're doing a factory simulation where thousands of bicycles are being manufactured by a small cadre of robots.

Consider the following diagram:

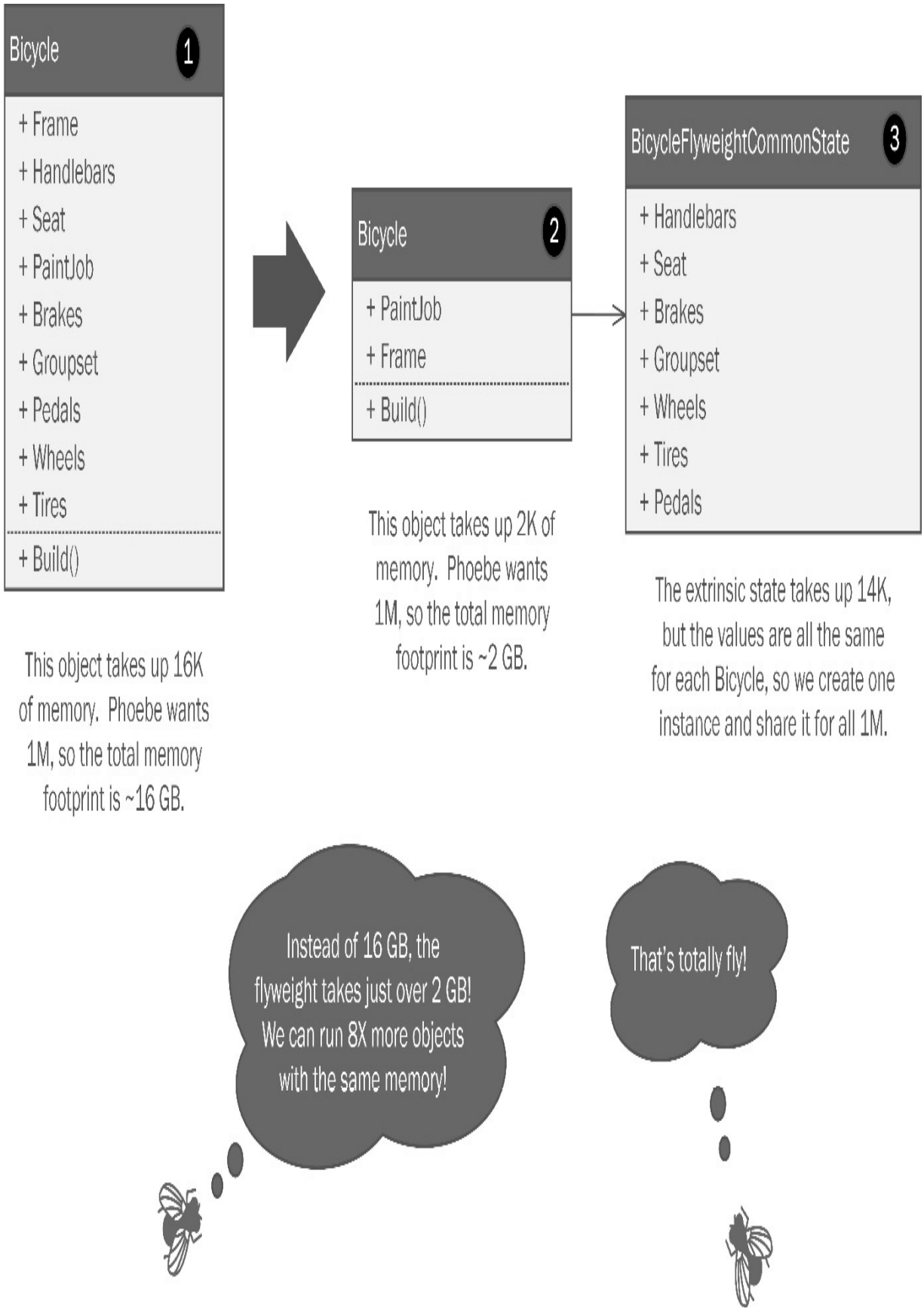

Bicycle
1
+ Frame
+ Handlebars
+ Seat
+ PaintJob
+ Brakes
+ Groupset
+ Pedals
+ Wheels
+ Tires
+ Build()
This object takes up 16K of memory. Phoebe wants 1M, so the total memory footprint is ~16 GB.
Bicycle
2
+ PaintJob
+ Frame
+ Build()
This object takes up 2K of memory. Phoebe wants 1M, so the total memory footprint is ~2 GB.
BicycleFlyweightCommonState
3
+ Handlebars
+ Seat
+ Brakes
+ Groupset
+ Wheels
+ Tires
+ Pedals
The extrinsic state takes up 14K, but the values are all the same for each Bicycle, so we create one instance and share it for all 1M.
Instead of 16 GB, the flyweight takes just over 2 GB! We can run 8X more objects with the same memory!
That's totally fly!

Figure 8.4: The Flyweight pattern entails shifting duplicated state variables into a separate object that can be shared to reduce memory footprint.

Let's break it down by the numbers in the diagram, as follows:

1. Here, we have a **`Bicycle`** class with all its parts. For our simulation, we're just going to make road bikes. The ordering system lets customers customize the bicycle frame with a special paint job, or pick from one of the standard colors. The seat can also be swapped out for one of several models. However, the rest of the properties offer no customization options. They will be the same for every road bike we build.
2. Now, watch what happens if we shift the parts of the state that don't vary into a separate class. Here, our **`Bicycle`** class only contains those properties that might be different between each bicycle we make.
3. The elements of state that are common for all objects are called the extrinsic state and are represented in the **`BicycleFlyweightCommonState`** class. It's a bit of a mouthful, but I wanted to be sure you recognized which part was the flyweight. We can make one instance of this class and share it with all 1 million of the **`Bicycle`** classes in Phoebe's simulation.

Let's say the `Bicycle` class weighs in at 16 **kilobytes** (**KB**) of memory usage upon instantiation. If Phoebe wants to generate 1 million of them, that's going to consume 16 **gigabytes** (**GB**) of memory. Once we shift the repeated state to the flyweight object, we chop out 14 KB for one object. Since we only instantiate it once, the memory footprint comprises only the intrinsic state. We instantiate 1 million bicycle objects at 2 KB, or total consumption is 2 GB plus the negligible 14 KB for a single instance of the extrinsic state in the flyweight class. That's a nearly 8X reduction! I don't know about you, but I would strut for a week if I could reduce my software's memory footprint by 8X just by rearranging some classes!

Chain of Responsibility

The success of Bumble Bikes isn't owed entirely to Kitty's designs nor Phoebe's brilliant robotics. Bumble Bikes also has a focus on quality and **quality assurance** (**QA**) is completely automated. A system of cameras does a series of inspections using **artificial intelligence** (**AI**) via OpenCV. The inspection begins with the frame, then moves to the handlebars, the drivetrain components, the brakes, the wheels, the tires, and finally the seat. The logic used by the AI to perform the inspection is not contained in a single method. That would violate the **single-responsibility principle** (**SRP**). Instead, each inspection's logic is encapsulated within a separate method for each inspection. The inspection happens sequentially, beginning with the frame. If any

of the inspections fail, the bicycle is flagged as a defect and set aside for remediation performed by a human bicycle mechanic.

You can see an overview of the QA checks in the following diagram:

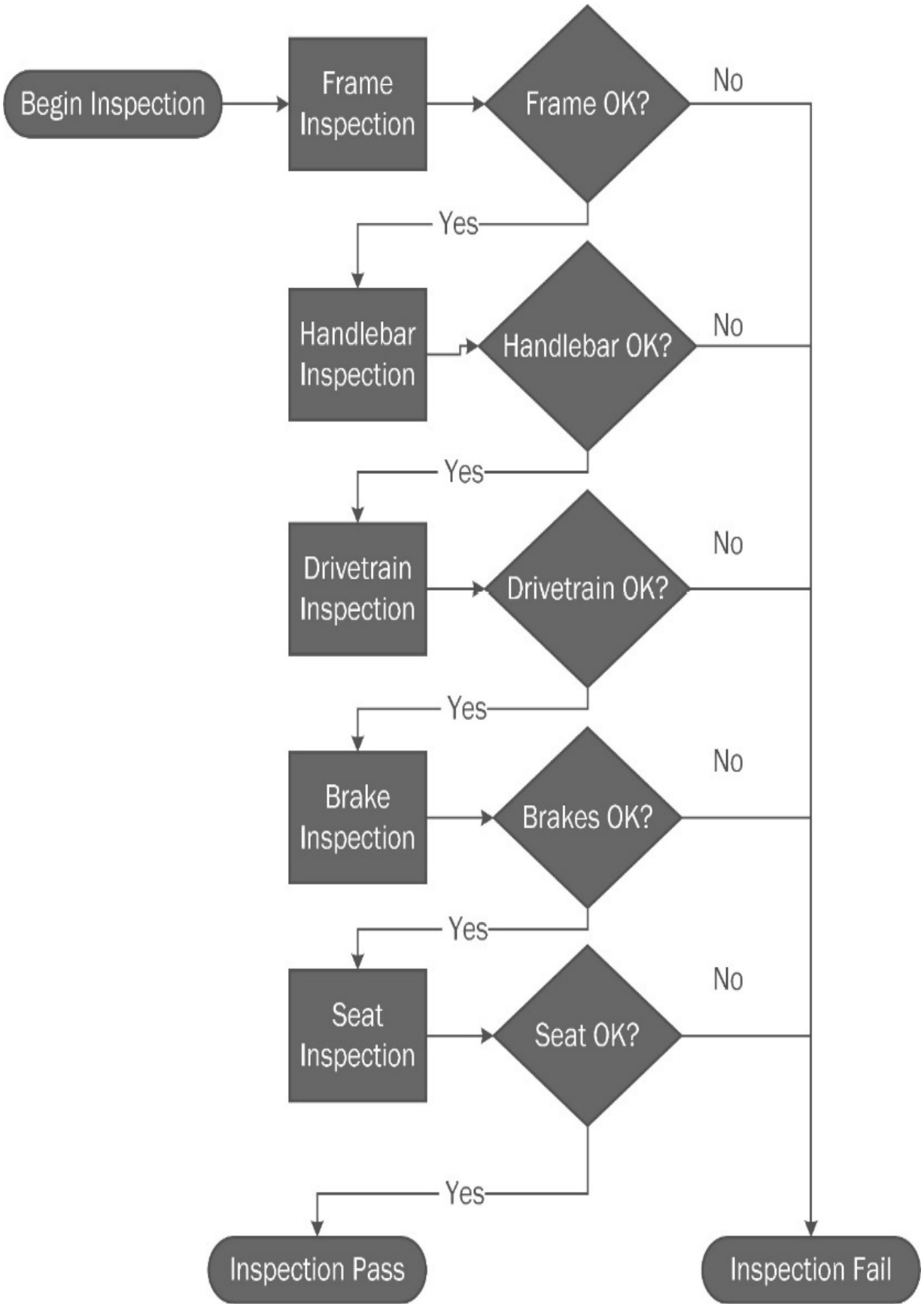

Begin Inspection
Frame Inspection
Frame OK?
No
Yes
Handlebar Inspection
Handlebar OK?
No
Yes
Drivetrain Inspection
Drivetrain OK?
No
Yes
Brake Inspection
Brakes OK?
No
Yes
Seat Inspection
Seat OK?
No
Yes
Inspection Pass
Inspection Fail

Figure 8.5: Sequential quality assurance checks on a bicycle are one example of Chain of Responsibility.

This sequentially ordered set of inspections uses the Chain of Responsibility pattern. If any of the inspections fail, the remainder of the inspections is not run. This is how Toyota makes cars. If a defect is found on the assembly line, everything stops until the problem is corrected.

The pattern itself takes on the form seen here:

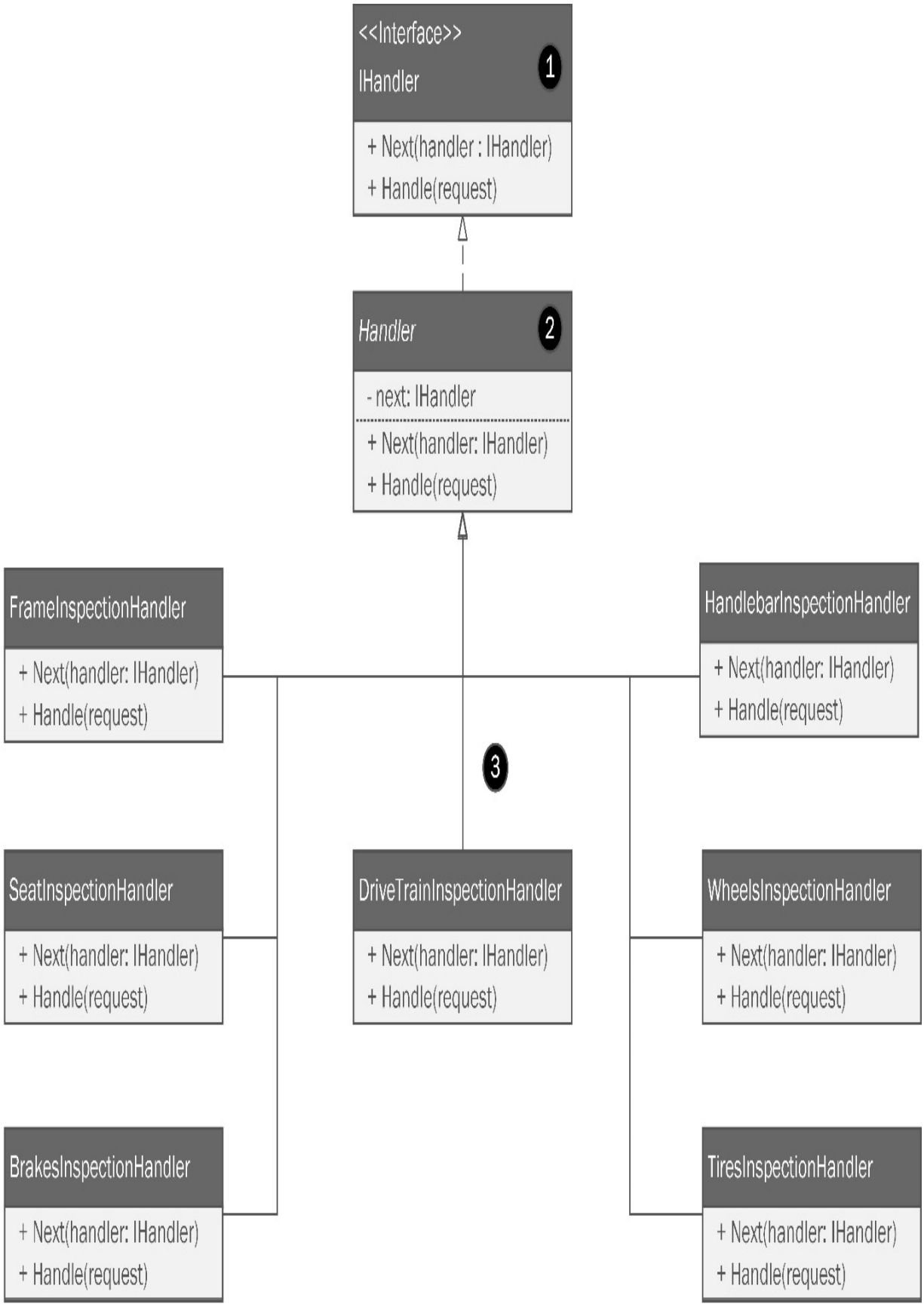

<<Interface>>
IHandler
1
+ Next(handler : IHandler)
+ Handle(request)
Handler
2
- next: IHandler
+ Next(handler: IHandler)
+ Handle(request)
FrameInspectionHandler
+ Next(handler: IHandler)
+ Handle(request)
HandlebarInspectionHandler
+ Next(handler: IHandler)
+ Handle(request)
3
SeatInspectionHandler
+ Next(handler: IHandler)
+ Handle(request)
DriveTrainInspectionHandler
+ Next(handler: IHandler)
+ Handle(request)
WheelsInspectionHandler
+ Next(handler: IHandler)
+ Handle(request)
BrakesInspectionHandler
+ Next(handler: IHandler)
+ Handle(request)
TiresInspectionHandler
+ Next(handler: IHandler)
+ Handle(request)

Figure 8.6: The Chain of Responsibility pattern is used anytime you need a stateful sequence of events.

Let's break it down, as follows:

1. An interface, usually called **`IHandler`**, defines two methods. The **`Handle`** method defines the method signature for the implementation of the inspection logic. The **`Next`** method allows us to move to the next inspection, assuming the current step passes.
2. An abstract handler class defines the next handler, which is a common property among the concrete classes.
3. The concrete classes inherit from **`Handler`** and implement their inspection logic with the **`Handle`** method, which overrides the base class method. Similarly, the **`Next`** method overrides the base and is used to pass the torch on to the next runner or, in our case, pass the current inspection and move to the next.

Flowchart logic has been around since the Turing machine. It should come as no surprise there is a pattern to encapsulate this universal concept.

Proxy

One day, Kitty and Phoebe get a call from Dr. Eloise Swanson. Dr. Swanson has started a new business called *U.S. Robots and Me-*

chanical Men. Her company's newest product is a high-end **software development kit** (**SDK**) for robotic control. Dr. Swanson had studied Phoebe's designs while in graduate school at the **Massachusetts Institute of Technology** (**MIT**) and thought Bumble Bikes would make a good partner for beta testing.

The SDK was very effective and easy to use. However, it wasn't a drop-in replacement for the software Kitty and Phoebe had written. First, there's no way an SDK can fully replace custom-tailored software for any business. The second problem was bloat because the SDK was designed to work with any robotic system. A great deal of code was used to account for every possibility.

Kitty and Phoebe found that the SDK was a good fit to control their painting robots. However, it was only needed when a custom paint job was ordered that had never done before. Once a paint job was performed, it was cataloged and the color formula could be reused. The SDK from *U.S. Robots* made much shorter work of the paint jobs, at the cost of slow initialization and a large memory footprint. If the girls were to couple their code to the SDK, which of course they wouldn't, all their work would be slowed down by an expensive initialization process they would rarely use.

The solution is to lazy load the SDK objects only when they are needed. The P roxy pattern allows you to define and use a placeholder for

an object. The proxy could then load the big, slow SDK classes only when they were needed. The rest of the production process would remain unhampered.

Let’s examine the structure of the Proxy pattern in the following diagram:

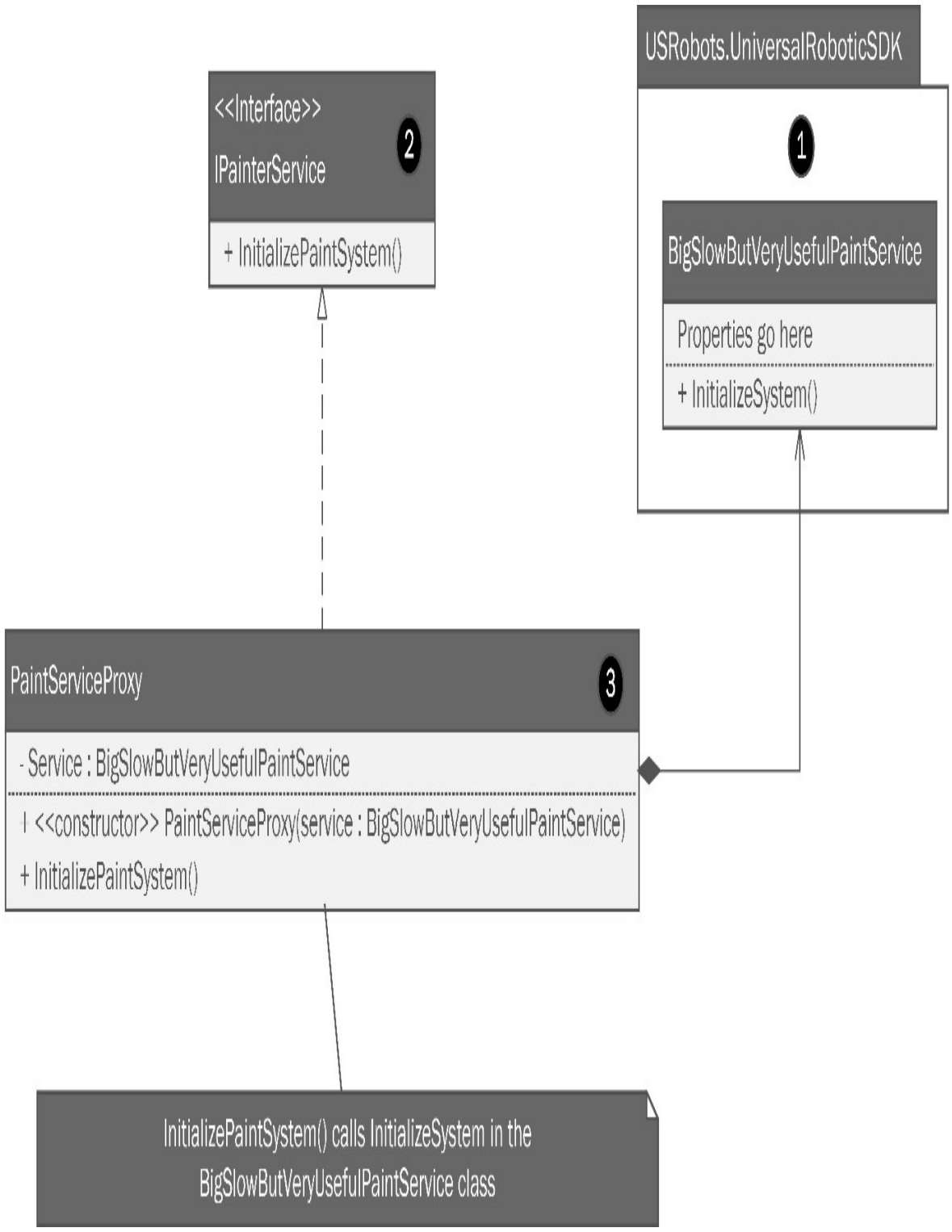


Figure 8.7: The Proxy pattern can substitute a simple object for a more complicated one until the complicated one is actu-

ally needed.

Breaking it down will help us understand further, so we'll do just that:

1. We have this SDK from a third-party vendor. We know tightly coupling to a third-party vendor's SDK is a bad idea in the first place. In this case, the SDK has a method we want to use, but its containing class has a big, slow constructor, and the object is rarely used. We need a way to lazily load this object only when it is needed while also preventing a tight coupling.
2. We create an interface to prevent tight coupling. Our client software can require the interface, and any changes in the future are fine as long as we can keep the interface.
3. Then, we create a wrapper that holds the SDK instance. The wrapper holds an instance of **`BigSlowButVeryUsefulPaintService`**, which might take several minutes to instantiate. That's a long time, considering we only need one method. Our wrapper instantiates **`BigSlowButVeryUsefulPaintService`**, then calls the expensive **`InitializeSystem`** method only when it is really needed. Since we bore the instantiation cost, we can store the instance in a private property and use it again if the need arises. That might sound like a singleton, but singletons ensure only one instance is ever created. Here, we are just reusing what we have already made, which isn't quite the same thing.

Remember, usese the Proxy pattern anytime you need a placeholder for a third-party, legacy, or overly expensive object. This will to prevent tight coupling and defer expensive operations that might happen on instantiation for when they are needed.

Interpreter

The Interpreter pattern is one you will likely never need. This pattern is used when you have a custom language you need to interpret that can be expressed using **abstract syntax trees** (**ASTs**). With the rise in popularity of **domain-specific languages** (**DSLs**), entire toolkits, such as JetBrains Meta Programming System and Visual Studio Enterprise's Modeling SDK, make creating a custom language interpreter a relatively simple project.

Since DSLs are well beyond the scope of a book on patterns, I will list a reference to the DSL tools I've mentioned here in the *Further reading* section at the end of the chapter.

Mediator

It isn't difficult to imagine a future for Bumble Bikes where their automated manufacturing system might expand and become more complex. Right now, we have the Builder pattern controlling the robotics that build bicycles and wheelchairs. There are two physical factories,

and each factory specializes in just a few products. All that could change in the future.

I worked for an aircraft manufacturer on a joint venture with another aircraft manufacturer. Our company made the powerplant (engine) and assembled the final aircraft. The partner company built the body of the aircraft. Other partner companies supplied avionics, which are the electronic flight-control systems present in modern aircraft. Another company still manufactures military components that I'm not allowed to talk about.

Imagine if Bumble Bikes had a slew of manufacturing operations like that. Even if they were all collocated in one plant, the level of signal communications between all the different robotic manufacturing systems could become very chaotic. If each system were to communicate with all the other systems directly, we'd very quickly find ourselves in trouble.

If you've ever created a large web application using HTML and JavaScript, you have run into a similar problem. You have dozens of different pieces of code modifying the **Document Object Model** (**DOM**) in response to potentially hundreds of signals emanating from user interactions, timers, **Representational State Transfer** (**REST**) **application programming interface** (**API**) calls, third-party SDKs such as jQuery, and third-party advertising sites injecting code into your site

for monetization. When a system such as this reaches critical mass in terms of signal complexity, it becomes slow and nearly impossible to debug and maintain.

Let's consider one more example: a natural disaster such as a hurricane or tornado. What if all the **first responders** (**FRs**) were able to contact each other directly? Every firefighter could radio every policeman, who could radio every **emergency medical service** (**EMS**) unit. The EMS responders could talk directly to triage nurses and Red Cross volunteers. All this happens on one big open communication channel. What are your odds of surviving a disaster in this environment? Open direct communications can sometimes be a good thing, but I think we all realize it doesn't scale. Now, imagine hundreds of objects in a piece of software that all have direct access to all the other objects on the stack. You see the problem, right?

What's needed in all these circumstances is some form of dispatcher: a central hub for communications. The Mediator pattern embodies this role. A mediator acts as a central hub for all communications and routes requests to the objects that need them in a controlled manner. Let's examine the diagram shown here:

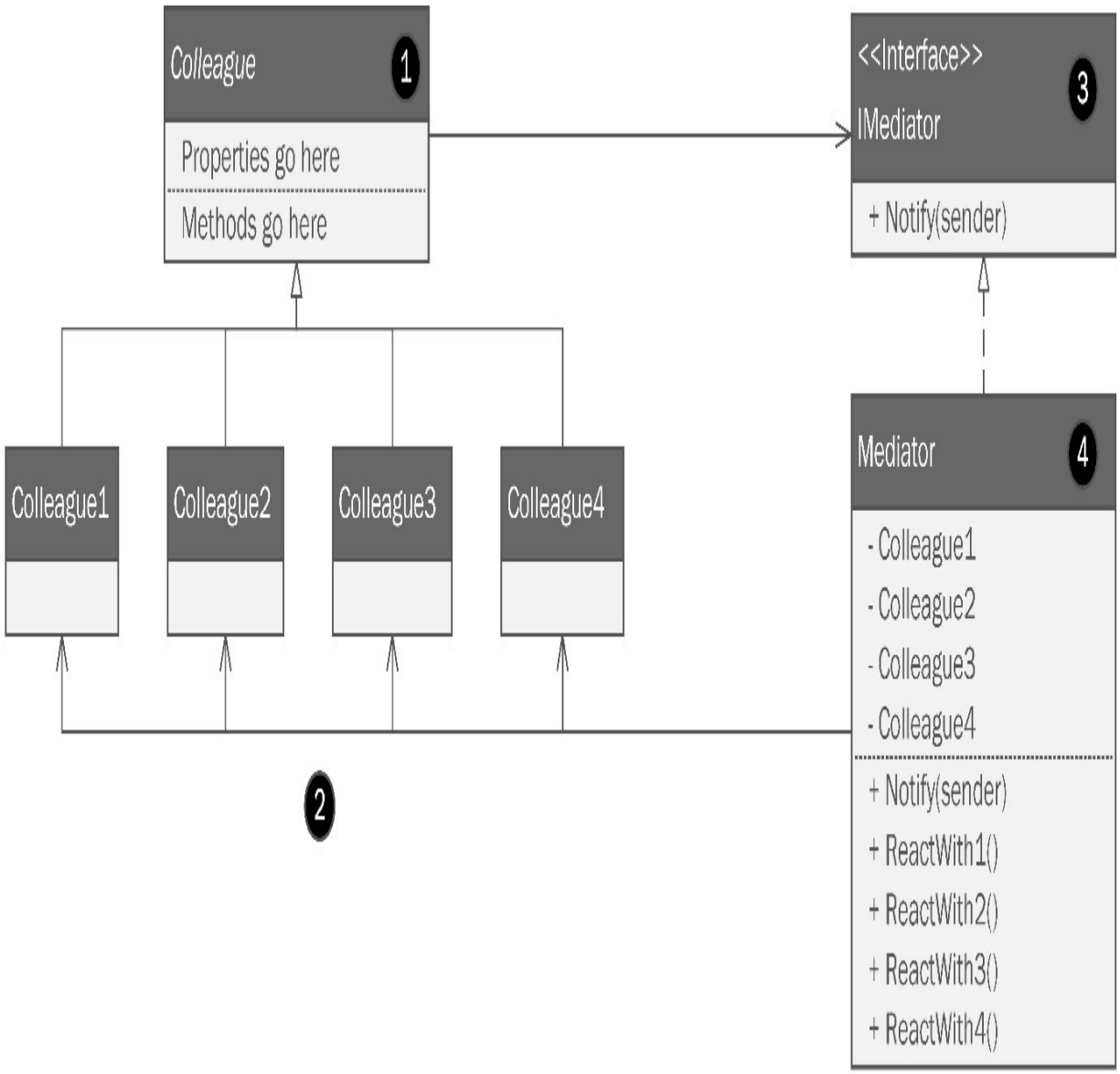


Figure 8.8: The Mediator pattern involves a single centralized object that directs calls between objects.

By the numbers in the diagram, it works like this:

1. This is a base class for the colleague objects.
2. You have a bunch of colleague objects that need a centralized way to communicate. Here, I just have four, which is probably no cause

to run to the Mediator pattern. But imagine 400 objects all communicating directly with each other! Like I said before, direct communications don't scale!

3. We make an interface called **`IMediator`**, which sounds cool when you say it out loud. This, as usual, prevents tight coupling.
4. Now for the good part. A central object based on the **`IMediator`** interface contains instances of all the objects and has defined communications channels between them in the form of methods that I called **`ReactWithX`**, where X is the number corresponding with the object sending the signal. **`ReactWith1`** calls an appropriate method on **`Colleague1`**, and so on.

This pattern is designed to simplify the communications process, but it usually results in a very large object with lots of internalized component instances and methods for communicating. You have to weigh the complexity of the **`Mediator`** class against the benefits of the centralization it offers. On the one hand, as a developer, it's nice to have one class where you can drop a breakpoint in your IDE. Like a lion scoping out the local watering hole, eventually, all message traffic passes through this one class, making it simpler to find bugs.

On the downside, the class itself can become unwieldy.

Memento

Have you ever saved a game or used undo in your favorite editor? If so, then you've likely interacted with the Memento pattern. The best analogy for a memento is a cool but outdated technology. When I was in grade school, the hottest camera was called a Polaroid. Most cameras back then involved a roll of film. You would shoot your pictures on the camera, then take the film to a drug store to be developed. It took about a week to get your pictures back. However, with a Polaroid, you could take your pictures, which were ejected from the camera and self-developed within a few minutes. The development process seemed to go a little faster if you shook the picture, which gave rise to a popular song lyric, and accompanying dance move, "Shake it like a Polaroid picture".

Back then, we called these pictures *snapshots*. This term is used today in conjunction with the Memento pattern. A snapshot, like a Polaroid picture, is a representation of an event or place at a particular point in time. So it is with software as well: a snapshot represents the state of an object at a point in time. Like a photo, a software snapshot can be saved as a memento—something to remind you of that one time when your object was in that state.

Text editors such as your IDE or Microsoft Word are constantly tracking the state of your documents and saving mementos as you type. When you hit *Ctrl*/*Command + Z* on your keyboard, you can go back sequentially to earlier and earlier mementos.

At first glance, this seems easy. All you'd need to do is create a **`List<>`** object to hold, say, the last 100 state changes your user has made. Maybe you store a memento every 30 seconds or so. That's easy. It's that easy only if every object in your state and every object used in inheritance and composition has 100% public properties. If the entire state is public, you wouldn't need a pattern for this.

The real impediment here is the same one we encountered with the Prototype pattern earlier. We were trying to copy Count Dracula's castle, but we weren't allowed inside the front gate. Making a shallow copy of the outside of the castle is straightforward, to make a full copy of the whole castle requires an inside actor. This is equally true in the Memento pattern. In the memento's case, the inside actor is a nested class. The nested class has access to the outer class's state and can store our suggested **`List<>`** object containing our undo history. Let's look at the diagram shown here:

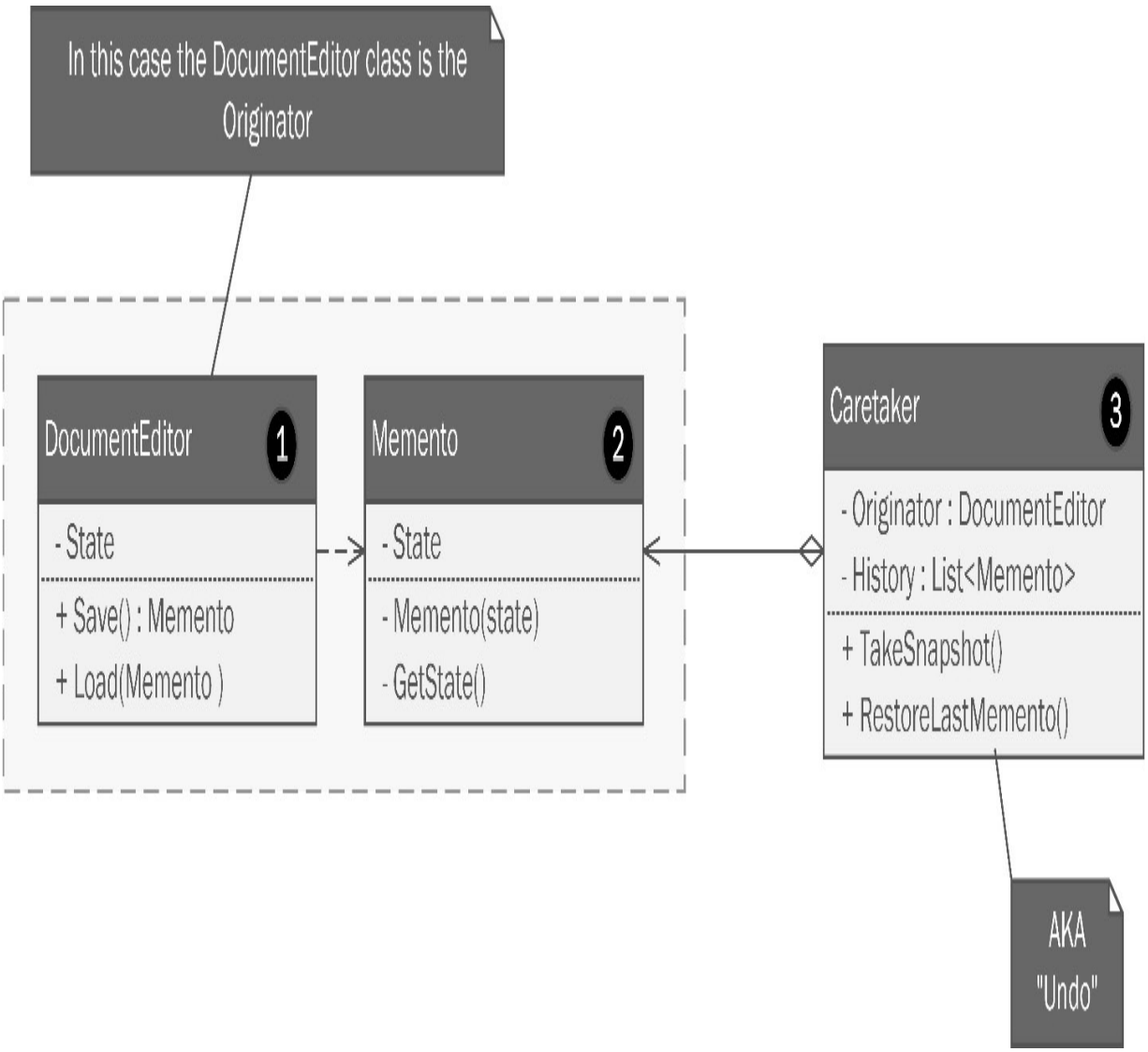


Figure 8.9: The Memento pattern.

Since the most familiar example is a text editor, I used that for the diagram. Let's review the numbered parts of the diagram, as follows:

1. This class represents the **`DocumentEditor`** class, which is our client. The memento pattern calls this the originator since this is where the stored state comes from.

2. When the editor needs to save an undo state—or, for that matter, a file—it can use the **`Memento`** class for storage. Note that all the internals are private. The **`Memento`** class is nested inside the originator; it is defined *inside* the **`DocumentEditor`** class as a nested class. Besides being locked down, you should consider **`Memento`** classes immutable: once you create them, you should never change them. The private properties are set via the constructor and subsequently never modified. This means when you implement, there should not be any setter accessor methods for these properties.
3. There isn't a standard way to draw internal classes in UML, so I put the **`Caretaker`** class outside the dashed box as a visual clue that there's something different about this class. Remember, **`Memento`** is inside the **`DocumentEditor`**, and not via composition. The **`Caretaker`** class, in turn, contains the **`DocumentEditor`** object inside the **`Originator`** field. The **`Caretaker`** class, then is responsible for creating and restoring mementos, as well as the undo history.

The Memento pattern can be difficult to get right, and you might go a long way in your career without needing it. Even if you land a job at a company that makes a document editor, chances are you'll be using one of the many excellent third-party **user interface** (**UI**) controls, such as those from Telerik, that have already implemented this for you.

The other point worth mentioning is that you can achieve a nearly identical effect using C#'s serialization libraries. Usually, saving the application state also entails persistence. Good editors let you undo the last 100 changes. Great editors let you do that between editing sessions. You can turn off your laptop, fly across the world, boot back up, and your undo history is still available because it's serialized (saved) to a file somewhere on your hard drive.

The .NET Framework gives us a slew of serialization options, including **`System.Runtime.Serialization`**, **`System.Runtime.Serialization.Json`**, and **`System.Text.Json.Serialization`**. Each contains a set of classes designed to make serializing your objects to files fairly trivial. However, you'll still have problems with private properties. Thankfully, Microsoft has supplied us with the **`DataContractSerializer`** class that helps you get around this limitation without resorting to an intricate class structure or worrying about the memento's immutability.

State

I'll confess looking back at the last eight chapters, I probably should have covered this pattern. You'll use it, especially if you work in game development with Unity 3D. If you're not familiar with this tool kit, it's essentially a game engine capable of creating AAA games for consoles, PC, Mac, mobile, and the web. I taught game development with

Unity for many years at a local college and so I have a soft spot for it, despite never having worked as a professional game developer. The Unity 3D game engine uses a finite state machine to control the animation of your game characters. It uses a visual editor to define the states and the animation to use when the game character's state changes.

The State pattern entails changing an object's behavior based on changes to its internal state. An object's state is conceptually just the values of all its properties at a point in time. If you make a game where you run around the countryside fighting zombies, you might define your zombies with different behaviors. Most of the time, they're just shambling around randomly looking for fresh BRAINS!

When something with a brain comes into view, the zombie's behavior changes. It shambles towards the brain-toting organism while hissing "BRAINS!" over and over. Once the zombie is within arm's reach, its behavior changes again. The zombie attacks!

We have one object with three behaviors, all controlled by the internal state. When we put the behaviors and states together, we form what's called a finite-state machine. It is finite because zombies have a finite number of behaviors. In this case, the number is 3. The finite state machine can be diagrammed as shown here:

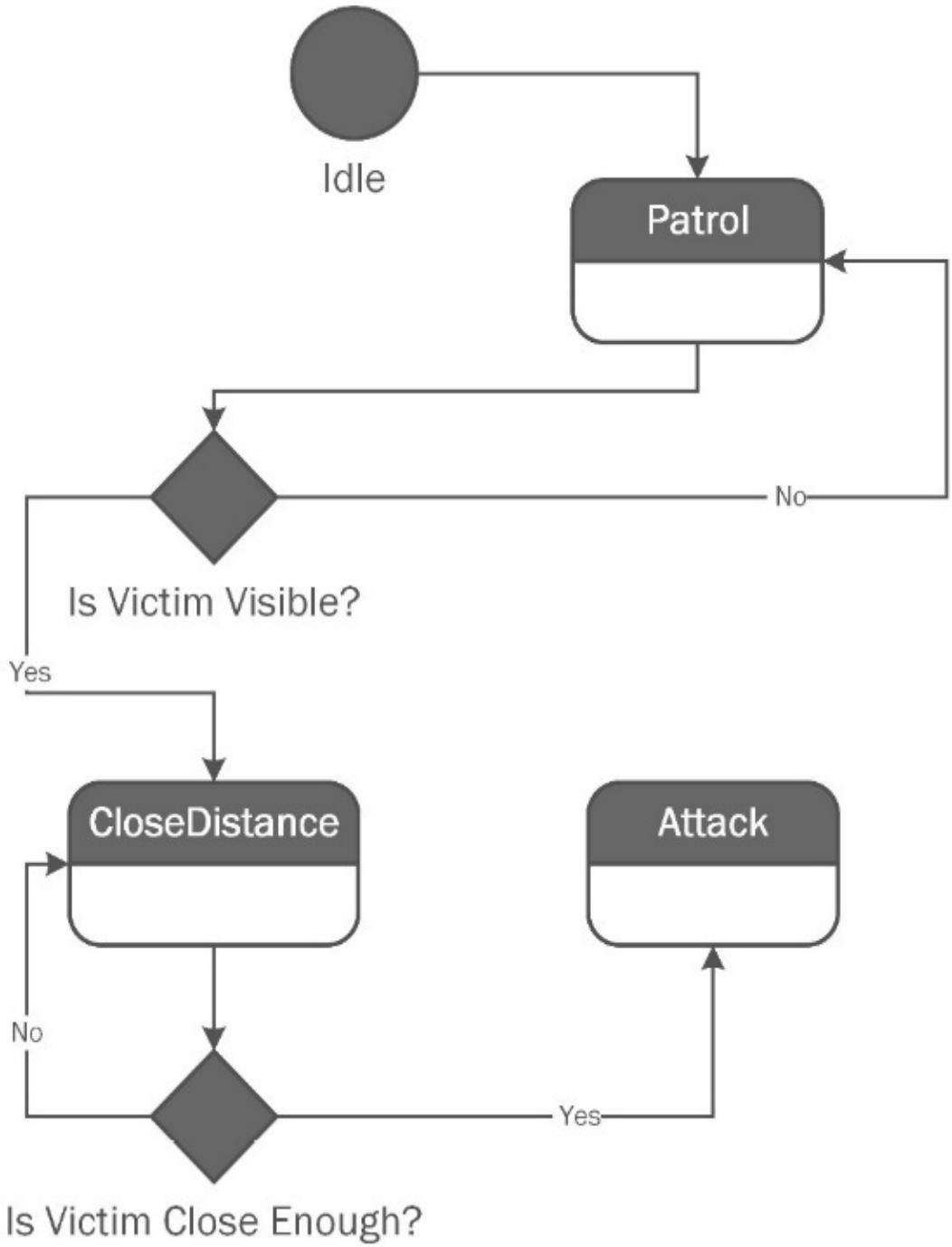


Figure 8.11: A finite state machine representing zombie behavior in a video game.

This state diagram shows the transitions between the different states. The zombie can’t randomly attack a victim that is too far away. It has to see the victim and close the distance first. A diagram of the pattern appears here:

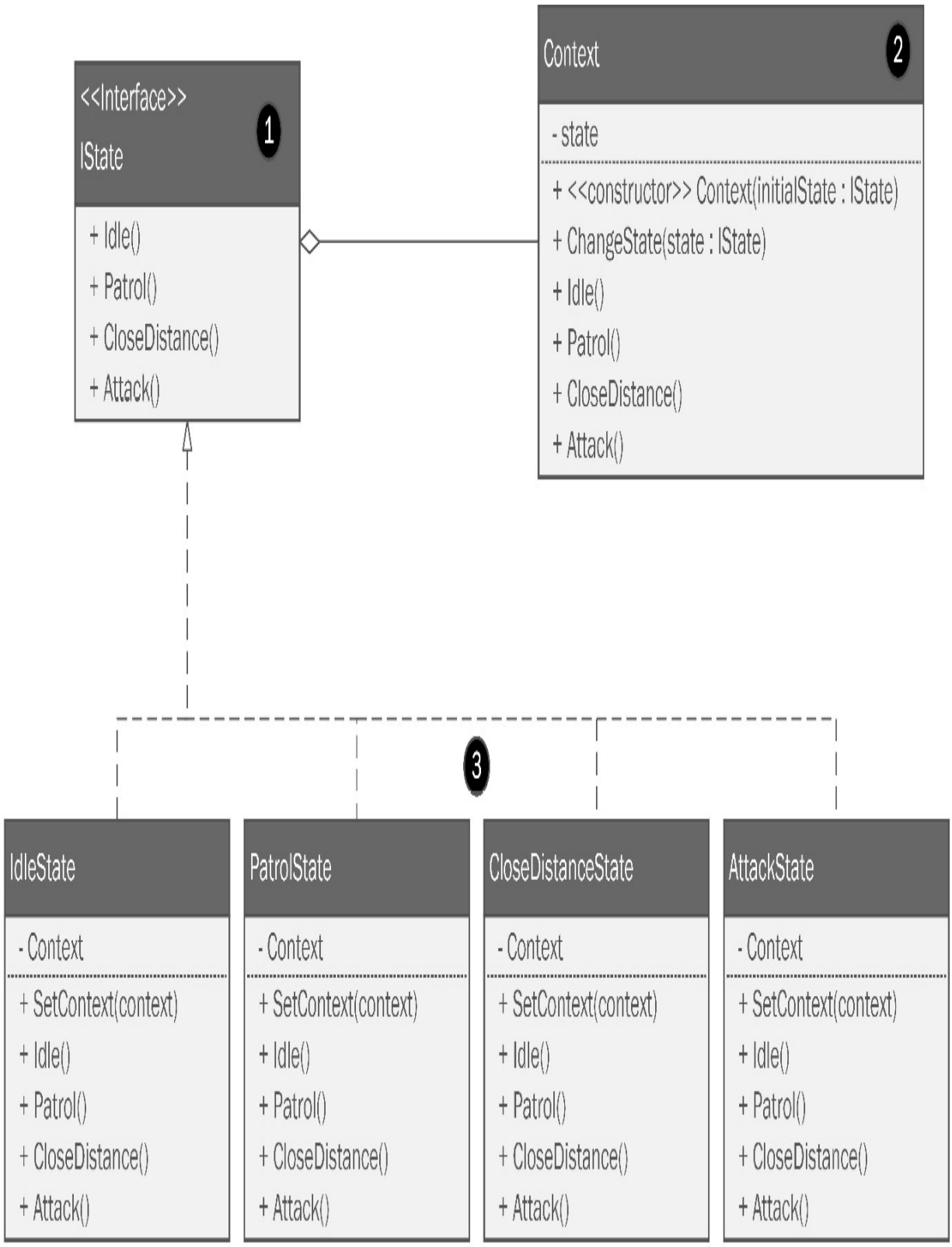


Figure 8.12: The State pattern.

Let's break it down, as follows:

1. The `IState` interface defines the behaviors that are possible. Our zombie can patrol, close distance to a visible victim, and attack.
2. The `Context` class holds a reference to an object implementing the `IState` interface, which is used to communicate with concrete state objects.
3. Concrete state objects contain state-specific methods.

The context object can be used to change the state by swapping concrete state objects held in the private state property. This means the context can enforce any logic for state transitions.

I didn't include the State pattern earlier because it is very similar to the Strategy pattern covered in *Chapter 5* . Kitty and Phoebe used the Strategy pattern to create a navigation system for bicycles. The behavior of the system changed depending on what type of terrain the user requested.

Template Method

The Template Method pattern is very similar to the Strategy pattern, which we covered in *Chapter 5*. This is a behavioral pattern that allows you to define the structure of an algorithm but defer the implementation to subclasses that override the actual logic but not the structure.

In *Chapter 5* , Phoebe designed a navigational computer for bicycles. It used the Strategy pattern to compute navigational routes depending on whether the rider wanted a route via paved roads, unpaved gravel roads, or extreme terrain. We could have done the same thing with the Template Method pattern, which is why I didn't feel the need to cover both.

The structure can be seen in the following diagram:

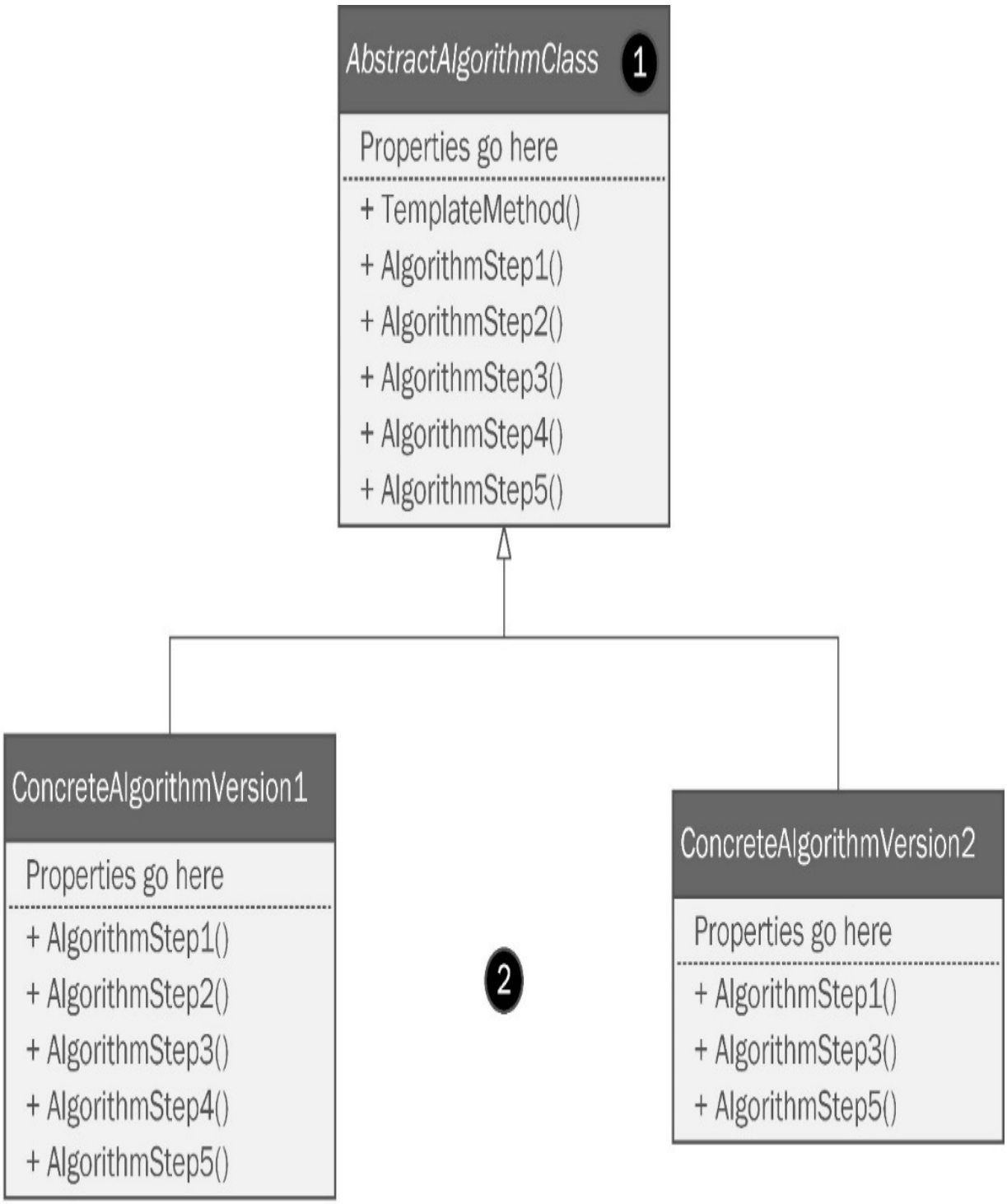


Figure 8.13: The Template pattern.

Let's look at the numbered parts, as follows:

1. The abstract template class defines the structure of a class that forms an algorithm. The steps to the algorithm are defined as abstract and can be overridden by a concrete implementation in a child class. The steps, however, are called in a method within the template class. In my example, I called it **`ExecuteTemplate()`**, which is not overridden. **`ExecuteTemplate()`** calls the steps in order, and this never changes. The template defines overrideable logic, but the structure is consistent each time the template is used.
2. These are the concrete classes that override the algorithm steps. Note the **`AlgoImplementationA`** class overrides all the steps in the parent class. The **`AlgoImplementationB`** class just overrides a few.

This pattern allows you to make very flexible algorithm implementations using nothing more than common household inheritance.

Visitor

The Visitor pattern is another Behavioral pattern designed to help you "bolt on" new behaviors to existing objects. Its motivations are focused on SOLID principles (head back to *Chapter 2* if you need an explanation of this acronym). The open-closed principle is honored because you are adding behavior without modifying existing classes. The SRP is honored because usually, the behavior you are adding is

new and may have little to nothing to do with the purpose of the original class.

The idea behind the name comes from the idea that an object encapsulating new behavior can visit an existing, established class, allowing it to perform the new behavior. You're literally teaching an old dog object new tricks.

Imagine a world where your body has an ability slot. You can slot in a new behavior as easily as slotting a memory card into a camera. Need to learn to fly a helicopter to escape evil secret agents? Slot the card with flight training and you can instantly fly any civilian or military aircraft! Do you need to cook like a 5-star chef? Slot a card with cooking skills and you'll be able to defeat Gordon Ramsay or Bobby Flay in any cooking competition! Do you need moves at the nightclub? Slot a card with nightclub skills and you'll dance the night away! Just be careful with that last one. There's no telling what other skills might be on the card.

Let's shake that last example off and look at the following diagram:

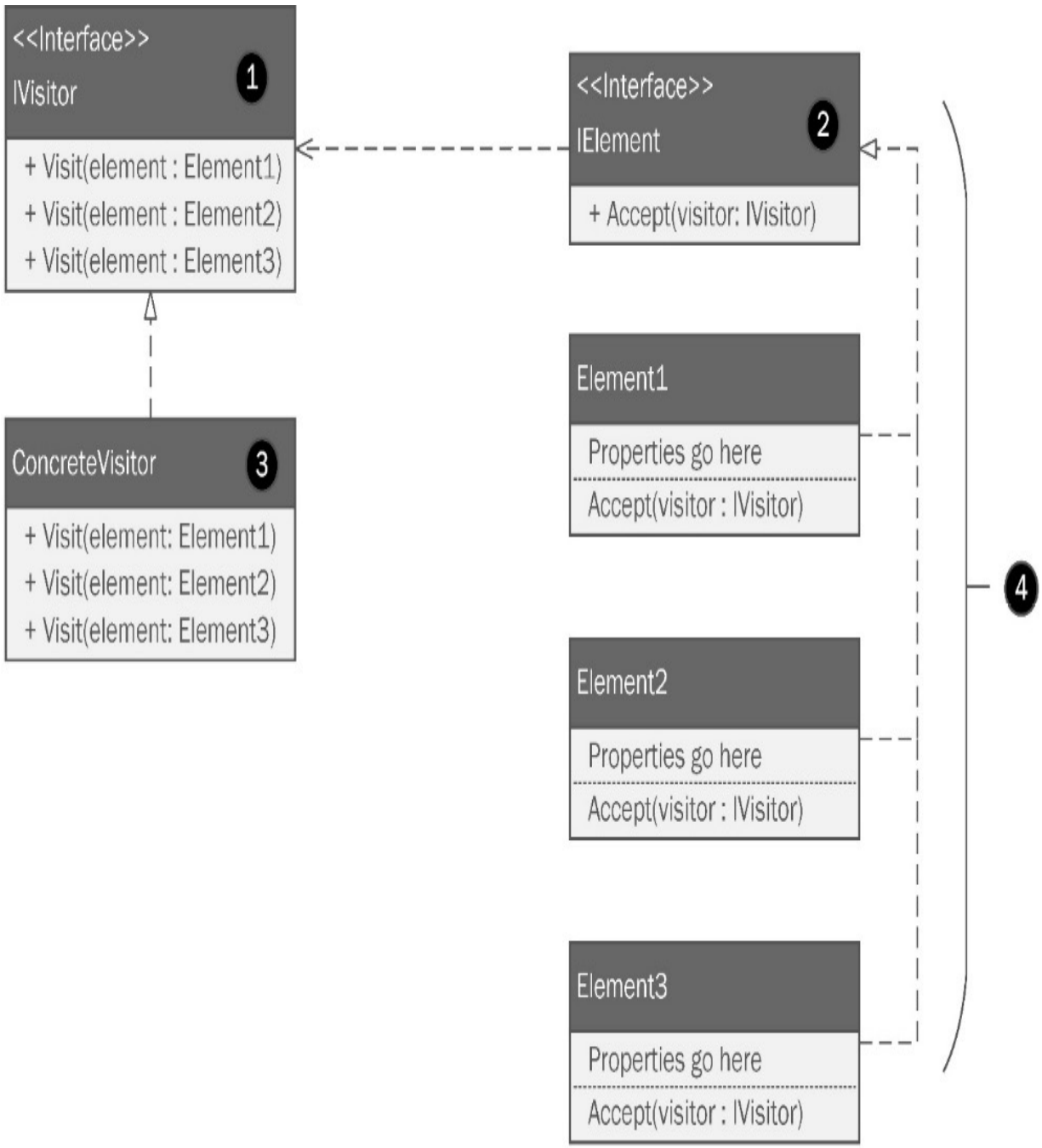


Figure 8.14: The Visitor pattern.

Let's break it down, as follows:

1. Our **`Visitor`** interface defines a set of methods required to implement a new superpower you wish to confer on an existing object graph. We're fortunate that C# supports method overloading. Method overloading refers to a language's ability to reuse method names as long as the method signature is different. If you're not sure what this means, check out *Appendix 1* at the end of this book. We cover it there. Not every OOP language supports this capability, but C# does. Our interface defines several methods with the same name but with different argument types.
2. A second interface called the **`Element`** interface defines how visitors are accepted. If you want new abilities in our future world, you'll have to get a slot added to your skull so that we have someplace to insert our ability cards. Likewise, your class will need an additional method based on this interface. You don't need to change any existing methods or logic in the class. You just need to add a method to accept visitors.
3. Concrete visitor classes implement our interface and provide the new behavioral logic. Note that we did not use the **`IElement`** interface as the argument type. The concrete object type passed in determines which method is run based on method overloading.
4. These are your existing classes implementing the **`Element`** interface.

A few things come to mind when I think of the Visitor pattern in C#. First, it feels a lot like the Decorator pattern, but with a focus on object

graphs instead of individual classes. Maybe that's just me?

Another similar pattern is the Composite pattern covered in *Chapter 5*, and again in *Chapters 7* and *8*. Kitty and Phoebe used the Composite pattern to perform a set of calculations for weight and cost on a graph of objects used to comprise a bicycle drivetrain. Tom improved the design by weaving the Composite pattern into the object graph at design time. The difference is that the Composite pattern operates from outside the objects in the graph whereas the Visitor pattern is inserting new behavior.

Finally, the first time I read about the Visitor pattern, I immediately thought of extension methods. Extension methods are unique to C#. They allow you to add behaviors to existing classes using a separate class file. They are not an implementation of the Visitor pattern, but if your needs are simple, you might want to start with extension methods. If extension methods come up short, then try the more complex Visitor pattern.

Patterns beyond the realm of OOP

The field of OOP was really just the start. There are patterns beyond the realm of OOP that you have very likely heard of, but perhaps

didn’t know were codified patterns.

Software architecture patterns

An obvious area for finding more patterns is within the domain of software architecture. It may seem like we’ve been talking about software architecture this whole time. We have. However, software architecture isn’t bound to OOP. Every pattern in this book relies on using C# because it is an OOP language. Software architecture patterns span every language and really help us define systems, not just enhance the structure of our code. Let’s look at some examples you have probably heard of before now.

Client-server pattern

We actually covered this one—we just didn’t call it out as a pattern. This pattern involves **peer-to-peer** (**P2P**) architecture consisting of a server, and usually many clients. The clients make requests to the server and do something with the result. This is pretty much how the internet works.

Microservices pattern

The Microservices pattern entails breaking large, monolithic applications into small, self-contained but interdependent services. The idea is to take the SRP to its ultimate implementation. Imagine a

REST API with a handful of endpoints that together serve only one purpose such as password reset. Instead of password reset being built into a bigger API that does dozens of other things, the password reset becomes a microservice.

The idea is that it is easier to maintain tiny single-purpose APIs. The trade-off comes from the latency introduced between interdependent system calls and the resulting complexity of your network topology.

Model-View-Controller (MVC) pattern

Surely you've heard of this one! You see it in web applications when you split the application code into three layers, as follows:

- A model layer that contains code that represents the data model, usually via an **object-relational mapper** (**ORM**).
- A view layer that represents the **user experience** (**UX**).
- A controller layer that represents the logic needed to accept requests from the view and process data returned from the model layer. This configuration embodies the ideals behind the **separation of concerns** (**SoC**).

Publish-subscribe (pub-sub) pattern

This is another popular pattern you'll see in many guises. Sometimes you'll see object-level implementations such as what we saw in the

Observer pattern covered in *Chapter 5* . You'll also see higher-level architectural implementations in software such as Redis, RabbitMQ, and Apache Kafka.

In every case, the idea is to allow communication from a central source. Messages are sent to a publisher who in turn publishes the messages to relevant subscribers. This is vital to distributed architectures.

Command Query Responsibility Segregation (CQRS) pattern

This pattern is meant to solve the situation that arises when database queries for data occur far more frequently than database updates. If you have a reporting system where reports are generated but rarely subsequently modified, this is a good pattern to learn. Its solution generally entails segregating data that rarely changes but is commonly queried into a separate database called a data warehouse. This can boost performance on reads and writes because the responsibilities are segregated. The trade-off comes from the implementation cost, which often involves standing up a separate database server to handle the segregated loads.

Data access patterns

Data access patterns occur wherever you have traditional code interacting with a relational, or even non-relational, database. Since relational databases have been around since 1970 and are largely unchanged and completely ubiquitous, it shouldn't be surprising that they have their own set of patterns. I cut my teeth with a book by Clifton Nock titled *Data Access Patterns: Database Interactions in Object-Oriented Applications*. I'll list the book's details in the *Further reading* section at the end of the chapter.

Here are just a few patterns from that book you'll probably recognize.

ORM

Ever heard of **Entity Framework** (**EF**)? As a C# developer, you'd have to have lived beneath some sort of highly academic rock to have never heard of Microsoft's flagship ORM. An ORM's job is to map data between relational structures in **Structured Query Language** (**SQL**) databases such as SQL Server to objects in C#. If a pattern is a solution to a problem that occurs frequently, this might be the most important pattern in software development.

Implementations of this pattern, usually encapsulated in a third-party library, or in our case, within the .NET Framework, allow developers to work solely with C# objects. With an ORM, you never need to create or update table structures in your application's database. You

never need to figure out complicated joins or concatenate long SQL statements in your code. You create a set of objects that represent your database structure. You then query those model classes and work with data operations to manipulate your database.

The upside is if you don't know—or don't like—SQL, you never need to work with it directly. The trade-off is in application performance. EF code is known to run significantly slower at scale than direct connections to the database. It also adds one more layer of dependencies to your application. It's just one more thing that can go wrong. If you're sensing I'm not a fan, you'd be right. I like ORMs for small projects with a limited number of users. For example, if you're making an app for internal use and you have limited time to create it, using an ORM might help you get the job done more quickly.

If on the other hand you are making an application for a large user base, eschewing the convenience of an ORM in favor of implementing our next pattern directly gives you tremendous control over performance by allowing you to leverage the full capabilities of your database software.

Active Domain Object

The **Active Domain Object** (**ADO**) pattern is a fun one because when you see ADO, you probably think of either Microsoft's **ActiveX**

Data Objects (**ADO**) or ADO's older cousin **Data Access Object** (**DAO**). Then again, that might only be if you're over 35 years old. It's been a good while since those technologies were front and center.

The ADO pattern encapsulates data access and relevant object implementations. Their aim is to remove any direct interaction with the database outside the pattern implementation. Sound familiar? That's pretty much what Microsoft ADO and DAO were designed to do. I doubt this is a coincidence.

Most developers don't use this pattern, having generally moved to working with ORMs such as EF. However, if you have SQL skills, you can often make a more performant application by accessing the database directly via native drivers.

Demand cache

My wife and kids do this all the time. Just kidding, but not really. There are quite a few cache patterns. A demand cache is embodied by a cache that is populated lazily "on demand". The main force at play here is you don't know when the data will be needed, but you would like to only pay the price of retrieving the data once.

My application includes a demand cache as part of a web application. Some of the queries in my application return large recordsets or include considerable processing that can take several seconds to com-

plete. That doesn't sound bad, but for a web application, it is an eternity. My demand cache method checks the cache (which in my case is Redis) first when data is requested. If it's there, the data is served from the cache. If it isn't there, I retrieve and cache it. The first customer to request the data pays the performance penalty, but every customer after that gets the data very quickly.

Transaction

This is another ubiquitous data processing term that is really a pattern. In database lingo, a transaction refers to multiple SQL statements that must be completed as a unit. For example, if you have an automatic teller machine and you want to withdraw money from one account and add it to another, you need this to occur as one unit of work—a transaction.

You withdraw $100 from account A and add $100 to account B. If either SQL statement fails, you need to consider the whole transaction a failure and roll back all the changes entirely.

Optimistic and pessimistic lock

Nothing drives application developers crazier than dealing with locks in the database. An optimistic lock takes place in order to prevent missing database updates. You find this pattern used in **inventory management systems** (**IMSs**) where up-to-the-second, on-hand

stock data is vital to selling that inventory. Last weekend, I bought a new clothes dryer at a popular store. My wife picked out the perfect dryer and the sign above the dryer said, “Order now and you’ll receive it in 3 days”. The salesman went to enter the order and found the units were not only out of stock, but the manufacturer wasn’t accepting back orders.

The sign assumed stock at the warehouse and was based on stale information. Now, replace the sign with a database query. The sales application queries to see if any dryers are in stock. At the exact same moment, at a store across town, someone else has just purchased the same model of dryer.

How can the clerk at the first store know whether there really is a dryer available? The optimistic lock pattern entails creating a version number on each row in a database table. The database in this scenario will optimistically assume the dryer is available if the row version number hasn’t changed since the transaction started and doesn’t lock the database row for update.

By contrast, in the same scenario, a pessimistic lock will prevent the clerk’s sale from going through if the second clerk’s transaction was in the middle of updating the inventory row data.

Creating your own patterns

Do you think you have an idea for your own design pattern? The GoF book presents a boilerplate documentation framework for publishing your own patterns. I won't duplicate it fully here, but I will outline it for you. It involves four essential elements, as follows:

1. Name and classification
2. The problem description
3. The solution description
4. Consequences of using the pattern

Let's talk a little about each section.

Name and classification

Every pattern needs a name that describes the pattern. Most of the patterns out there have names that make you think of general words from normal language. The word *memento* refers to a physical object that invokes memories of times past. The word *singleton* invokes the idea there's a single thing. Come up with a short, memorable name that invokes the idea behind the pattern.

You also need a classification. In this book, we observed the three classifications in the GoF book: Creational patterns, Structural patterns, and Behavioral patterns. You saw in the previous sections that other knowledge domains that use patterns have their own lists of

classifications. Maybe your pattern fits into an existing classification. If not, you'll need to invent a new classification.

It is also possible your pattern might have an alias; an "**also known as**" (**AKA**) name. The Decorator pattern is also known as a wrapper. Document any aliases for your pattern if any exist.

The problem description

This section of your documentation describes the problem your pattern aims to solve. When the GoF describe the problem, they break it into several smaller sections, as outlined here:

- *Intent* refers to the overall goal of your pattern.
- *Motivation/Forces* outlines why someone would use your pattern. This is deeper than "because it's cool" or because it solves a particular problem. When we discussed antipatterns in the first chapter, we described a set of forces that caused the antipattern to come into being. In this section, we're looking for similar driving factors around why your new pattern is relevant and useful.
- *Applicability* lists the context for the pattern. Make a short list of situations where the pattern is useful.

While it isn't necessary to fill in every single niche, you'll recognize elements we've used in this book.

The solution description

When describing the solution, you need to describe the elements used to construct the design. Expound on the relationships between the elements. Describe how the elements collaborate, and make sure to explain the responsibilities of each element. You might consider breaking this section into smaller chunks as well, as follows:

- *Participants* list classes, objects, interfaces, enumerations, and so on that participate in the pattern solution.
- *Collaboration* describes how the participants collaborate.
- *Structure* will contain UML diagrams of a generic form of the pattern, and perhaps a diagram of a more real-life concrete use case. This is the format I have used in every pattern I covered in this book. I find the generic diagrams less useful by themselves. If a concrete example follows, developers can look at both diagrams and more easily relate them to their work.
- *Implementation* contains a description of how to implement the pattern. In this book, I've used numbered diagrams to describe how the pieces fit together.
- *Sample Code* or *Pseudocode* mostly speaks for itself. I would recommend real-world examples instead of class A inheriting from class B, which uses class C in composition. That's too abstract. If you want people to use your pattern, find ways to make it relevant to their work.

Consequences of using the pattern

You should always discuss the positive outcomes and trade-offs that come with using your pattern. Most patterns have defenders and detractors. If you read up on patterns on the internet, you'll see a healthy debate about how some patterns might be antipatterns. We presented such a case when we discussed the Singleton pattern in *Chapter 3*. The more complex a pattern, the more likely it is to have trade-offs. The Template Method pattern is extremely simple. I doubt whoever came up with that one lost any sleep over thinking about possible negative outcomes. In contrast, the Façade pattern presented in *Chapter 4* presents a trade-off between the complexity of the third-party framework and ease of use for developers that don't need everything the third-party framework exposes.

Not everybody likes patterns

We demonstrated in *Chapter 3*'s coverage of the Singleton pattern that not everybody agrees that patterns are a positive contribution to the field of software development. The usual argument is design patterns are simply workarounds for incapable, inefficient, or incomplete OOP languages. Academic literature has shown as many as 17 of the 23 patterns within the GoF book become unnecessary when you use

languages such as **List Processing** (**LISP**) or Dylan. What? Who even uses those?

Another group of academic detractors advocates switching your paradigm from OOP to **aspect-oriented programming** (**AOP**) as a solution to all your problems. As you're hopefully aware, OOP aims to solve problems by modeling things in the real world. AOP aims to model behavior as crosscutting concerns. AOP is not supposed to be a competitor with OOP, yet some arguments place it that way.

The bottom line is there will always be a debate that says "If you would only switch to language X, framework Y, or paradigm Z, all your pattern problems will be solved!" My retort to that is to remind you to beware of the Golden Hammer!

Summary

Patterns are everywhere. There is a field called biomimicry that aims to study technology inspired by patterns in nature. It's difficult to talk about software development anymore without bringing up **artificial intelligence** (**AI**), whose main job is to find patterns in massive amounts of data using techniques called **machine learning** (**ML**). The software industry has been humming along now since 1843 when Ada Lovelace wrote what most consider to be the first computer program. In that time, we have collectively run into the same challenges

and frustrations over and over again. Eventually, we got smart enough to start writing things down and talking about them.

We've been learning patterns, which at their core are really just a way of communicating our best ideas as they relate to organizing and optimizing our code.

In this chapter, we briefly covered the patterns from the original GoF book that we didn't cover in the earlier chapters as part of this book's story. I left these patterns out for one or two reasons in each case, as follows:

- If a pattern was remarkably similar to another pattern already covered, I didn't cover it. The State pattern is very similar to the Strategy pattern. The Strategy pattern fit my story, so that's the one I used.
- If a pattern was very complicated and rarely used, I didn't cover it. The Memento pattern is a good example. There are easier ways to handle the use case of representing snapshots of objects such as .NET's serialization features that negate the need for a complicated Memento pattern implementation.

We went beyond GoF patterns, and even beyond OOP patterns to list some common and highly familiar patterns in other domains of software and database architecture. We concluded with a synopsis of

how you might go about documenting your own patterns should you ever discover a new one. I left you in this chapter with a final warning. You're going to be tempted to close this book and run through the office like a crazy person yelling "PATTERNS!" everywhere you go. Don't laugh. I've seen it happen. If you do this, expect some eye rolls. Not everybody thinks patterns are a good idea, and sometimes they are right. Patterns themselves can easily succumb to the Golden Hammer antipattern. Remember—patterns are here to simplify and improve your software. If they make things slower or more complicated, you must abandon them in those cases. Patterns are tools, not dogma.

There's just one loose end to tie up.

Sundance Square – Fort Worth, Texas

It was a hot spring day in Fort Worth, Texas. And it was the last day of the MS-150 bicycle rally. The MS-150 is an event that generates millions of dollars per year toward research into a cure for **multiple sclerosis** (**MS**). Thousands of cyclists in the Dallas area ride in the 150-mile, 2-day event. Most of the local bike shops have tents set up in Sundance Square, a shopping and entertainment district in the heart

of Fort Worth. Bumble Bikes, being a platinum sponsor, has a large tent set up at the finish line.

Tom, Lexi, and Karina are working the tent, giving water and high-fives to the intrepid few that cross the finish line. While upward of 3,000 riders start the rally, fewer than 10% actually finish. Most quit along the way when their knees give out or their equipment breaks. Tom and Lexi are watching keenly for a group of riders who they fully expect to finish dead last. It's a rally, not a race. The people who ride aren't competing against each other—they're competing with themselves to see if they have what it takes to complete the course. They start in Plano on day 1 and pedal 75 miles to Texas Motor Speedway where local **National Association for Stock Car Auto Racing** (**NASCAR**) races are held. The next day they start again but first, they do a lap on the speedway, then ride 75 more miles into Fort Worth.

"*There they are!*" Lexi yells as she squints toward the top of a hill in the distance. *Team Bumbles* is cresting the hill, all riding brand new Hillcrest bicycles: Kitty, Phoebe, and most of the Bumble Bikes employees, along with Kitty and Phoebe's father.

10 years have passed since his diagnosis. The chemotherapy and steroids had been ineffective in treating the disease. In fact, the steroids had caused a severe case of osteoporosis. The doctors had

given up, saying there was nothing further that could be done. Kitty and Phoebe had prayed countless times, ultimately joined by an informal coalition of local churches.

The wheelchairs Kitty and Phoebe built were extremely helpful. They had built and distributed over 1,000 wheelchairs free of charge to children's hospitals all over the world. Bicycle and wheelchair sales at Bumble Bikes grew steadily until it became the number 3 bicycle manufacturer in the world, and the only company to produce its products in the **United States** (**US**). MegaBikeCorp, the company where Kitty and Phoebe interned all those years ago, was ranked at number 19.

Their father had been using his *Texas Tank* for many years. The design had proved too expensive to mass produce, so they only made one. As much as he loved his *Tank*, every day he made it a practice to try to stand up. Most days he fell down. Until one day, he didn't.

As suddenly as it had started, one day it stopped. The doctors could not explain it. They had run out of ideas years ago. Kitty and Phoebe knew their prayers had been answered.

It took years of physical therapy. Electrical shock therapy was needed to get their father's voice box working again after the disease had nearly destroyed the muscles in his throat. He took shots in the stom-

ach every day for 2 years to cure the osteoporosis. He took short walks—a few blocks at first, but later it was a few miles. Month after month, and year after year, he got a little stronger. He traded his *Texas Tank* for one of Bumble Bikes' new electric bicycles with a motor. Tom was happy to take the *Tank* off his hands. The new electric bike offered a motor to assist with pedaling. The motor didn't do all the work—it just helped.

Eventually, he was able to ride a normal bicycle again, and today, he had ridden it 75 miles to the finish line with his daughters and a small armada of handicapped riders. Bumble Bikes had produced a hand-cranked bicycle for riders who couldn't use their legs and invited any rider who didn't have such a bicycle to ride with *Team Bumbles*.

Everyone in the square stopped what they were doing and cheered as they saw the *back of the pack*, followed by a police escort crossing the line. It had been a very long road, not just because 75 miles is a long way to ride a bicycle in 1 day. The finish line meant so much more than the end of the rally. Most importantly, they crossed it as a family.

Further reading

- https://www.redhat.com/architect/14-software-architecture-patterns

- *Nock, Clifton. Data Access Patterns: Database Interactions in Object-Oriented Applications. Boston: Addison-Wesley, 2004.*
- The companion website for this book: https://csharppatterns.dev

Appendix 1: A Brief Review of OOP Principles in C#

The landscape of programming languages contains hundreds of choices. If you're reading this book, you have probably come across C# at some point along your journey. For some, C# is the only language they've ever learned. For others, it's the third or fourth. Maybe you develop in C# every day and have for many years, or maybe you've just picked it up, and it's now your new best friend.

I realize that readers will be coming to this book with different levels of experience, different backgrounds, and different career objectives. I used to have a very popular video series published on LinkedIn Learning that was designed to teach C# to beginners. To be honest, this has always been my favorite audience. Teaching software development to newbies is like teaching magic in a jaded world sadly short of vision. I get to see *Aha!* moments nearly every week with my students at Southern Methodist University, where I teach at the Full Stack Code boot camp.

One of the things that excited me about the prospect of writing this book is this book helps you take the next step. Either you've just learned how to code or maybe you've been using the same magic tricks for many years with success, but you realize there's still more to

learn. Patterns are a great next step. Learning patterns makes you better at almost everything else in programming.

The problem here is that I don't know where you are in your journey. If you're a student of mine from SMU, you've learned JavaScript really well and you want to take the next step. C# is radically different from JavaScript. Jumping into this book from only knowing JavaScript would be hard, but not impossible. If you're self-taught, you've probably focused on what I call "survival skills." They include basic OOP plus how to work with a database and probably web technologies. If you're in a university, and you're learning how to code, your textbooks are likely not very illuminating. I know. I've taught at colleges and universities for 25 years. Usually, I write my own material because of the dearth of good books out there.

No matter where you are right now, or where you came from, this chapter is here to orient you. Originally, I had intended this to be the second chapter of this book. My editor wisely suggested we get into working with patterns as quickly as possible. She suggested an appendix, which we've mentioned throughout the book. You'll either get a good review or a crash course in C#.

In this appendix, you can expect to learn the following:

- A quick background on C# and how the language can be defined along a number of lines of taxonomy. That sounds fancy, but you'll see it's fairly basic.
- The syntax mechanics of C# with enough length and detail to get you through this book. Since this book mainly deals with plain old C# objects (POCOs) and a few common types from .NET Framework such as Lists, this section is possible.
- How to set up the projects in the book using three common IDEs in Windows. There are two project types we use: command-line projects and libraries.
- How to clone the sample code projects covered in the book.

Let's get started!

Technical requirements

This appendix is largely an overview of basic topics rather than a collection of projects. With that said, at the end of the appendix, you are guided in the use of the three most popular C# IDEs on the market today. You'll have a choice to make. Regardless of which IDE you choose, you'll need a computer running the Windows operating system. I'm using Windows 10. The projects used in this book probably work fine on a Mac or in Linux, but I didn't test them there, so your mileage might vary.

To follow the IDE tutorials at the end of this chapter, you'll need the following:

- One of these IDEs:
  - Visual Studio
  - **Visual Studio Code** (**VS Code**)
  - JetBrains Rider
- .NET Core 6 SDK

You can find the completed project files for this chapter on GitHub at https://github.com/Kpackt/Real-World-Implementation-of-C-Design-Patterns/tree/main/appendix-1.

A quick background of C#

Let me just say it out loud right off the bat: C# is a knock-off of Java. If you know Java, but not C#, you're going to have a very easy time. Now, pretend I didn't lead with that and allow me to slip on my corduroy sport coat. The one with the patches on the elbows. I have a tobacco pipe in the front pocket of the jacket sticking up so that you can see it. Naturally, in this day and age, nobody would dream of putting tobacco or anything else into it. That'd be wrong. But I need to look as much like a college professor as possible, so I can go all historical on you for a bit.

The C# language is Microsoft's flagship language product for corporate and game programming. It was designed by Anders Hjlsberg in the year 2000 AD. Some of my students claim I am ancient, so I figured I'd clarify by stating that it is, in fact, AD and not BC. The language was submitted and approved as a standardized language via the **European Computer Manufacturers Association** (**ECMA**). You might know this is as the same body that standardized JavaScript, which is really called ECMAScript. Many languages are standardized. This simply means that there is an open specification available for the language and that it is possible for others to create a competing implementation of a language based on the specification. C# is Microsoft's implementation. There is an open source competitor called Mono that used to be popular in several arenas, including cross-platform mobile and game development.

When Microsoft introduced C#, it also released .NET Framework and Visual Studio. .NET Framework is a massive set of libraries that provides a language support infrastructure for C# along with other Microsoft languages. Each language supported can compile its code into an intermediate format. The intermediate form at, called **Microsoft Intermediate Language** (**MSIL**), can then execute using the .NET runtime. This makes the overall language architecture very similar to Java, which compiles to an intermediate form called bytecode, which is then executed on a **Java Virtual Machine** (**JVM**).

The stated design goals for the language include the following:

- A simple, general-purpose, object-oriented language
- Support for a strong, statically typed variable system
- Automatic bounds checking on array types and the detection of uninitialized variables
- Automated garbage collection
- Source code portability; code should be executable in a variety of environments without changing the code significantly and without recompiling

Let's expand on these ideas.

C# is a general-purpose language

Pretty much everyone has heard of an invention created by Richard Clyburn in 1842. I present you with an adjustable wrench in *Figure A.1*:

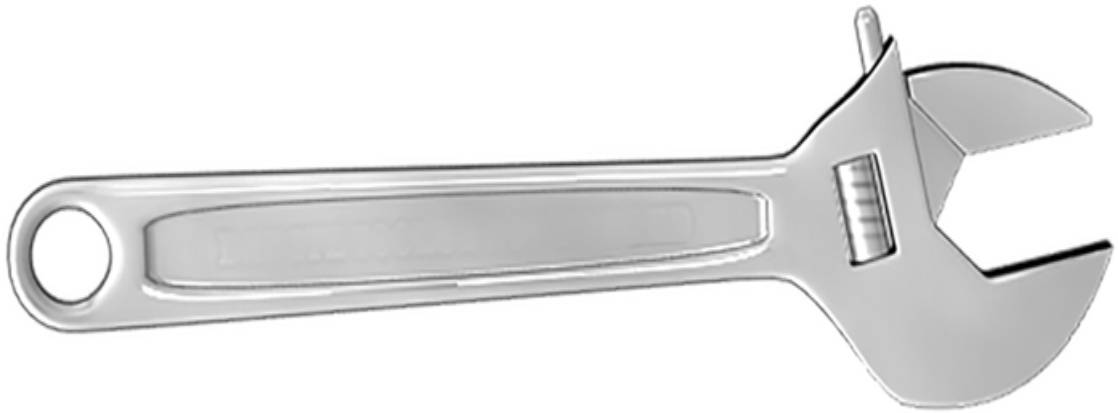


Figure A1.1: An adjustable wrench. You can use this for almost any wrenching job. If you're in a jam, you can also use it to open bottles and drive nails. It's a general-purpose tool just like the C# language, which can be used to make almost any kind of software

We call them *crescent wrenches* in the US, while many other countries refer to them as *spanners*. This wrench is a general-purpose wrench. It can be used for a variety of tasks befitting a wrench from the loosening and tightening of metal fasteners on your IKEA furniture to installing a new engine in your '57 Chevy, to installing a new garbage disposal for your kitchen sink.

Now consider the wrench in *Figure A1.2*:

Figure A1.2: This is a basin wrench. It's only good for one thing: tightening down fastening nuts on faucet hardware under a sink. It is a special-purpose tool just like SQL is a special-purpose language. Both can only be used for one purpose

You might have never encountered one of these before. This odd-looking contraption is called a *basin wrench*. It has only one job. It is used to tighten the faucet coupling when installing a new kitchen or bathroom sink. The wrench head pivots across 180 degrees, so you can maneuver it around all the pipes. The long handle helps you deal with the sink bowl that protrudes downward, blocking your access were you to attempt this with a general-purpose wrench.

Likewise, there exist a few special-purpose languages, the most popular being **Structured Query Language** (**SQL**). SQL is only used to query relational databases. You can't make a video game or an operating system with it.

On the other hand, C# is a general-purpose language. It can be used to make just about anything from line-of-business software to AAA video games. And yes, someone did once try to make an operating system with C# called *SharpOS,* showing us that it is possible to write an operating system with C#.

The biggest reason to choose C# as your main language of choice is the flexibility. Coupled with .NET Framework, you have thousands of building blocks. You can use them to build whatever software you might need.

C# is purely and fully object-oriented

There are two major paradigms that you'll see in programming languages: OOP and functional programming languages. Perhaps a third paradigm occurs when it is possible to mix the two within a single language. Let's cover the differences:

OOP	Functional programming
Code is organized into classes as the primary building block of your program.	Code is organized into distinct functions, which use their arguments in place of properties.
Methods often have side effects. They can change the state of the object from within the method.	Pure functions never have side effects.
In OOP, we have support for mutable and immutable objects.	Function programming never supports mutable variables. If you want to change something, you have to create a new variable.
Data is stored in properties within	The design goals are fo-

the object and is prioritized from a design perspective above methods that often serve only to mutate the state of the object.

cused on functions. They take immutable inputs and produce an output.

Figure A1.3: The difference between OOP and functional programming.

Some languages, such as C# and Java, are strictly object-oriented. Languages such as Haskell and F# are strictly functional, while languages such as JavaScript, Python, and PHP can support either or both paradigms. The original **Gang of Four** (**GoF**) software design patterns, which are the focus of this book, were built around OOP using a language called *SmallTalk*. Naturally, we'll keep our focus strictly on OOP.

C# uses a static, strong type system

There are three schools of thought with respect to type systems. First, let me tell you what I mean by a **type system**. Programming languages share a common purpose: they all take some sort of input and turn it into some sort of output. The input and output for the program are called **data**. There are three basic types of data as far as your computer is concerned.

Strings are alphanumeric data – think letters and numbers. If you can type it on your keyboard, it is alphanumeric.

The second type of data is **numbers**. I think you know what this means. While numbers can be alphanumeric strings, you can't use strings to do math. This distinction is at the heart of a strong type system, so remember this for just a moment.

The third basic type of data is **booleans**. Booleans are binary, meaning they can have a value of either true or false. Sometimes, the value is expressed as 0 (false) and 1 (true), or 0 (false) and not 0 (true).

The three basic types are called **primitive data types**. In a **strong type system**, you have to tell your program what kind of data you will be using when you declare a variable or create a function.

Beyond primitive types, C# supports the ability to create your own types as objects. Like primitives, these types are strong. They are also static. A **static type system** means that when you say you are going to use a variable to hold a string, you can never change the variable to any other type. It is static, meaning it can never change its type.

While C# and many other languages use a strong and static type of system, other languages do not. JavaScript took the opposite approach. JavaScript uses a weak and dynamic type of system. In such

a system, it is perfectly fine and normal to create a variable and assign a value of any type. Let's say we create a variable called *foo* and set it equal to "this is a string."

The very next line can assign a value of 9 to the variable of *foo*. The fact that you didn't have to declare a type with the variable indicates a **weak type system**. The fact you can change the type any time you'd like indicates a dynamic type system. JavaScript is incredibly powerful and flexible, but it also feels very foreign for developers who learn a strong static system leading to the idea that JavaScript is a "toy language" and not for serious work. This is, of course, wholly false.

The third type of system that you'll encounter is called duck typing. Likely the most famous system using duck typing is Python. In Python, you see the middle ground between strong static typing in C# and the weak dynamic system in JavaScript.

In Python, you don't declare a type with your variables. However, the Python interpreter observes how the variable is used, and it infers a strong type. This kind of system is called duck typing because the designers who created this type of system claimed "If it walks like a duck and quacks like a duck, it's a duck."

Later versions of C# support also duck typing, so we'll see some examples later.

C# has automatic bounds checking and detection for uninitialized variables

C# was created to correct deficiencies in other languages. You hear that for pretty much every new language created in the last 20 years. Some group hates the way something works in C or Java, so they make a new language. Usually, they are an amalgamation of the so-called good parts of other languages. If you don't believe me, read up on Apple's *Swift* language, the *Rust* language, or the *Golang* languages created by Google. C# isn't any different.

Studies have shown that the lack of bounds checking and uninitialized variables are both major sources of software bugs. Really, then, the objective here is to create a language that makes it harder for developers to make mistakes. In the next few features, we'll discuss embodying this goal.

Bounds checking refers to arrays. An array is a special type that allows you to store multiple values in a single variable. Since C# is strongly typed, array elements must all be of the same type. You can have an array of strings, an array of numbers, or an array of objects. Furthermore, in C# when you define an array, you must tell your program how many elements you will put into the array. Once you set it, the size is fixed and you can't change it without jumping through some proverbial hoops.

Uninitialized variables are not just representative of sloppy work – in some programming languages they can be dangerous. The C programming language comes to mind here. C went through some drastic changes over the last 10 years, but back when I was your age, the mantra of C was “With great power comes great responsibility.” I think that’s also Spiderman’s mantra. Maybe Peter Parker should have been a coder instead of a photographer.

In C, as in most languages, a variable is just a pointer to an address in memory. Computer memory is a lot like an apartment complex. In fact, older versions of C# and Visual Basic called their memory models “apartment threading.” Apartments are containers for you and your stuff. Some apartments are small and economical (unless you live in New York City, in which case they are simply small). Others are larger and more expensive. Regardless of their size, shape, and decor, their function is to hold you and your stuff.

Each apartment in an apartment complex has an apartment address number. So, you can effectively say *I want to put my stuff in apartment 122*. If you get a raise at work, and you decide to upgrade, you can move your stuff into apartment 300, which is larger. The address tells us where your stuff is located, and the manager of your apartment building knows how big each apartment is and how much it costs.

Computer memory works the same way. There are registers of addresses (such as the apartment complex) that can be told to store stuff of varying sizes. We can extend our analogy to say the apartment manager is the computer running your program. It knows where the containers are since it keeps a list of each apartment's address and how much space each apartment occupies.

The addresses are not simple numbers such as 122 or 300. They are a bit more complicated and are not easy to work with. So, instead of dealing with addresses, we can give our apartments, or containers, a straightforward human-sounding name that describes how your variable is being used in your program.

When you create a container, such as an apartment, of a defined size (small and cheap versus big and expensive), it has an address. But instead of using that, you use whatever name you think is easy to use and remember.

Now imagine an apartment that was built as part of the building, but it is full of construction debris and trash left over from when it was built. Nobody has ever rented the apartment, so nobody has ever cleaned up the mess. This is an uninitialized variable. If someone wanted to rent the apartment and move in on the same day, that is, access the address, they would find it's full of trash. When a running program with a strong static type system encounters trash that it doesn't ex-

pect, the program crashes. If you have ever encountered a **blue screen of death** (**BSOD**) in Windows, you know what this looks like. Uninitialized variables will have whatever junk was left over from the last time that memory address was used. Back in the day, a major headache of working with C was manually allocating and deallocating memory. If you did it wrong, things crashed in the least graceful way possible. What the C developers really need is some way to handle garbage collection.

C# supports automated garbage collection

The nightmare scenario we just covered shows the need for language improvements. C# is meant to be simple and useful. In C, back when C# was conceived, we were manually allocating memory addresses. This meant executing a statement that defined your allocation using a hexadecimal value. Most humans are doing well working in base 10. Our old environments had us mentally converting decimal (base 10) numbers into base 8 (octal) and base 16 (hexadecimal) to do our memory allocations. Then, we used that memory, and when we were done, we needed to remember to deallocate the memory. In our apartment metaphor, this would entail moving out of the apartment and giving it thorough cleaning in the process. The apartment should be move-in ready when we deallocate.

What if we had a language that handled allocation and deallocation automatically? We do. C# has a garbage collection system that handles everything for you. You create a variable, and the details are taken care of by the .NET runtime. When your variable goes out of scope, it gets marked for cleanup and, ultimately, the garbage collection process in the runtime deallocates and cleans up the memory for you. As you can imagine, the manual allocation and deallocation of memory in a system that had no safety net was also a major source of bugs.

C# code is highly portable

The C# language specification aims at having portable code. In the earliest versions of C# and .NET, we weren't very portable. Windows was all we had. You couldn't run C# code on a Mac, Linux, or a phone. Gaming consoles were also out.

The open source community created Mono, an open source version of C# designed to run on Linux operating systems, but it has always been several years behind Microsoft's implementation in C#. Eventually, Mono became a very popular way for C# developers to leverage their language skills to create mobile applications for the Android and iPhone platforms using Mono alongside a framework called Xamarin.

A game development engine called Unity 3D also leveraged Mono and brought a AAA quality game engine to the mass market. Up until then, game engines such as Unreal were strictly in-house. The only way to develop a quality game was to work at a company that could afford the very expensive SDKs. Unity 3D busted the game development industry wide open, and they used C#/Mono to do it.

As time moved on, Microsoft made each iteration of the language more portable. We saw a big shift when Microsoft announced its Azure cloud platform. They wisely understood most IT professionals were never going to use a system that limited them to Microsoft products. Linux owns the majority share of web servers. When cloud computing started to reach critical mass in terms of popularity, we saw a huge proliferation of open source compatibility introduced into Microsoft's landscapes including .NET Core, which allows us to compile and run C#, or any .NET language, to run on nearly any type of hardware environment making your code truly portable. The portability comes from the .NET runtime. Let me explain what that means.

There are a few different ways any given programming language is able to execute code. First, there is code that can be compiled into native machine code. For this, you need a language such as C, C,++, or Rust. Programs that can be compiled this way run on "the bare metal," making them very fast in terms of execution, but the languages required for this are typically more demanding to use. C and

C++ require you to keep tabs on your memory utilization, which is the primary source of bugs written in these tools. Rust aims to eliminate memory errors with very strict compile time restrictions. Using these languages can be frustrating, and your time to market is usually slower, especially when adopting these languages for the first time. There's a trade-off between developer productivity and the performance of the software when it runs for the users.

The second camp is languages that use interpreters. Usually, these languages are scripting languages such as Python, PERL, and Lua. These languages are built around developer productivity and, typically, sacrifice execution speed and efficiency.

The third group is sort of in-between. All .NET languages, in a similar fashion to Java, compile to an intermediary binary format. Java calls this byte code. C# and other .NET languages call it **Common Intermediate Language** (**CIL**). These binary formats execute faster and more efficiently than an interpreted language, but they do require a runtime to execute the intermediate form.

It is important to remember how the published version of code in any given language might perform. On the whole, compiled C# code is generally slower than the "bare metal" compiled languages, but not by much.

Language mechanics in C#

We've mentioned already that C# is strictly an **object-oriented programming** (**OOP**) language. OOP languages support a set of abstract features, which you need to understand before we can continue. Firstly, OOP languages organize code differently than procedural languages. In OOP, we model elements from the real-world using code. The real-world elements are described using **properties**, **methods**, and **events**.

For example, if I were modeling a circle, I might give it a set of properties as seen in Figure A1.3:

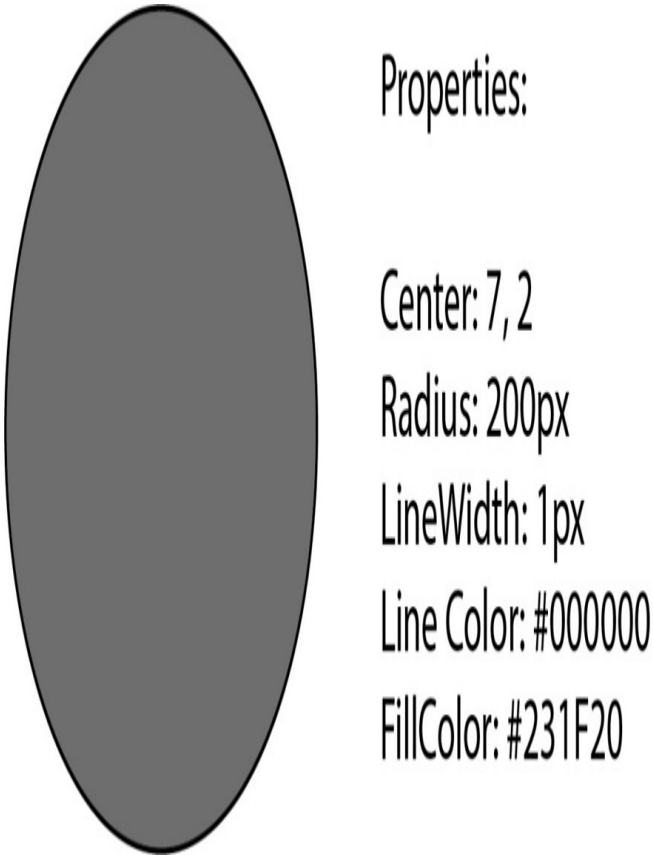


Figure A1.4: A circle, like anything else, can be represented by describing it as a set of properties

Using these properties, I can describe any circle I might want to draw on the screen in my program.

Effectively, properties are variables that belong to the circle. They contain the data that describes the circle. When describing a variable that belongs to an object, you refer to these as *properties*, *member variables*, or *members*.

Additionally, objects describe the actions they might perform. When we talk about functions in the context of OOP, we call them methods.

We could say our circle object has a method called **`draw()`** that draws the circle to the screen given the properties that describe it. We could also add a method to resize the circle. Methods are the same thing as functions, so we can pass in arguments. Our resize method will need to know the new radius of the circle. It might be called this way: **`resize(100)`**. This would change the radius of the circle from 200 to 100.

Objects also commonly have a specialized set of methods designed to respond to events. Events can be described as *things that happen* while your program is running. Common examples might include the user clicking on the circle, hovering a cursor over the circle, right-clicking on the circle with your mouse, long-tapping the circle on a touch screen, or the passage of time. Maybe you want the circle to disappear after 10 seconds.

It isn't necessary for an object to have all of these pieces. Some objects only represent data using properties. Some are just used for constants and methods, such as the **`Math`** class in C#. However, most of the time, there is a mixture.

In addition to objects being organized with properties and methods, there are a few other buzzwords to learn:

- Encapsulation
- Composition

- Inheritance
- Polymorphism
- Open recursion

All of this sounds horribly complicated, but don't worry, that's what I'm here for. We'll touch on all of these concepts within this chapter.

OOP first surfaced in the 1980s with the Ada programming language. Ada was named for programming pioneer August a Ada King, Countess of Lovelace. You've probably heard of her as Ada Lovelace. The language that bears her name was created under contract for the US **Department of Defense** (**DoD**) and was designed to replace over 450 disparate programming languages that were in use at the DoD at the time. As you can imagine, it had some big shoes to fill. The Ada language is to programming what the Tucker automobile was to the American automotive industry. The 1948 Tucker automobile was the first to offer standard seat belts, a water-cooled aluminum engine, disk brakes, fuel injection, and an independent suspension. These are all standard features of modern cars, and we'd be hard-pressed to consider buying a car today without them. However, they didn't exist on cars designed and built in the 1940s, or certainly not all on one car.

The original Ada language supported object orientation, design by contract, strong typing, an explicit syntax for concurrency, tasks, mes-

sage passing, encapsulation using private and protected classes, and code safety enforced by a compiler. Sound familiar? Many modern languages, including C#, include these features out of the box owing largely to the virtues embodied in Ada.

OOP became popular because it allows you to organize your software in a way that makes sense and is easy to comprehend. Developers who learn OOP first have trouble switching away from it. Having worked with both kinds of thinking for many years, I believe OOP is the best way to create large-scale software projects because it forces you to think a certain way. The main tenets of OOP, some of which are covered in SOLID principles, are like a built-in way to make your software maintainable, testable, and extensible in the safest way possible. I believe it is important to use an object-oriented language when you initially learn patterns. The original work on patterns was based on Java, which is remarkably similar to C#. Many of the ideas presented in this book won't work or must be shoehorned into working with non-object-oriented, multi-paradigm, or dynamic languages. For example, JavaScript ES5 is considered object-oriented, but objects are dynamic. By this, I mean you can change the structure of any object, including those internally supported by the language at any time while your code is running. You can even alter all instances of an object owing to JavaScript's use of prototype inheritance. JavaScript is weakly typed, and the list of types it supports is limited. Classes and encapsulation simply don't exist in ES5. It is difficult to

study patterns in a language such as this because patterns require a foundational set of restrictive rules, which don't fully exist in many other languages.

By choosing to work with C#, you align yourself with all the key tenets of developing software with well-known, battle-tested patterns.

Variables in C#

I consider the simplest definition of a variable to be as follows:

A variable is a named container that holds data in computer memory.

When you learn a programming language, you need to know whether the language is strongly or weakly typed.

In a **strongly typed** language, also referred to as a statically typed language, you must define your variables' data type. You have to tell your program what kind of data you are storing in any given variable.

In a **weakly typed language**, you don't need to declare a type because, effectively, there is only one type: everything is an object.

Here is a great way to think of it. Data can have a shape, as can a container. Containers also have a definite size. You cannot fit one pound of sugar into a container designed to hold no more than a

quarter of a pound. However, you can fit that same pound of sugar into a 5-gallon bucket with room to spare.

C# is statically typed with support for **implicit typing**, also known as **duck typing**. In a strongly typed language such as C#, you would see something like this:

```
int myNumber = 5;
```

Here, we're creating a variable called `myNumber`, of type `int`, and setting it to an initial value of `5`.

Implicit typing allows us to make a small change:

```
var myNumber = 5;
```

Instead of the type declaration appearing before the name, you use a one-size-fits-all keyword, `var`. The compiler will see these statements as identical. The compiler can figure out what type your variable is supposed to be based on its initial assignment. Here, we see an initial assignment of 5, so the compiler will assume you meant this to be an integer (`int`). If that's not what you wanted, you can give the compiler a hint. Let's say we really wanted `myNumber` to be the `decimal` type. Since there is no mathematical difference between 5 and 5.0, the compiler gets it wrong and, eventually, you get red wavy lines in your

code indicating you have a problem. How can you specify the type and still use the **`var`** keyword syntax? Behold:

```
var myNumber = 5d;
```

We add a *d* as a suffix to the number. Just like that, the compiler now knows you want this to be a decimal. But I'm getting ahead of myself.

Signed numeric types delineated by memory size usage

C# supports signed and unsigned numeric types. Numbers in programming languages are handled in memory using a mathematical operation on binary numbers called two's complement as seen in Figure A1.4:

Figure A1.5: Integer representation on computers happens using a mathematical concept called two's-complement. One year, I went to the office Halloween party as an integer. I had this figure printed on a t-shirt. The best costumes are the ones you have to explain

If you're a math geek, check out the Wikipedia page for this topic provided in the further reading section.

For the rest of us, this concept can be explained simply. Any variable that's meant to store a number will have a minimum and maximum size for the number. For example, a common 32-bit integer (**`int32`**) has a range of -2,147,483,648 at the minimum end and a maximum value of 2,147,483,647. We had to chop one off the maximum to account for zero. The size ranges are dictated by two's complement given the amount of memory space used. The range I just showed you represents the capacity for a **signed** type, meaning it supports numbers less than zero. If you don't care about negative values, C# allows you to use **unsigned** types that extend the range from 0 to 4,294,967,295.

C# allows you to control memory usage by selecting from a myriad of signed and unsigned ranges. In other words, signed numeric types are delineated by memory size usage. I think this is a point that most developers forget or ignore since most of us, when we need an integer, just use **`int`**. I used an analogy earlier:

You cannot fit one pound of sugar into a container designed to hold no more than a quarter of a pound. You can, however, fit that same pound of sugar into a 5-gallon bucket with room to spare.

There is a lot of wasted space in the 5-gallon bucket. You can control the waste by selecting a type that has a range that is compatible with

how it's used. Think about how many times have you either seen or done this:

```
int myAge = 54;
```

That's a huge waste of space. First, a person's age can't be a negative number. It makes sense to change this to the following:

```
uint myAge = 54;
```

Here, we're using the unsigned version of the 32-bit integer. See if you can pick a more appropriate type from the following table:

Type	Description	Minimum	Maximum	Bits
`bool`	Boolean	False (0)	True (1)	1
`byte`	Unsigned byte	0	255	8
`sbyte`	Signed byte	-128	127	8
`short`	Signed short integer	-32,768	32,767	16
`ushort`	Unsigned	0	65,535	16

	short integer			
int	Signed integer	-2,147,483,648	2,147,483,647	32
uint	Unsigned integer	0	4,294,967,295	32
long	Signed long integer	-9e18	9e18	64
ulong	Unsigned long integer	0	1.8e19	64

Figure A1.6: A list of C# integer types with ranges and how much memory each consumes.

As you can see, as we move down the chart, we use more and more memory. A bit is the smallest thing a computer can work with. A 64-bit integer can only be understood using the term *±9 kajillion* or *18 kajillion* depending on whether it's signed.

Why would we use a 32-bit unsigned integer with a maximum value of 8 billion for a person's age? Either we don't know about numeric type

delineations or we're simply lazy, which is one of those debilitating forces we talked about in *Chapter 1*. We've just broken a window that leads us into a big ball of mud. We should be using an unsigned byte that has a minimum value of 0 and a maximum value of 255. Most normal humans, aside from Elvis, who is living on a secret island in Hawaii, and Chuck Norris, who is obviously immortal, don't live past 100 years. When you consider the earth is about 4.5 billion years old, you begin to see the folly of this common practice. If you do this, you're in good company. Microsoft does it too throughout its C# documentation. I am tempted to quip writers make bad coders, but I'd be shooting myself in the foot if I did. I won't fault you if you don't immediately change all your production code from **`int`** to **`byte`** so long as you now realize you're wasting a lot of memory.

Figure A1.7 shows us all the valid integer types. There are similar names and ranges for floating-point numbers and text types. All of these types are considered **primitives**. Primitives are what you think they are based on the name. In many C-based languages, these are usually simple structures. C# doesn't treat them that way. C# and, by extension, .NET Framework treat them as objects. The keywords in *Table 1* are primitive aliases to the real objects from .NET Framework. The objects have different names from the list of keywords presented in *Figure A1.6*. The example you've most likely encountered is the **`int`** keyword mapping to the **`Int32`** class.

The real list turns out to be something like the list in *Figure A1.7*:

Alias/primitive keyword	.NET type
`bool`	System.Boolean
`byte`	System.Byte
`sbyte`	System.Sbyte
`short`	System.Int16
`ushort`	System.Uint16
`int`	System.Int32
`uint`	System.Uint32
`long`	System.Int64
`ulong`	System.UInt64

Figure A1.7: The full list of C# integer ranges with the alias in the first column and the real implementation class in the sec-

ond column.

C# allows for aliases, and often the alias points to the same class name expressed with an uppercase letter. All the primitive types are in the .NET **`System`** object. Almost every project you create in Visual Studio has a **`using System`** statement at the top. This is why you can see strings being created in several different ways.

A string defined using all lowercase characters works because of the alias:

```
var string foo = "bar";
```

A string defined with a capitalized **`String`** keyword works because of the **`using System`** statement that is likely at the top of your project:

```
var String bar = "baz";
```

Of course, you can also write the following:

```
var System.String fooBarBaz = "foo bar baz";
```

Honestly, I've never seen anybody do this. Years ago, when I taught introductory programming courses in C# at a nearby college, the difference between **`String`** and **`string`** was a widely asked question, especially coming from students who had previously studied Java. Java lacked these kinds of aliases.

Separate character and string classes with different quotation marks used for each

This point is short, but worth mentioning if you are coming from another language, or if you switch languages frequently. C# has separate class types for strings and characters. The string type is `System.String`. It's an alias that is used far more frequently, which is just `string`. The character type, which is a single character string, is created using the `System.Char` class or just `char`. The quotation marks used in the assignment of values matter, too. Many languages, such as JavaScript and Python, allow you to use single and double quotes interchangeably. In C#, strings are denoted with double quotes, while characters are assigned using single quotes. They are not interchangeable.

Collections

JavaScript has `Arrays` (a.k.a. what I'm hoping I get at work for writing a book). Python has `Lists`. Java has `ArrayLists`. C# has a broad set of **Collections**. C# has arrays such as JavaScript, but they are more limited. You have to set the number of items you are putting into the array in advance, and once set, that number can't be easily changed. A more useful form of array in C# is the .NET List class.

List classes are like arrays, except you can add and remove items any time you need to. They are a lot more flexible. Lists are strongly typed, like everything else in C#. This means all the items in the List have to be the same type. C# has a system for this called **Generics**. I won't get into the details of generics here, as it would take more time and room than we have. Just know that generics are denoted in C# with angle brackets (**<>**). In the book, you'll see references to **`List<>`**. This indicates a generic list, and once again, that just means you can put anything into the list as long as everything you put into it is of the same type. If you wanted to create a list of strings, it might look like this:

```
var myStrings = new List<string>();
myStrings.add("foo");
myStrings.add("bar");
```

To use these generic collections, you have to add a statement at the top of your class file like this:

```
using System.Collections.Generic;
```

Generic collections are some of the most versatile and widely used classes in .NET Framework. You will definitely encounter them early and often.

Classes

So far, my assumptions regarding your exposure to OOP have presupposed you know what a class is and its purpose. Just in case that's an invalid assumption, let's clear it up now. A *class* can be thought of as a blueprint for a house. The blueprint describes everything you need to know to build the house. You can use the blueprint to make as many houses as you need, and as you build them, you can change the individual properties on each house. It isn't necessary that they all be the same size or color, or that they all have a certain number of windows. You can alter any of these properties when you build the house.

In most OOP languages, the class is the main form for defining how objects are constructed. There are even special methods called constructors, which run when an object is instantiated, which is to say when you create an instance of an object using the new keyword.

Here is the example of the **`Person`** class I used earlier. This time, I added all the parts so that I can show them to you:

```
using System;
namespace MyProject;
class Person
{
```

```
    public string Name { get; set; }
    public byte Age { get; set; }
    public Person()
    {
      Name = string.Empty;
      Age = 0;
    }
    public Person(string name){
    {
      Name = name;
      Age = 0;
    }
    public Person(string name, int age)
    {
      Name = name;
      Age = age;
    }
    public override string ToString()
    {
      return "Person: " + Name + " " + Age;
    }
  }
```

A class has a few main parts:

- Using statements to define dependencies
- The namespace
- The class name

- Constructors
- Properties
- Methods

Let's go over each part.

Using statements

The **`using`** keyword has a double meaning in C#. When you see it at the top of a class, it refers to the class' dependencies. This is common in most languages. In Java and Python, the word is **`import`**. In JavaScript, it's either **`require`** or **`import`** depending on which convention you use. In C#, the keyword is **`using`**. The statement signals to the compiler that some of the classes we're going to reference in this class exist in other parts of our project, or even outside our project, which is the case when we use third-party libraries.

For example, in order to use the **`Console.WriteLine`** method, which prints text to the console, first, we must declare that we will be using the **`System`** namespace. We do this with the following statement:

```
using System;
```

The **`Console`** object is in the **`System`** namespace. The **`using`** statement tells the compiler we'll be using classes from that namespace. That

last statement makes for a natural segue into the next topic.

Namespaces

Namespaces are a way of organizing your code in C#. If you're coming from Java, you've used packages. In Python, they're called modules. In JavaScript, you export objects you want to expose in modular code. In C#, a namespace appears at the top of your class files. Usually, it corresponds to the name of your project. If you create folders inside your project to break up your code, usually, the namespace mirrors the folder structure.

In the preceding sample code, we have a namespace called `MyProject`, which would be the name of the C# project. If I made a folder called helpers to hold some helper objects, the namespace in those folders would be `MyProject.helpers`.

Namespaces are not an absolute requirement. Most IDEs put them in automatically, so you'll usually see them. Five years ago, when I last worked with the Unity 3D game engine, the IDE didn't put namespaces in the class code automatically. It's possible to work without namespaces, but it's rare. Visual Studio and Rider add the namespace automatically when you create the class. In VS Code, everything is manual, so you might have to add it yourself.

The class name

The line that names the class is not difficult to spot:

```
class Person
```

You just use the keyword class followed by the name of the class. By convention, class names use the Pascal case convention, which means each word in the class name is represented by a capital letter. If we wanted to create a class to represent a floor manager at a retail outlet, we would call it **`FloorManager`**.

Classes, like properties and methods, can use access modifiers to define how they can be accessed. We'll talk more about this later.

Constructors and instantiation

A constructor is a special method designed to run when you **instantiate** the class. Semantically, a class is not an object. A class is a blueprint used to create objects. Simply put, class is your code. When your code runs, it creates instances of objects through a process called *instantiation*: **`Create an instance`** = **`instantiation`**. See? It's a fancy word every software developer needs to understand. Instantiation happens when you put the **`new`** keyword in front of a constructor method. This is why there is always a parenthesis at the end of your **`new`** statement. The constructor returns the instance of the object set

up per any code in the constructor. If you don't supply a constructor, the .NET compiler adds an empty one when you compile.

There are defining rules that govern the constructors. First, the constructor's method name must match the name of the class. Second, you can't define a return type since it is fixed to returning an instance of the class it is constructing.

C# allows you to have multiple overloaded constructors. This means you can have more than one constructor as long as the type and number of arguments are different. Our code example from earlier features three constructors:

```
public Person()
  {
    Name = string.Empty;
    Age = 0;
  }
  public Person(string name){
  {
    Name = name;
    Age = 0;
  }
  public Person(string name, int age)
  {
    Name = name;
```

```
        Age = age;
    }
```

Note the following:

- All of the constructors are named **`Person`** because that is the name of the class. They must match the name of the class or they aren't valid constructors.
- There is no return type specified. The return type would be between the public keyword and the name of the constructor method.
- Even though all three functions have the same name because they have different numbers of arguments, it's legal. The first constructor has zero arguments, the second has one, and the third has two. This is called *method overloading* and is a form of polymorphism.

Constructors are a very useful way to establish the initial state of your object the instant the object is created. I often say that the number one job of a software developer is to make sure no object within your program is capable of entering an invalid state.

In the preceding example, the constructor is syntactic overkill. C# will initialize the age property to 0 automatically. I initialize strings to empty strings because I don't like the ambiguity of dealing with null values. Sometimes, your language of choice, in this case, C#, will try to handle things such as initialization for you. Relying on automatic language features will lead to mental laziness, one of the destructive

forces I talk about in *Chapter 1* that leads to chaotic, unmaintainable code. If you don't believe me, go to YouTube and find videos on JavaScript's automatic semicolon insertion "feature." You need to remember that your code is a set of instructions to be executed by one of the dumbest things on planet earth. Computers take everything literally, and if your instructions contain any ambiguity, bad comedy always ensues. Granted, some people call that job security, and everybody needs good war stories to tell. It's like that one time someone on the team pasted an incomplete SQL update statement into SSMS pointed to a production database and updated all the customer records to one account. A system with a million users a day was down for 7 hours. To be clear, no, it wasn't me. If it were, I would never speak of it. I'm making a point. That painful experience happened because someone was too lazy to do their job the right way. Personally, I'd rather do things right and keep my weekends free. When I write code, I am always explicit and intentional. It keeps me out of trouble. Always pay attention to your object's state and write your code to guard against invalid states. Your constructors are the first line of defense. The second line of defense is encapsulation, which we'll talk about shortly.

Properties

Objects consist of state variables and functions capable of doing work on those state variables. This is a fancy way of saying an object

is like a marriage between two common programming concepts: variables and functions. Classes are designed to represent real-world objects. Any object in the real world is defined by its descriptive properties. What color is it? Is it bigger than a breadbox? Physical objects are further defined by what they can do. A car can go, a dog can bark, and so on.

For now, let's focus on the objects' descriptors. It's easy to call them properties. A basketball has properties such as its circumference, its orange color, its black lines, and how much air it is holding. These are properties of the basketball. In C# classes, properties look like this:

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

The word **`public`** is an access modifier. We described this earlier in the appendix.

Every property has a name. This property's name is **`Name`**, and it is defined as a string. The part that says **`{ get; set; }`** makes it an auto-implemented property, which is an easy form of encapsulation that I'll explain later.

You access a property on an instantiated object using dot notation. For example, to set the **`Name`** property on a **`Person`** object, you instantiate and then set it like this:

```
var somebody = new Person();
somebody.Name = "Bruce";
```

Access modifiers define the availability of properties, methods, and even classes in C#.

Methods

Methods are functions attached to objects. Since C# is strictly object-oriented, we should probably always refer to them as methods rather than functions, but the words are effectively synonymous. Even the strongest adherent to OOP will occasionally slip and say *function*. With that out of the way, there are a few things to remember with respect to methods in C#. First, they follow the same rules as variables in that they are strongly typed. They need any arguments passed in to be typed as well as the return value to be typed. Like variables, the method names need to be unique within scope.

C# uses C-like method signatures common to all C-based languages. A method signature consists of the parts of the method that define it as unique:

- The name of the function
- The number and types of arguments
- The type of the return value

In the preceding example, we have this method:

```
public override string ToString()
{
  return "Person: " + Name + " " + Age;
}
```

In this method signature, we can see this method is public. It can be accessed by any class in our program. We can see that its name is **`ToString`** and that it takes no arguments. Furthermore, we can see this method is expected to return a string.

This one has an extra keyword stuck between the access modifier and the return type. It says **`override`**. The **`ToString()`** method is on every object in C# because that method is in the Object base class. It is always inherited from the base class in every object you make. The base class implementation isn't very useful. It is very common to override the function with something more useful. That is what's happening in the preceding example. We've changed the implementation in the base class by overriding the implementation with our own. We do this in plenty of places within the projects covered in this book.

Encapsulation

This concept entails one of the most important aspects of OOP: the maintenance of the object's state.

Earlier, in *Figure A1.3*, I presented a circle with a set of properties with values that describe the circle you're seeing. As your program runs, it's likely those properties will change in response to events within that program. If you take a snapshot of that circle at any point in time during your program's execution, you can talk about its current state. It's currently sporting a radius of 200 pixels, with a line color of black, and fill color of dark gray. Let's create some code to represent our circle in the simplest way possible, without any encapsulation, so we can see it added later:

```
public class Circle
{
  ushort centerX;
  ushort centerY;
  ushort radius;
  byte lineWidth;
  string lineColor;
  string fillColor;
}
```

In this listing, we create a **`Circle`** class, and within it, we define the fields based on our earlier discussion. Note that I did not say proper-

ties. Properties and fields are different in C#. Fields are simply member variables. These fields are wide open. Effectively, any other class can directly modify the fields at any time. Why is this bad? Remember, it's all about guarding the state. It really doesn't make sense to have a negative radius, which is why I used an unsigned short. Likewise, graphics coordinate systems on a computer are usually strictly positive, with $0,0$ located in the upper-left corner of the screen. To that end, I used unsigned shorts here because we don't need values beneath zero, and the upper value seems reasonable. So far, I've done everything right, except I haven't guarded the state beyond limiting invalid values based on types.

Let's imagine that you're working on a program such as *Adobe Illustrator* or the open source program *Inkscape*. These programs deal with vector graphics, and they most definitely need a structure to handle a circle. Both programs are pretty big in terms of lines of code and the number of classes. I realize neither program is written in C#, but if they were, imagine all the different objects each program would have flying around at any given moment. If any one of them has the ability to, at any time, modify the properties of our circle object, finding and debugging odd behavior changes in the circle becomes nigh impossible. Usually, lack of encapsulation is caused by the cardinal sin of sloth, a lack of imagination, or both.

Our program could change any of these properties at any time. Encapsulation aims to prevent you from entering your object into an invalid state. For example, what if I try to specify a negative number for the radius? That doesn't really make sense.

To protect our object from entering an invalid state, we can hide the properties behind access modifiers. Access modifiers are keywords that define "who or what" is allowed to make changes to an object's state. In an ideal program, the object should always solely be in charge of its own state. A different object in the program shouldn't be directly manipulating the radius. Aside from access problems, there's nothing policing the type of data being passed in, other than our strongly typed variable system. To improve things, let's start with the radius. Earlier, I mentioned that I thought **`ushort`** was an appropriate type because being unsigned, it doesn't support negative values. Further, I think the next lowest unsigned type, byte, which tops out at 256, is too small. I can easily see needing the ability to make a circle larger than 256 pixels in diameter. But 65K is probably too big. Let's say I want to limit the maximum radius to 1,000 pixels. This is probably smart because, in a large graphical composition, we might have thousands of circles and we should be thinking about our memory consumption. We can hide the radius field behind a special function that checks to see whether the radius you are setting is valid, in this case, a number between 0 and 1,000.

There are two steps at play here. First, you have to hide the radius with an access modifier. The most common access modifiers are listed as follows:

- **Public**: Anything can access the property or method.
- **Private**: Only this object can access the property or method.
- **Protected**: Only this object and its descendants can access the property or method. More on this after we talk about inheritance.

You'll find these available in most object-oriented languages. C# has a few additional access modifiers:

- **Internal**: Anything within the same assembly can access the property or method.
- **Protected internal**: The property or method can only be accessed by code within the assembly wherein it was declared or from a subclass of the class where it was created.
- **Private protected**: The property or method can be accessed by subclasses that are declared within the same assembly.

You can visualize this using the following table:

Caller's Location	public	protected internal	protected	internal	private protected	private
Inside the class	☑	☑	☑	☑	☑	☑
Subclass (same assembly)	☑	☑	☑	☑	☑	⊗
Different Class (same assembly)	☑	☑	⊗	☑	⊗	⊗
Subclass (different assembly)	☑	☑	☑	⊗	⊗	⊗
Different Class (different assembly)	⊗	⊗	⊗	⊗	⊗	⊗

Figure A1.8: Relative access levels by the caller's location.

Let's go back to improving the **`Circle`** class. The first improvement we'll make is adding a private access modifier to the radius field. I'll also move **`radius`** to the top just so that it is easier to see for this discussion:

```
public class Circle
{
  private ushort radius;
  ushort centerX;
  ushort centerY;
  byte lineWidth;
  string lineColor;
  string fillColor;
}
```

The private access modifier tells us that only methods in “this particular object” are allowed to change the radius. That’s perfect, except now the radius field is totally hidden. There isn’t any way to set it from the outside, and we’ll, of course, need to do that at some point. Let’s change the access modifier to something more permissive:

```
public class Circle
{
  private ushort radius;
  ushort centerX;
  ushort centerY;
  byte lineWidth;
  string lineColor;
  string fillColor;
}
```

OK. Now we’re back to where we started. The public access modifier means that anything can alter the radius. That’s not good! Dagnabbit!

We need private back, but we need it to be a little better. We need accessor methods to allow us to get and set the field’s value from outside this object, but in a manner we control. When I do this, I use a common convention of renaming the field so that it starts with an underscore. This is done so that we can easily remember it’s a private backing field:

```
private ushort _radius;
```

Next, let's add an accessor method that allows us to get the radius. The **get** method is commonly called a *getter*. We don't need any restrictions on this one:

```
public ushort getRadius()
{
 return _radius;
}
```

Next, let's deal with the real problem at hand, which is controlling how the radius is set. Another accessor method can be used:

```
public void setRadius(int newRadius)
{
 if(newRadius >= 0 && newRadius <= 1000)
 {
   _radius = newRadius;
 }
 else
 {
   throw new Exception("The radius must be betwe
   and 1000.");
 }
```

```
    }
  }
```

Now we have a **`set`** method, commonly called a *setter*, that can set the value. Note that its access modifier is public, which we just learned means anything can access it. This setter method contains code that checks to see whether the new radius conforms to the required range. If it does, it goes ahead and changes the radius to the new value. The object is now in command of its own state. You can't just randomly set the value to an invalid number. You have to *ask* the object to change its radius value, and it doesn't automatically say *OK* – it checks to make sure you didn't pass something that doesn't make sense.

Additionally, accessor methods can be used to define a read-only property. By simply neglecting to create a public setter method, you limit the ability to set the value of the private field to other methods within this class.

C# auto-implemented properties

If you're a C# veteran, you're no doubt yelling at this book right now. Don't do that. First, it makes you look crazy unless you have your earbuds in, then at least people around you will think you're just yelling at your mother-in-law. Second, this book is very sensitive. You might

hurt its feelings. I get it. The earlier example is definitely not how we usually make accessor methods in C# – at least not these days. Back when I was your age, gas was 35 cents per gallon, and that's exactly how we did it. I like showing it to you that way because I think just throwing **`int foo { get; set; }`** at someone without explaining how it works is kind of mean, which is to say that's how the average C# book might cover it. As we all know, *mean* is the average. Was that a math pun? Yes, I think it was.

Modern C# has a different syntax for defining properties predicated on the idea that most properties are really fields. But if you leave them as fields, your OOP professor will give you a B because you didn't encapsulate them. We can't have that. I could climb back up on my soapbox and decry their existence, but I have a page count I need to stay under, so let's just see how they work and you can decide whether they are a good idea or not.

If you want to say your class is fully encapsulated, but you don't need any logic to control state, your code winds up with a lot of boilerplate code like this:

```
public class Student {
  private string _firstName;
  private string _lastName;
  private int _age;
```

```
    public string getFirstName(){
      return _firstName;
    }
    public void setFirstName(string firstName){
      _firstName = firstName;
    }
    public string getlastName(){
      return _lastName;
    }
    public void setLastName(string lastName){
      _lastName = lastName;
    }
    public int getAge(){
      return _age;
    }
    public void setAge(int age){
      if(age > -1 && age < 970){
        _age = age;
      } else {
        throw new Exception("Age must be between 0
        970 years");
      }
    }
  }
```

That's a lot of code that doesn't do anything. It only exists in the service of calling the class members *encapsulated*. In this case, there

might not be a need to control the first and last name fields. I can't think of any restrictions you might put on the kind of data, aside from it being a required field within a user interface. Let's say the boiler-plate is legitimate but we want to be modern and use auto-imple-mented properties. Your code can be reduced to this:

```
public class Student {
  public string FirstName {get; set;}
  public string LastName {get; set;}
```

Auto-implemented properties allow you to use this shortened syntax if you don't have logic to put in the getter and setter methods.

Accessor logic with backing variables

The real power of encapsulation happens when you use the accessor methods to control the state of your object. Your biggest job as a de-veloper is making sure your object is incapable of entering into an in-valid state. For example, take **`Age`**. A negative number in the age property doesn't make sense. Neither does a billion years. The oldest human lifespan ever recorded that I know of was Methuselah, in the Old Testament, who live to be about 970 years old. Since there is no accessor logic, I can make the age 100 years or $1{,}000{,}000{,}000$ years. If I want the controlling logic to limit the input values, I need to create something like this:

```
    private int _age;
```

Here, we create a private backing variable called **_age**. We need this to hold the value since the property is now under the control of accessor logic. It is common to name private variables beginning with an underscore.

Next, we add the logic. We don't really need anything special on the getter:

```
    public int Age {
      get => _age;
```

The setter is where the magic happens:

```
      set {
        if(value > -1 && value < 970){
          _age = value;
        } else {
          throw new Exception("Age must be between
          970 years");
        }
      }
    }
  }
```

Then, we make the property with the auto-implemented method syntax, but this time, there's more to it. Essentially, the syntax for the getter is a fat-arrow function that returns the `_age` backing variable. The setter is straightforward except for the value variable. Where does that come from? It's magic. It's part of the language for use in this scenario. It holds the value of whatever was passed into the setter.

Inheritance

C# is a statically compiled language that supports a classical inheritance model. By statically compiled, I mean the structure of your objects can't change unless you stop your running program, alter the source code for the class, recompile, and rerun. You can contrast C#'s static nature with a language designed to be dynamic: JavaScript.

JavaScript breaks from a great many conventions, not the least of which is that it uses prototypes for inheritance instead of classes. For that matter, it doesn't support encapsulation. It is based heavily on the idea of Lambda functions, which was also novel when JavaScript was invented. JavaScript uses lexical scope instead of the more conventional block scope we find in C#. In short, JavaScript is really weird when compared with C# if C# is the only language you know.

Now that we've established there's more than one way to do inheritance, let's switch back over to how it works in C#.

Let's suppose you are writing software to track the faculty and student body at a school, or university. You need to model a student class – maybe something like this:

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }
    public float GradePointAverage { get; set; }
    public List<Course> Courses { get; set; }
    public void Study()
    {
        Console.WriteLine("I am studying");
    }
    public void TakeTest()
    {
        Console.WriteLine("I am taking a test!");
    }

    public void DoHomework()
    {
        Console.WriteLine("I am doing homework.");
    }
```

```
    }

    public void AskParentsForMoney()
    {
        Console.WriteLine("Hey Dad, do you have a
        minute?");
    }
}
```

Additionally, you need to model a **`professor`** class – maybe something like this:

```
public class Professor
{
    public string FirstName { get; set; }

    public string LastName { get; set; }
    public string EmailAddress { get; set; }
    public bool HasTenure {get; set;
    public List<Course> Courses { get; set; }

    public void GradeTest()
    {
        Console.WriteLine("I am grading a test.");
    }

    public void TeachClass()
    {
```

```
        {
            Console.WriteLine("I am teaching a class.
        }
        public void BegForResearchFunding()
        {
            Console.WriteLine("Hey National Science F
            do you have a minute?");
        }
    }
```

We have a problem here. There are a lot of properties that are redundant since they are used in both classes. OOAD would have us redesign these two classes by abstracting the common properties into a superclass. This leaves only those elements that are unique to the **`Student`** and **`Professor`** classes. The obvious targets for hoisting to the superclass are the following common properties:

- **`FirstName`**
- **`LastName`**
- **`Email`**
- **`Courses`**

Less obvious targets for hoisting include the **`AskParentsForMoney()`** method in the student class and the **`BegForResearchFunding()`** method in the professor class. They effectively do the same thing. The only difference is who is being asked to part with cash.

Let's make a superclass:

```
public abstract class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }
    public List<Course> Courses { get; set; }
    public void BegForMoney(string who)
    {
        Console.WriteLine("Hey " + who + ", do yo
        minute?");
    }
}
```

I've dubbed the superclass **`Person`**. It has the common elements between the student and the professor. Now, let's make the **`Student`** class inherit from the **`Person`** class. To signal inheritance, I'll add a colon followed by the superclass on the first line:

```
public class Student : Person
{
```

Now I'll only keep the properties that aren't defined in the superclass. There is just one:

```
        public float GradePointAverage { get; set; }
```

The same goes for methods. I only need those that are not in the superclass:

```
        public void Study()
        {
            Console.WriteLine("I am studying");
        }
        public void TakeTest()
        {
            Console.WriteLine("I am taking a test!");
        }

        public void DoHomework()
        {
            Console.WriteLine("I am doing homework.");
        }


    }
```

I can do the same thing with the **`Professor`** class:

```
```

```
public class Professor : Person

{
    public bool HasTenure { get; set; }

    public void GradeTest()
    {
        Console.WriteLine("I am grading a test.")
    }

    public void TeachClass()
    {
        Console.WriteLine("I am teaching a class.
    }
}
```

Now, the class only contains the properties and methods unique to the **`Professor`** class. The common properties and methods are in the **`Person`** superclass.

The key to understanding inheritance is to vocalize the relationship between **`Person`**, **`Student`**, and **`Professor`**. We can say a **`Student`** *IS A* **`Person`** (debatable in their freshman and sophomore years), and we can say a **`Professor`** *IS A* **`Person`** (debatable after they get tenure). This vocalization both defines and illustrates the relationship between the superclass, sometimes called the parent class, and its descendants, called subclasses or children.

This leads us to a question: would it ever make sense to instantiate the **`Person`** class? The answer is no. **`Person`** is designed to be a parent class. It isn't intended to be used directly. Classes that are not meant to be instantiated are called **abstract classes**. I've already put that in the class – I just didn't call it out. The first line of the **`Person`** class was this:

```
public abstract class Person
```

The **`abstract`** keyword will prevent direct instantiation.

Just remember that inheritance is a vital part of your OOP toolbox. Inheritance is like eggs. It's part of a balanced breakfast, but a really good breakfast has some fiber in it too. Let's get some metaphorical fiber in our code by learning about interfaces. Interfaces do nothing to lessen your risk of heart disease like a good bowl of low-sugar oatmeal does. Using them might lower your stress levels later when you discover how flexible your code will be once you incorporate them, which is just as good. If you eat low-sugar oatmeal *and* use interfaces in your code, maybe you'll give Elvis a run for his money in terms of lifespan.

Interfaces

An **interface** defines all or part of the structure a class must take by defining the public method signatures and properties that must be present in the class. This is a powerful tool that allows you to create a specification for an object's behavior. The power of interfaces is multiplied by the fact that you are not limited to implementing a single interface versus using subclassing where you are only allowed a single parent.

Interfaces serve to loosely define behavior or type. Here's a shout-out to all the *Unity 3D* developers in the crowd: imagine creating a video game where the player battles zombies and ancient creatures conjured by the imagination of H.P. Lovecraft – oh, and cats because cats are scary. Each monster could have its own class. You could create an interface to define behaviors such as fighting, running, eating humans, and more. Such a video game might look like *Figure A1.9*:

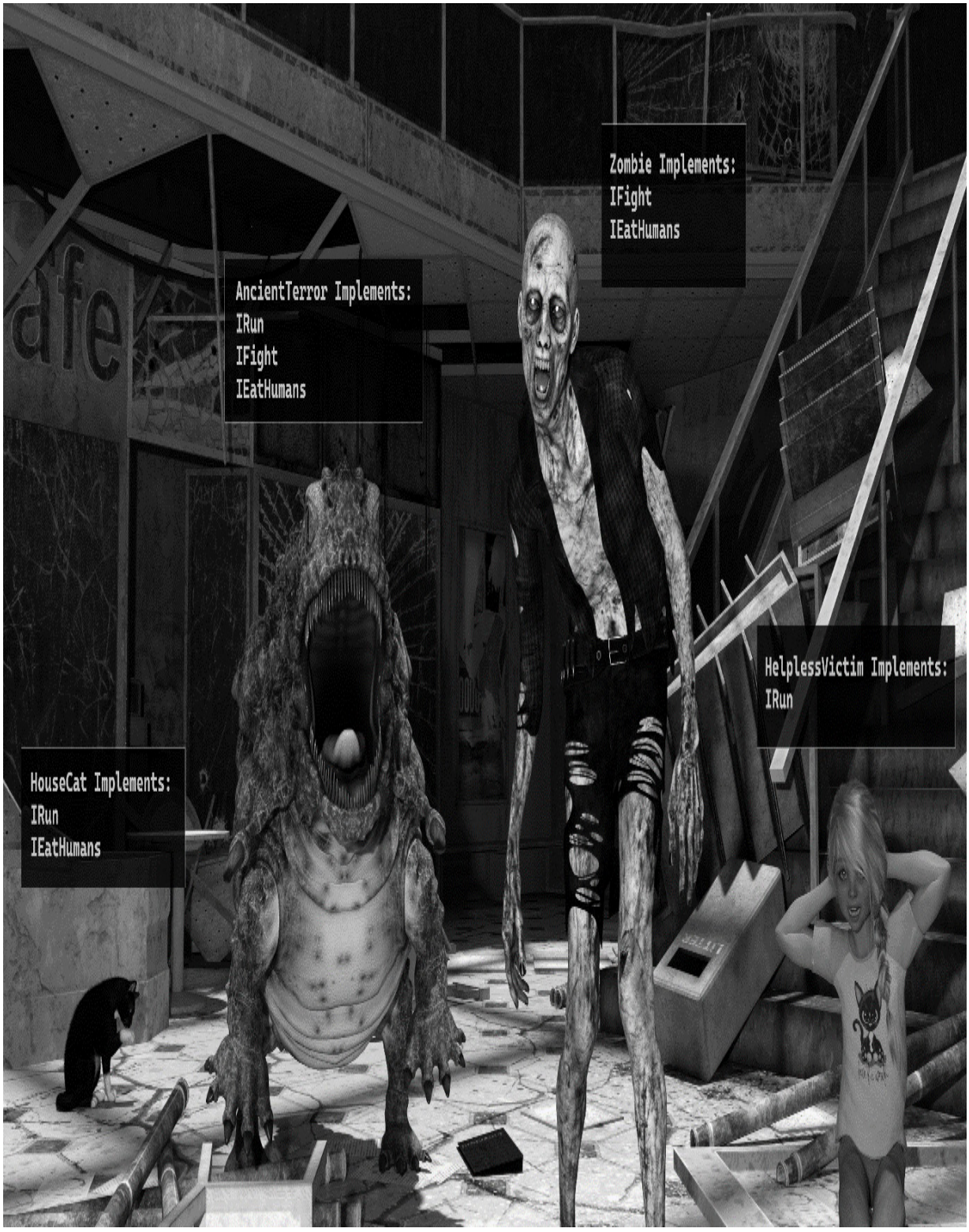


Figure A1.9: A terrible video game design that masterfully uses interfaces to map behaviors to characters in the game.

With these interfaces, you could define behaviors on objects that don't necessarily fit together in an inheritance chain and wind up with game code that is very easy to extend.

The key benefit to interfaces versus inheritance is flexibility. In *Chapter 2*, we learn about SOLID principles and we learn about techniques for avoiding tightly coupling your classes together. If you define a property on a class with a type of **`Monster`**, then only **`Monster`** or a subclass of **`Monster`** will work. Your class is tightly coupled to **`Monster`**. If you define it by an interface, such as **`IEatHumans`**, you can pass any object that implements the interface as long as all you need are the methods defined by that interface.

Defining interfaces

Practically speaking, if next week the game director decides to add a new monster to the game, we don't have to reshuffle our object hierarchy if the new monster doesn't happen to fit the structure we've created. We just make a new class and use the interfaces to define its structure. Let's code out what we have in *Figure A1.9* in C#, so we can see what this looks like in code instead of just a diagram. There are three interfaces used by the four characters:

1. **`IRun`**
2. **`IFight`**

3. **`IEatHumans`**

Let's start with the **`IRun`** interface. Making interfaces is pretty easy. All you need to do is specify the basics of the method signature – specifically, the return type, the name of the method, and the names and types of the arguments:

```
public interface IRun
{
    void Jog();
    void Sprint();
}
```

That was easy. What we've done here is to specify that any class implementing this interface must have two methods, both of which have a **`void`** return type. One must be called **`Jog()`**, which takes no arguments, and the other must be called **`Sprint()`**, which also takes no arguments. If you're using a good IDE, any class you create that implements an interface will mark your code with red squiggles until you meet all the requirements of the interfaces. Let's do **`IFight`** next:

```
public interface IFight
{
```

```
    void Attack();
}
```

IFight says any class that implements this interface must have a method called **Attack()**, which returns **void** and takes no arguments:

```
public interface IEatHumans
{
    void Chomp();
}
```

IEatHumans requires the implementing class to have a method called **Chomp()**, which takes no arguments and returns **void**. Interfaces are easy. Let's look at how they're used.

Implementing interfaces

First, let's make a class for **HelplessVictim**. The class implements the **IRun** interface. The syntax is the same as it is for inheritance. We use a colon to indicate the interface implementation, which would be the same if we were inheriting from a superclass:

```
public class HelplessVictim : IRun
{
    public void Jog()
```

```
    {
        throw new NotImplementedException();
    }
    public void Sprint()
    {
        throw new NotImplementedException();
    }
}
```

To satisfy the interface, we must implement two methods, **`Jog()`** and **`Sprint()`**, exactly as specified in the interface. That's nice. What if we need to implement more than one interface? It's not possible for a class to have two parent classes. C# mercifully doesn't support multiple inheritance, which is the leading cause of clinical insanity among C++ programmers. However, classes can implement as many interfaces as you'd like. Let's make the **`AncientTerror`** class, which implements three interfaces – **`IRun`**, **`IFight`**, and **`IEatHumans`**:

```
public class AncientTerror : IRun, IFight, IEatHu
{
    public void Chomp()
    {
        throw new NotImplementedException();
    }
    public void Attack()
    {
        throw new NotImplementedException();
```

```
        }
        public void Jog()
        {
            throw new NotImplementedException();
        }
        public void Sprint()
        {
            throw new NotImplementedException();


        }
    }
```

IRun requires the same **Jog()** and **Sprint()** methods to be implemented. Note that it isn't necessary to have the exact same implementation for the different classes. The instance methods simply have to conform to the interface. **IFight** requires the addition of a method called **Attack()**, which returns **void** and takes no arguments. **IEatHumans** requires us to add a **Chomp()** method per the interface.

I'll save the scariest monsters, the zombie and the house cat, as an exercise in the questions section at the end. Challenge yourself and see whether you can come up with the basic implementation for these remaining two classes!

IDEs for C# development

Whenever I work in a new or unfamiliar language, the first thing I want to know about is the tools used to work in that language. Good tools make learning and working with the language much easier. Microsoft realized this, and when they released the C# language and the accompanying .NET runtime, they also released Visual Studio – an IDE written specifically for use with C# and another very popular programming language called Visual Basic.

Visual Basic was Visual Studio's predecessor. During the 1990s, Visual Basic was the most widely used development language product from Microsoft. The company also sold an IDE geared for C++ development called Visual C++, and briefly and ineffectively dabbled in Java with Visual J++. Of these toolkits, Visual Basic was by far the most important. At the time, Visual C++ was used by "serious" developers. Microsoft Windows is written in C and C++, so naturally, the tooling in Visual C++ was first and foremost designed to support that effort. Corporate software really became possible and mainstream with the BASIC language. In fact, the BASIC language formed the cornerstone of Microsoft itself. Bill Gates bought the rights for a BASIC compiler, and along with the **Microsoft Disk Operating System** (**MS-DOS**), he formed what would become one of the largest software companies in the world.

Visual Basic was designed by Alan Cooper. Cooper was a visionary. The Visual Basic UI was the first WYSIWYG design platform for com-

puter software. Before the World Wide Web, we built software that ran solely on the desktop, and Visual Basic was the innovation that drove the industry.

But that's enough history. Today, there are three important IDEs I want to tell you about. Chances are you've used one of them already. These tools include the following:

- Visual Studio
- VS Code
- Rider

There are other IDEs out there, but these three are the most popular, complete, and frankly, important. If you use a Mac, it is worth mentioning that Visual Studio for Mac is not a port of Visual Studio. Earlier, I mentioned the open source version of C# called Mono. The team that developed Mono created an IDE called *Monodevelop*. It looks a lot like Apple's X-Code IDE, and it was designed to allow Linux developers to write C# programs. *Monodevelop* was open source and was ultimately forked to become *Xamarin Studio*. *Xamarin Studio* is an IDE geared toward mobile development. Microsoft bought out Xamarin several years ago. A separate fork of the technology became Visual Studio for Mac. I mention it because it looks nothing like the IDEs we'll be covering, so if you're on a Mac, my screenshots won't help much.

To learn patterns, you only need a fraction of the functionality these IDEs provide. In this section, I wanted to walk you through creating the two project types found in this book: the command-line project and the library project.

Command-line programs are the simplest programs you can make that will actually run. They have no user interface and are run from, you guessed it, the command line. Libraries are code projects used to house objects meant to be shared between projects. In *Chapter 3*, I create a shared library that is used across many other chapters because they share a lot of the same code, as typing the same classes over and over would get very tedious.

Let's take a look at these IDEs starting with Visual Studio.

Visual Studio

Visual Studio needs no further introduction. You can get a copy at https://www.visualstudio.com. The IDE comes in three editions:

- Visual Studio Community
- Visual Studio Professional
- Visual Studio Enterprise

The first two are effectively identical except for how they are licensed. The Community Edition is free assuming you meet the licensing re-

quirements. These things tend to change over time, so I'm not going to try to quote those requirements here. Review the Visual Studio website to see whether you qualify for a free license.

Visual Studio Professional is the paid version of the same tool. If you work for a company with a certain number of developers, or that makes a certain amount of revenue, you need to buy a subscription.

Visual Studio Enterprise is a more expensive version of the paid editions and ships with a myriad of extra features not present in the Professional Edition.

Any of these editions will be fine for use with this book. I'll be demonstrating the Community Edition, which, again, is indistinguishable from the Professional Edition.

Creating a command-line project

Having downloaded, installed, registered, and launched the IDE, you are greeted with a screen that looks something like *Figure A1.10:*

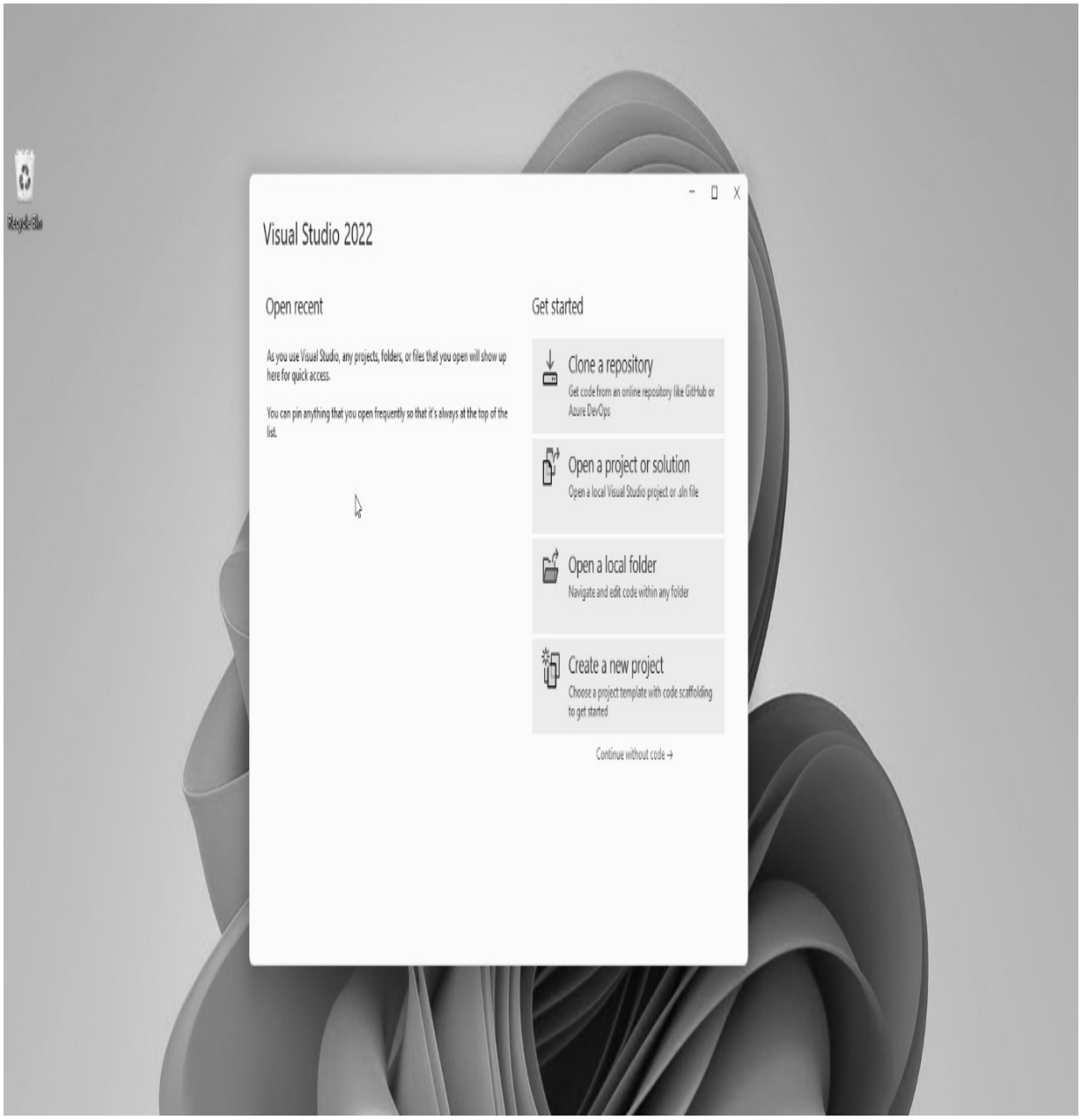


Figure A1.10: The opening screen for Visual Studio 2022.

Click on the **Create a new project** button in the lower-right corner of the window. Doing this presents you with *Figure A1.11*:

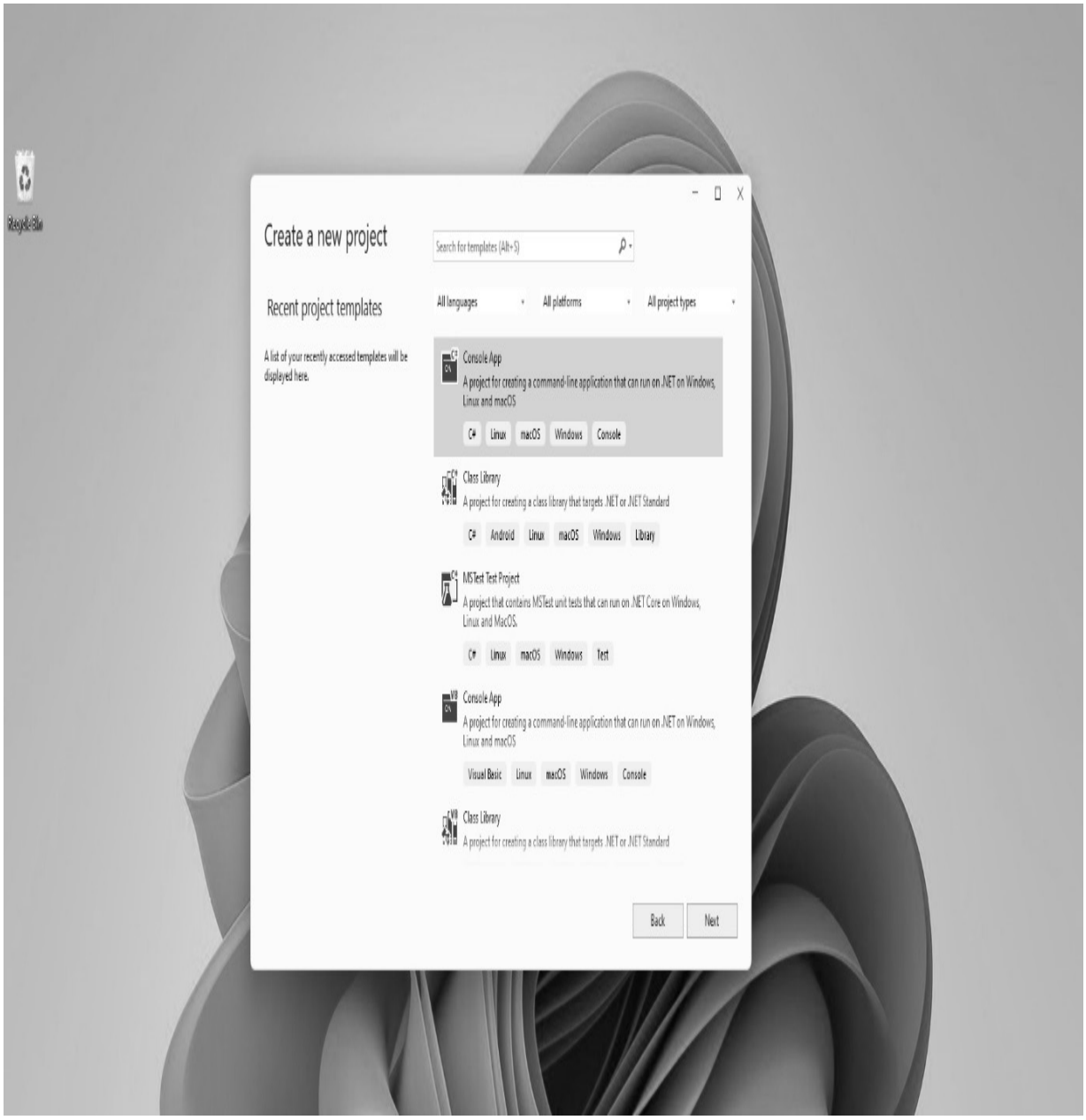
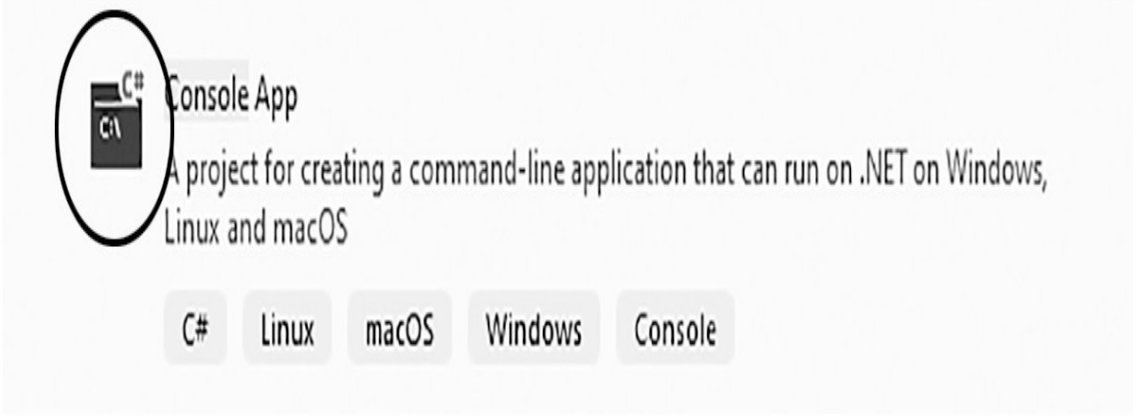


Figure A1.11: Create a new project in Visual Studio using this dialog.

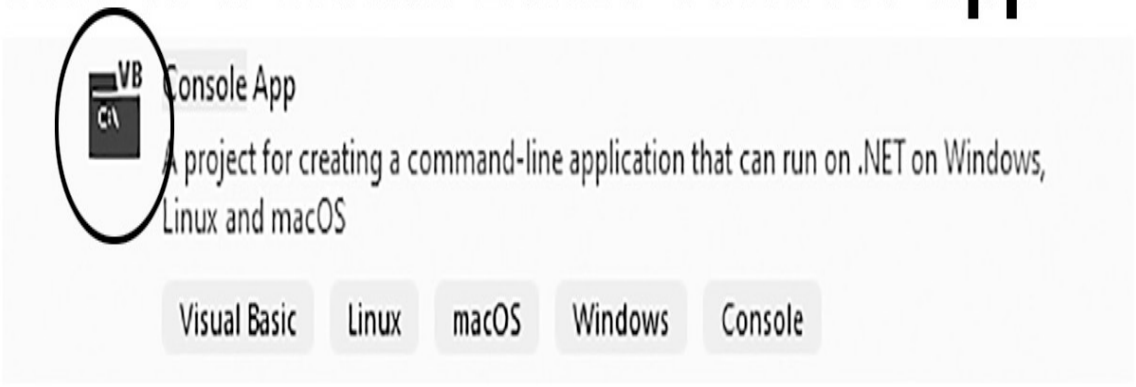
You're looking for the *Console App* project in C#. If you did a full install of Visual Studio, beware that the tool might show you Visual Basic projects as well as C#. Make sure you pick the C# version of the

project template. Note that the icons on the templates denote the language they use, as shown in *Figure A1.12*:

This is a C# console app



This is a Visual Basic console app



This is a F# console app

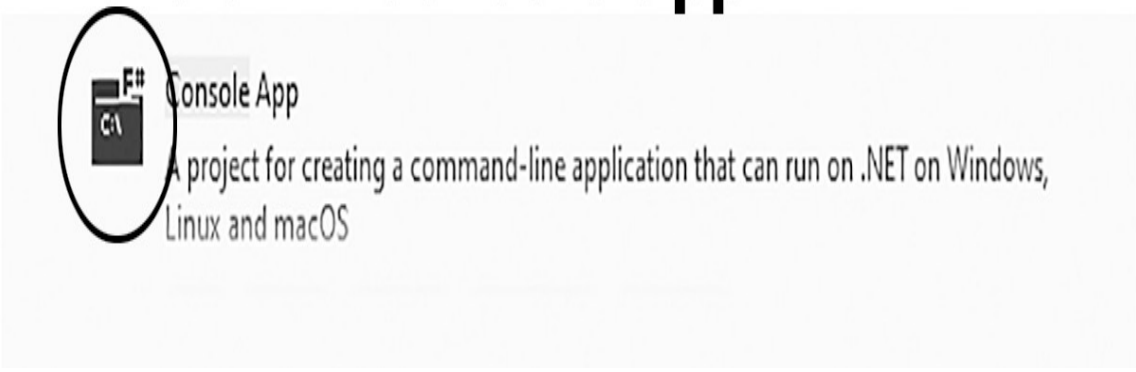


Figure A1.12: Watch the icons on the project templates; it is easy to accidentally generate a new project in the wrong language.

If you don't see the **Console App** project template, you can search for it using the search dialog at the top, as shown in *Figure A.13*:

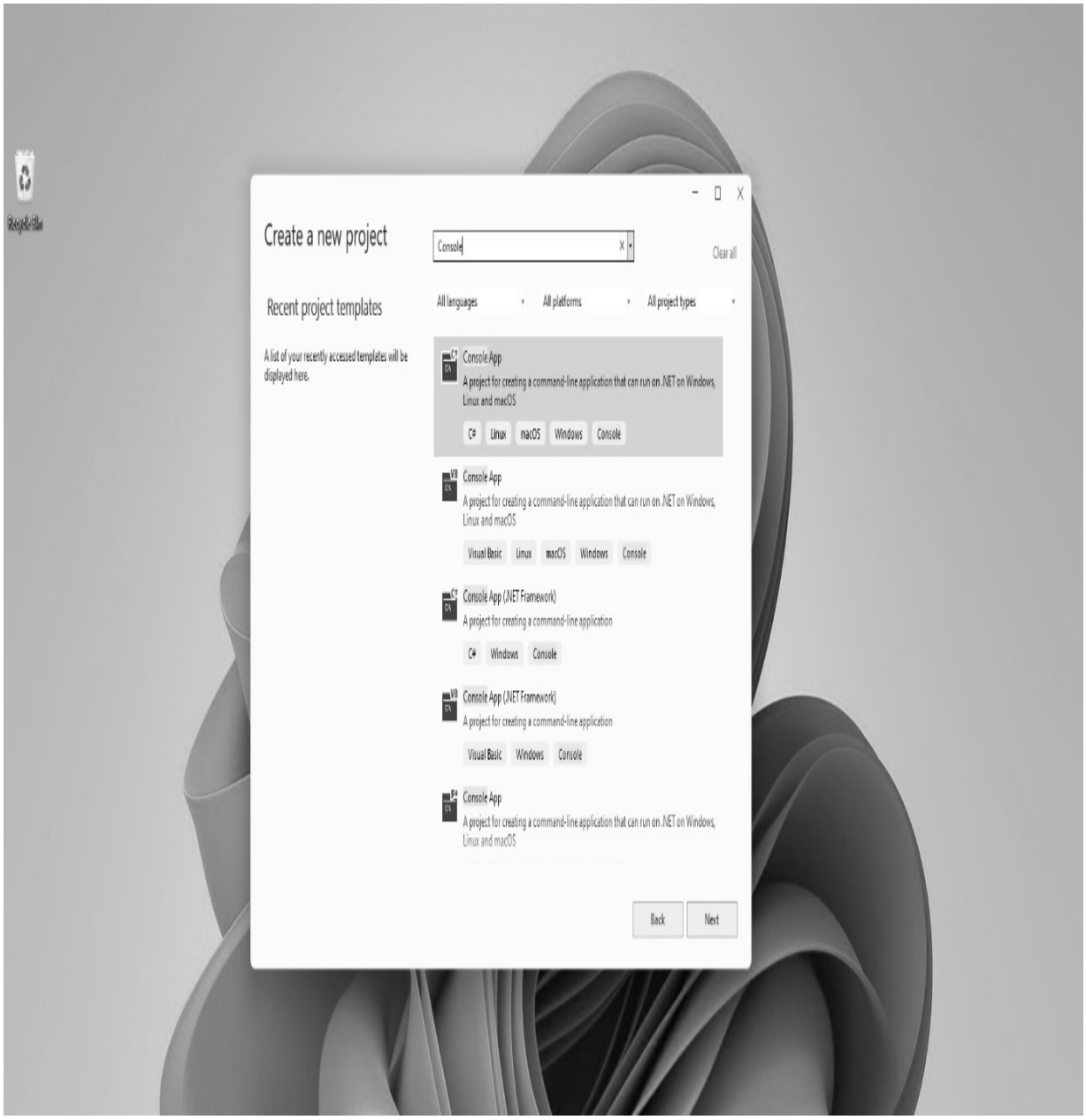


Figure A1.13: You can search for the console app template if you don't see it listed.

Click on the **Console App** template, and click on **Next** in the lower-right corner of the dialog. This takes you to a dialog titled **Configure your new project**, as shown in *Figure A1.14*:

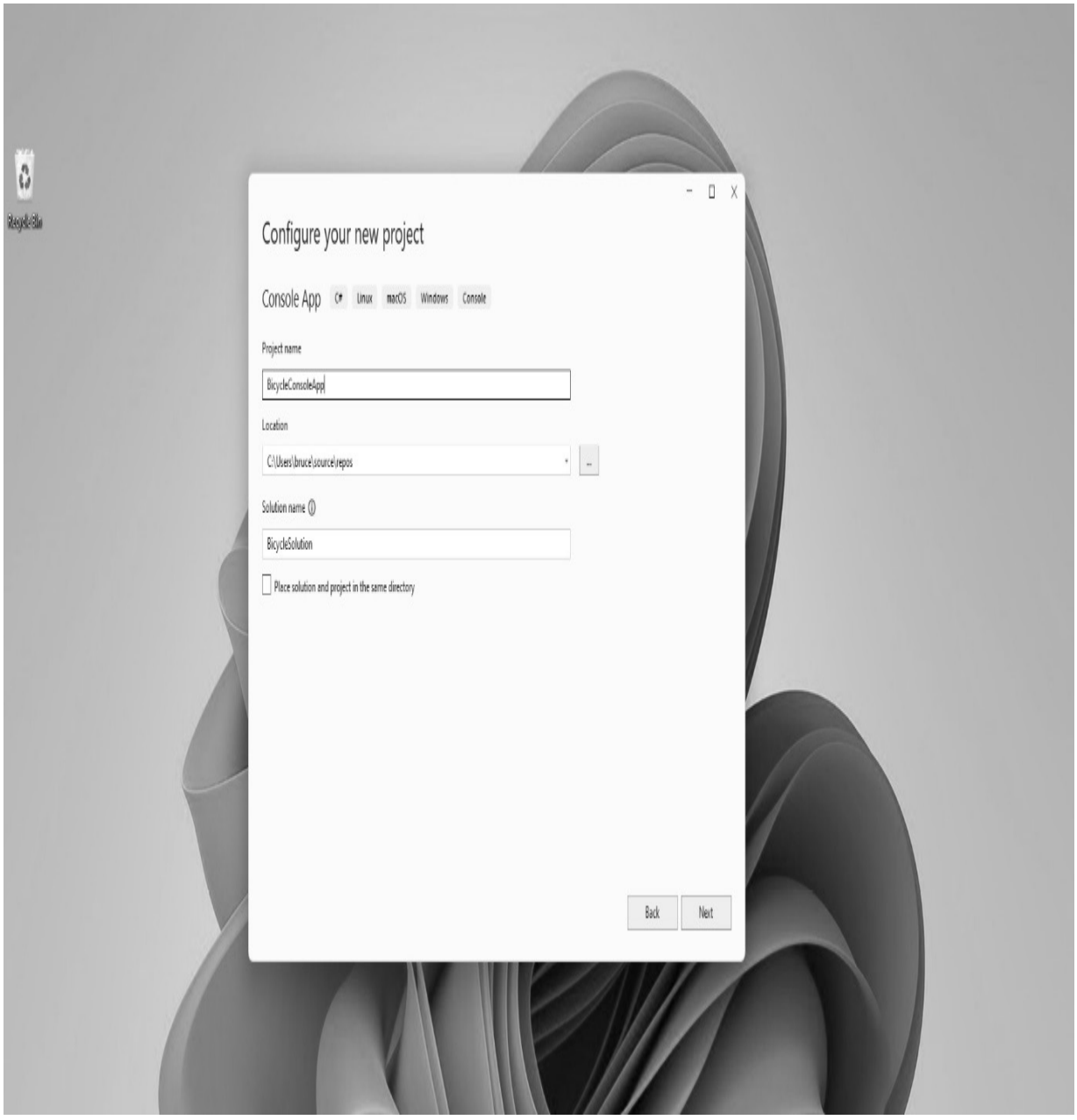


Figure A1.14: Configuring your new project by giving it a name and location

This dialog allows you to name your project and specify its location on your hard drives. You can also choose to name your solution differently than your project. A solution is a collection of projects bound

into one set of files. This allows you to work on multiple related projects in a convenient way. Maybe you have a web application and a related command-line program that belong together. A solution allows you to store the related projects together. If you are storing related projects together, you might not want to name the solution the same as the project. For the exercises in this book, it doesn't matter.

We're going to name the project **`BicycleConsoleApp`**. We're going to name the solution **`BicycleSolution`**. Use a location on your computer that is convenient. The checkbox at the bottom specifies a folder structure. You can just leave it unchecked. Your project configuration should resemble *Figure A1.15*. Click on the **Next** button in the lower-right corner of the dialog:

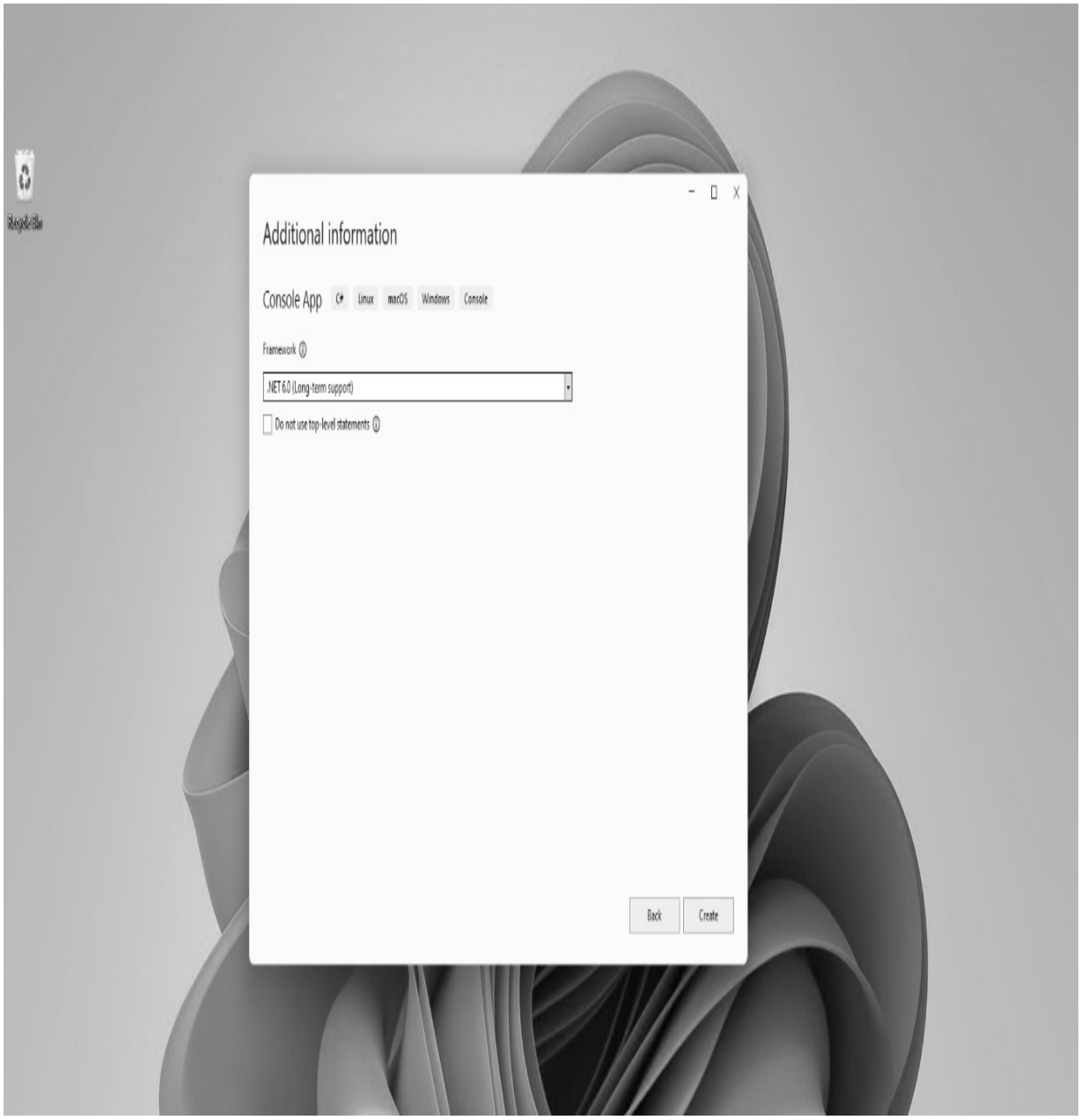


Figure A1.15: The Additional information dialog lets you set the framework for the project.

The **Additional information** dialog allows you to pick the .NET Framework type that you'll use for development. The default at the time of this writing is **.NET 6.0 (Long-term support)**. Just accept the

defaults as they are and click on the **Create** button. This will complete project creation, and you'll see the full IDE, as shown in *Figure A1.16*:

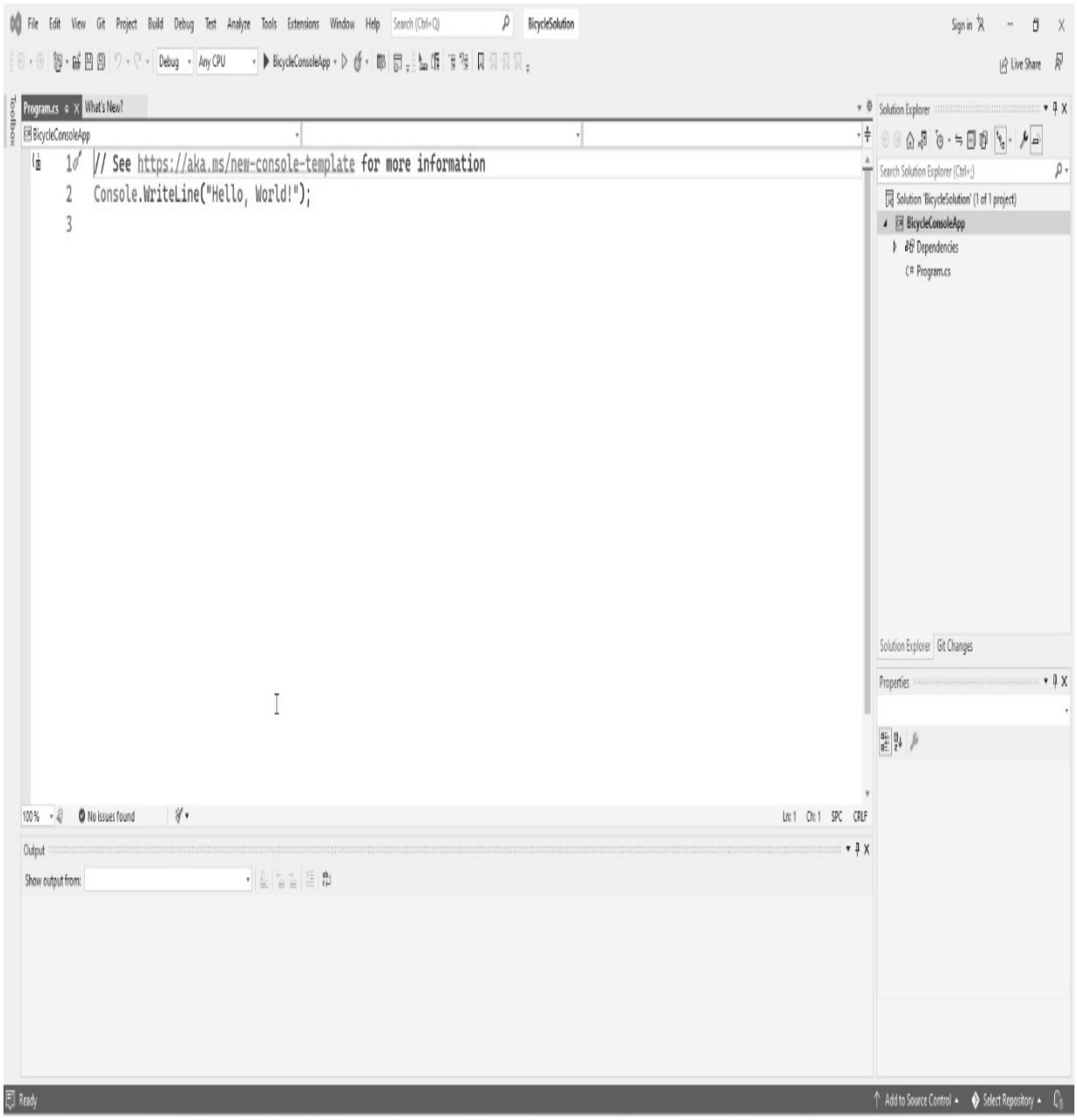


Figure A1.16: The new project is created and ready in Visual Studio.

Your project is ready! You can start developing your code.

Adding a new class

Once your project has been created, you'll be needing to add new classes. To add a class, right-click on the **`BicycleConsoleApp`** project, and you'll find a context menu. Find the **Add** option and hover over it. A second menu expands. Find **Class...** at the bottom of the menu and click on it. You can see the menu layout in *Figure A1.17*:

Figure A1.17: The menu item for adding a class accessed from Solution Explorer in Visual Studio.

The next step is to specify what you wish to add using the dialog in *Figure A1.18*. In this case, we're adding a class. Note there is also an option for adding an interface. You'll need both, but right now, let's focus on adding a class:

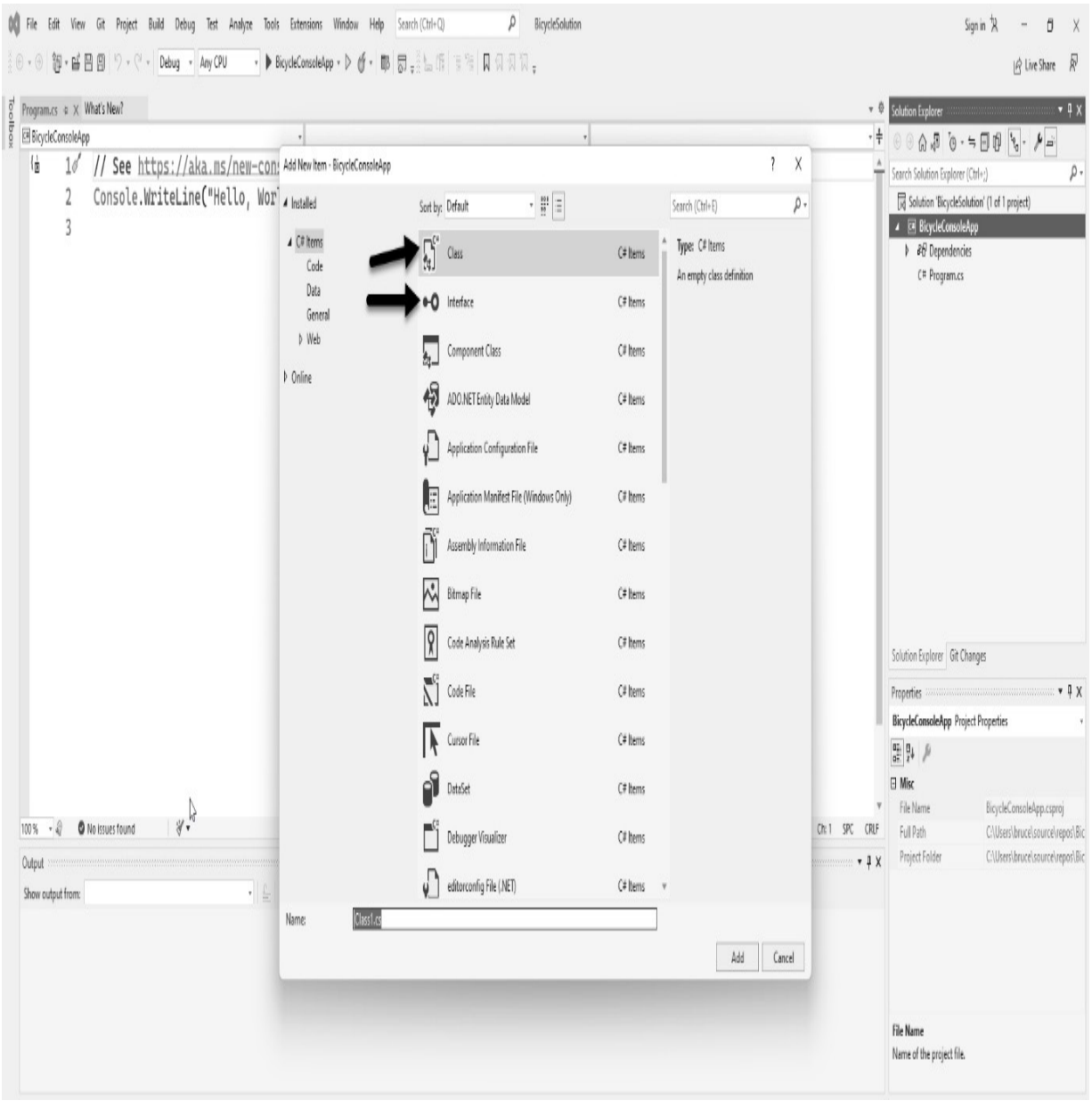


Figure A1.18: The Add New Item dialog in Visual Studio can be used to add anything, including classes and interfaces.

Click on **Class**, then give the file a name that matches the name of the class you want to create. Usually, I honor the age-old Java convention of having one class per file. Click on the **Add** button, and the class will be created and added to your project.

To add an interface, follow the same procedure, but select **Interface** instead of **Class**. By convention, C# interfaces always start with the letter I (as in “Interface”), so name your file the same as what you plan to name your interface.

Adding a library project to the solution

When you created the command-line app project, Visual Studio also created a solution. Solutions are containers for projects. Even if you only have one project, it will be inside a solution. You can add more projects to the solution. This is a handy way of keeping related projects together.

Library projects are used to house reusable code designed to be accessible between projects. In *Chapters 3–5* of this book, I use a library for common classes like those that model the basic parts of the bicycle classes we’re building in those chapters. In the real world, you generally put your business logic in a library. When you do this, you can leverage that logic in a web application, a mobile application, and a desktop application without repeating your code in three places.

Creating a library project is easy. Right-click on the solution in the solution explorer. Locate the **Add** option, hover over it, and click on **New Project**. A new project dialog appears, as shown in *Figure A1.19*:

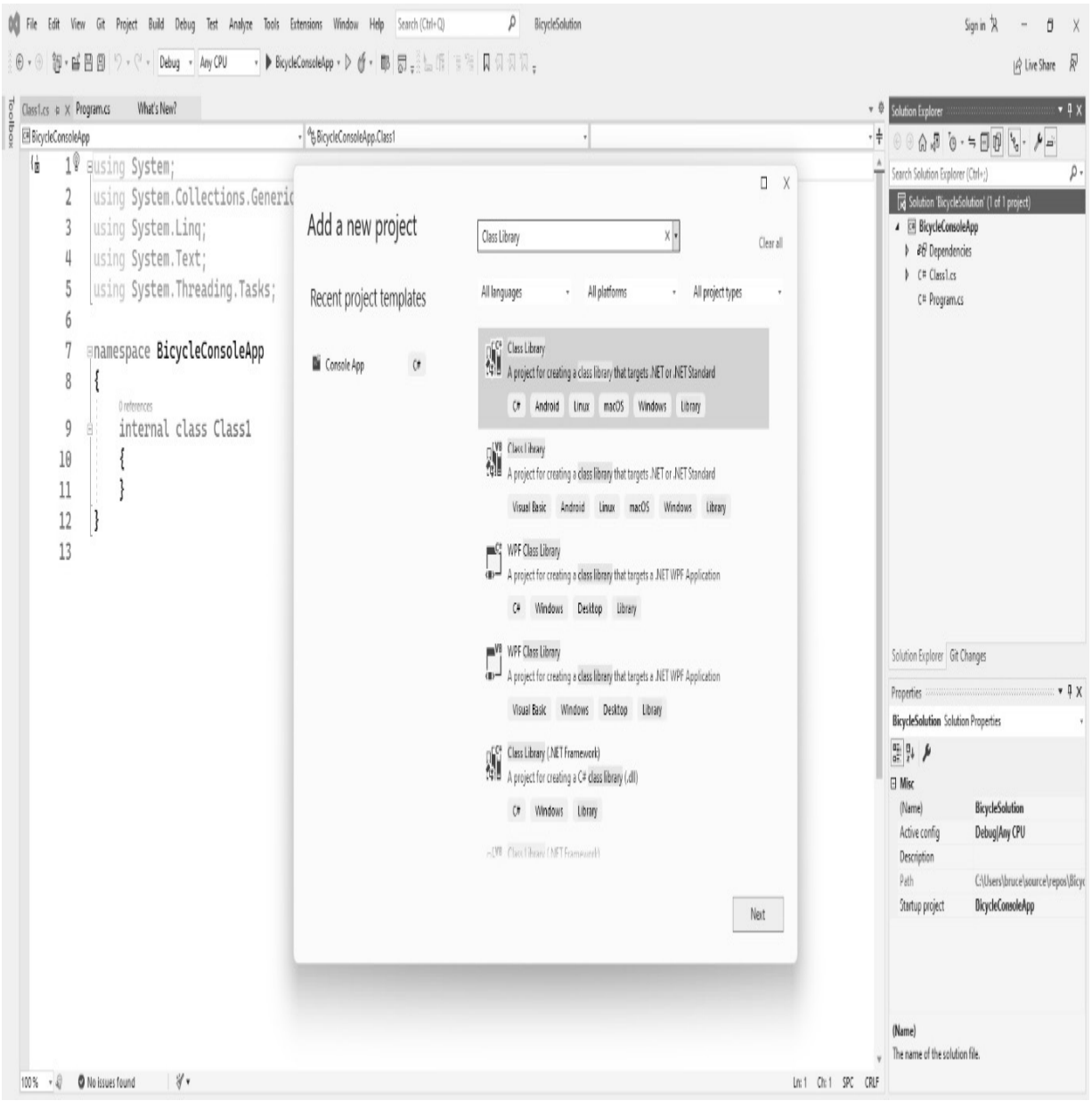


Figure A1.19: The context menu in Visual Studio used to add a new project to the solution.

Locate the **Class Library** project template. If you don't see it, type the term **`Library`** into the search box at the top of the dialog. Make sure the class library template is using the C# language and not some other language such as Visual Basic or F#. Click on the **Class Library**

project option, and click on the button labeled **Next** near the bottom-right corner of the dialog. You're taken to another dialog asking for your project name and its location. Let's call the project **`BicycleLibrary`**. The location should be defaulting to the location of the solution and, by extension, the console app project we just made. Click on the **Next** button.

The next dialog allows you to select the .NET runtime for your project. Generally, you want these to match. We picked .NET 6.0 for the console app, so we should pick the same thing here. Click on the **Create** button, and Visual Studio will generate the project's files for you.

Linking the library project to the console project

Before you can use the library, you must create a reference to it in the **`BicycleConsoleApp`** project. To create the reference, right-click on the **Dependencies** option in the console app, and then click on **Add Project Reference**. This reveals a dialog, as shown in *Figure A1.20*:

Figure A1.20: The menu used to add project dependencies to a project in Visual Studio.

The **Projects** option is at the top of the dialog and is probably already selected. You should see the name of your library project in the list. Click on the checkbox next to the library project you just made, and then click on **OK**.

Your library is now available within your console app project.

Referencing the library

When you created the library project, Visual Studio created a file called **`Class1.cs`**. Let's add a method to that class and then see how to reference it in the console app.

Open the **`Class1.cs`** file in the class library project. Add this code:

```
namespace BicycleLib;
public class Class1
{
  public string SayHello()
    {
        return "Hello from the Bicycle Library!";

    }
}
```

Save the file.

Now, open the **`Program.cs`** file in the console app project. You should see a "Hello World" program. Replace the starter code with this:

```
using System;
using BicycleLibrary;
var test = new Class1();
Console.WriteLine(test.SayHello());
```

Notice that as you type the code (type it, don't copy and paste – copy and paste teaches you nothing!), you'll find that IntelliSense is giving you hints on the method you added in the library. Effectively, you are using the code in the library as if it were part of the console app project!

There's just one thing left to do: build the project.

Building and running the console project

C# applications are compiled. If you're used to an interpreted language such as JavaScript or Python, you might not be familiar with this step. The C# compiler, code-named **Roslyn**, has been interacting with you this whole time. The compiler checks your code in Visual Studio as you write it. It flags mistakes and underlines things you should fix. Additionally, it gives you the IntelliSense for your code completion. A distinct difference from what you're seeing in Visual Studio is that this tool uses the compiler and a feature of the .NET called **introspection**. This means Roslyn can literally see the classes and code within your project. This makes your Intellisense hints ex-

tremely accurate. If you've seen code completion in other editors such as Sublime Text or VS Code, the hints you are getting are not nearly as accurate. Those are coming from, at best, depending on which editor we're talking about, an index of the code you have in place so far. When you type `foo`, where `foo` is any given object, an augmented text editor will probably give you every possible option that could come after that dot. Really good editors might use probability or AI to narrow the field. Visual Studio only shows you viable options based on introspection. The latest versions of Visual Studio are starting to include AI features. When AI is combined with IntelliSense, the results are frightening. You can expect Visual Studio to write entire blocks of code for you, not just boilerplate; the actual methods in your code.

Better code hints and early warnings about code mistakes are among the benefits of working with compiled languages. The compiler doesn't allow the most common types of mistakes, so you find and fix them early because you must. The product of a compiled language is also generally faster in terms of execution speed.

Building in Visual Studio is very simple, and there are a few ways to do it. The most straightforward is to run your program. To do this, just click on the green arrow in the toolbar. *Figure A1.21* shows the location of the **Run** button. Unfortunately, *Figure A1.21* isn't in color. The book editor's bosses muttered something about budgets and money

not growing on trees. I haven't met the boss of bosses at Packt, but I suspect they used to hang out with my father (God rest his soul) *A LOT*. The good news is you really can't miss it in the IDE:

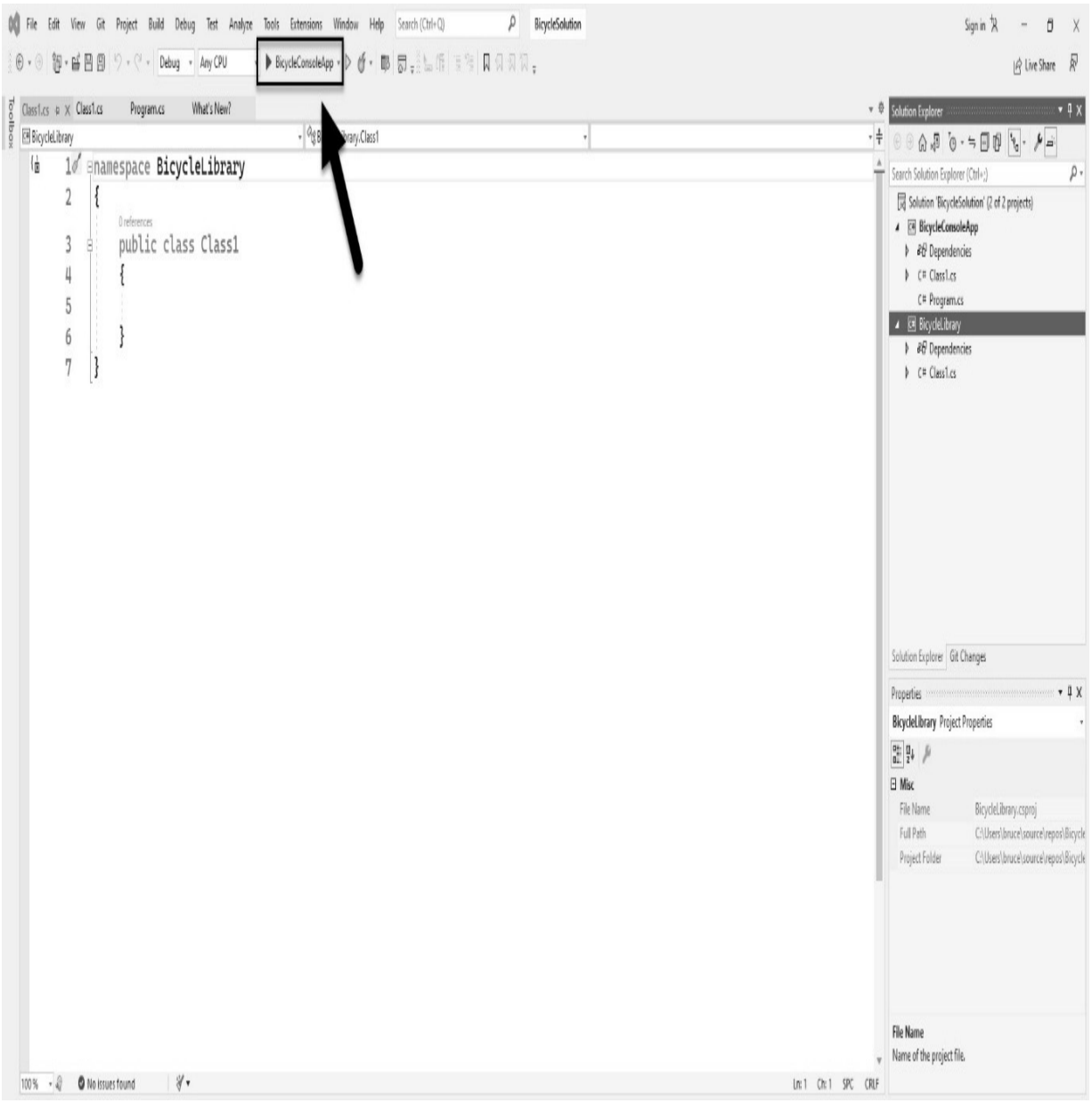


Figure A1.21: Locate the green triangle in the IDE; this is the Run button.

When you click on the **Run** button, your program builds, and it is then executed with an attached debugger. You'll see your running program, as shown in *Figure A1.22*:

Figure A1.22: The program runs in a window.

To stop the program, press your computer's *Any* key in the running console app window:

Figure A1.23: Press the Any key to exit Visual Studio's program run; if your computer doesn't have an Any key like mine, you can use the spacebar.

The *Any* key is common on older keyboards. It does limit your ability to write code using the letter *H*. Since I live in Texas, I just use *J* instead. It's a small price to pay for being able to quickly exit from your running programs.

If your computer doesn't have an *Any* key, just press the spacebar. Note that the pause message isn't part of your program, and it won't be there if you publish your application. This is Visual Studio pausing the application. Console apps tend to run and exit in under a second. If something goes wrong, the program will finish, display an error,

then kill the window before you have a chance to see anything. So, Visual Studio does you a solid and freezes the window before the program exits so that you can inspect it. Your program should have run without any errors.

This concludes our brief tour of Visual Studio. If you're interested in a video walk-through that coaches you through setting up a Visual Studio development environment in Windows, you'll find links for this at https://csharppatterns.dev.

VS Code

VS Code bears a similar name to Visual Studio but is an entirely different beast. Rather than being a full-fledged IDE, VS Code is really an augmented text editor rather than a true IDE. We just covered Microsoft's Gold Standard for editing tools. Why should I even talk about VS Code? The answer: VS Code currently has over 50% of the market share, making it the most popular tool used by most developers every day. There are some good reasons for this:

- VS Code takes a fraction of the space as Visual Studio, which all-in requires nearly 45 GB of space. You can reduce the footprint of Visual Studio, but VS Code is still going to boast a smaller footprint.

- VS Code launches very quickly. Visual Studio doesn't. You can use VS Code to quickly look at a repository you pulled off GitHub or some folder a colleague shared with you. Launching Visual Studio is a commitment. If you're on a less powerful computer, you can likely catch up on your emails, take a quick peek at social media, get some coffee, and call your mother before you get the IDE's opening screen. By the way, call your mother! She misses you.
- VS Code works with nearly any programming language. Naturally, it works well with Microsoft-supported languages and other popular languages such as Golang or Rust. Also, you'll find extensions that allow you to use it with oddball languages. The original GoF book used the SmallTalk programming language, which is offered by two extension vendors. You can find Ada, Haskell, and many more. Visual Studio is mainly used for mainstream work in Microsoft language products.
- VS Code has a consistent user experience across all operating systems. Visual Studio is a Windows-only program. It won't run in Linux. Visual Studio Mac is a completely different program and looks nothing like the Windows version. This consistency makes the tool a popular choice for schools, universities, and code boot camps like the one I teach at Southern Methodist University.

VS Code has a lot going for it. Like Visual Studio, VS Code can also be downloaded at https://www.visualstudio.com.

If VS Code is going to be your editor of choice, you're going to need to install the .NET Core SDK to give you the tools and compilers to work with C#.

Installing .NET Core

You need the .NET Core SDK if you are going to work with VS Code. You can download it from https://dotnet.microsoft.com/en-us/download. Naturally, that web address might change after this book is published, in which case you'll have to be resourceful. Installing it is straightforward.

With VS Code and .NET Core SDK installed, you're ready to get started. If you read the steps for setting up a project in Visual Studio, the steps for VS Code are very different.

Creating a new solution

Visual Studio creates a project and a solution in one step. This is VS Code. Each step is performed separately, and it is all done from the command line. Technically speaking, VS Code isn't really involved. You use the **`dotnet`** command-line tool that was installed when you installed the .NET Core SDK. You can do all of this in Windows Terminal.

Since Windows Terminal comes with Windows 11, there is nothing further to install. If you've never used it, click on **Start**, and then search for **`Terminal`**, as shown in *Figure A1.24*. Launch the app. If you're still using Windows 10, search for **`PowerShell`** instead of **`Terminal`**:

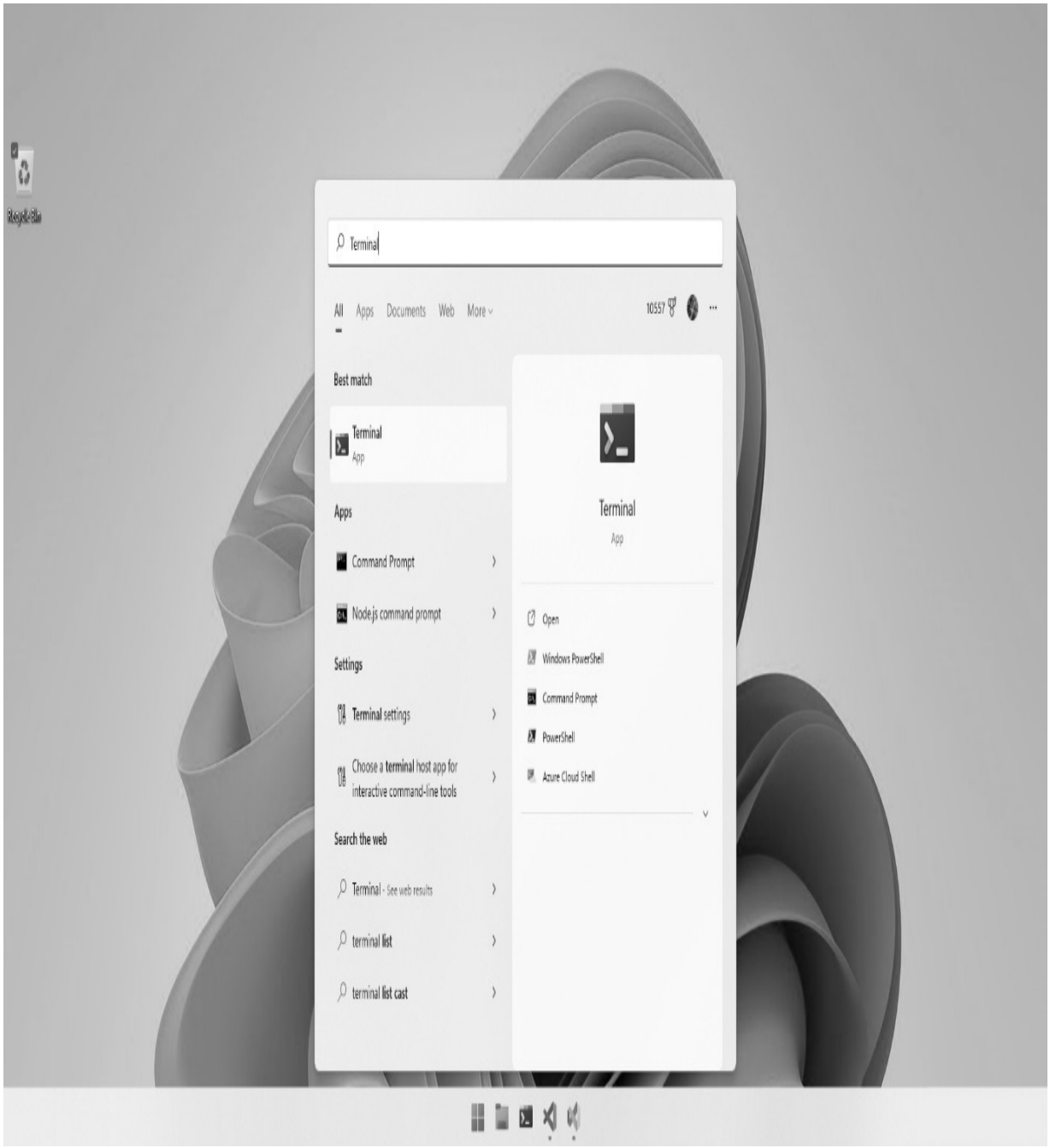


Figure A1.24: Most of the project setup work with VS Code happens in the Terminal window.

With Terminal (or PowerShell) open, type this command:

```
dotnet new sln -o BicycleSolution
```

Here, **`-o`** is for *output*. This tells the command to create a folder for the solution. In addition to creating the folder, the command generates a set of files in the **`BicycleSolution`** folder. Let's take a look. Type this:

```
cd BicyleSolution
```

Then, type in this:

```
dir
```

The **`dir`** (directory) command will list all the files in the current working directory. You should see something like *Figure A1.25*:

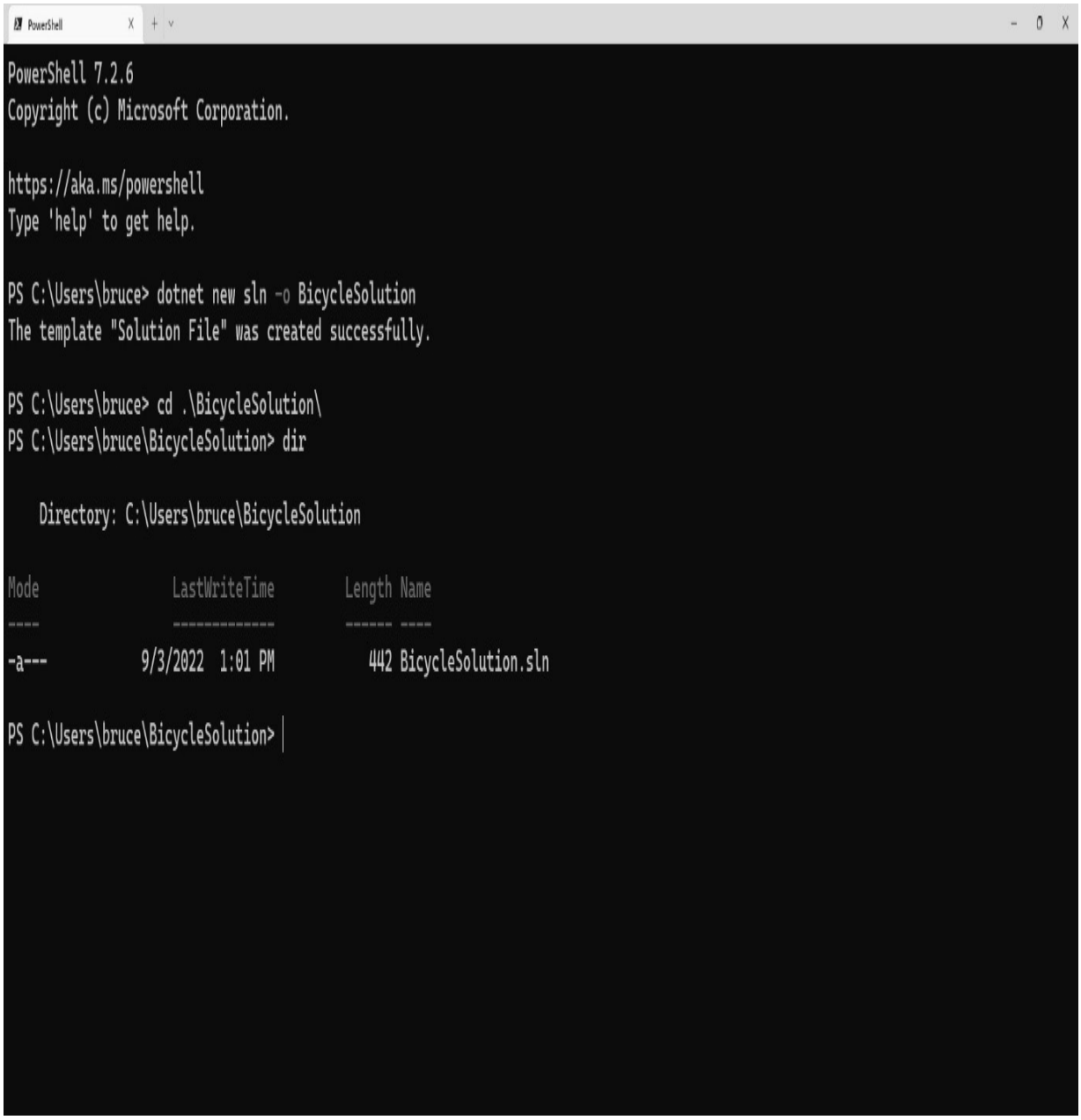


Figure A1.25: The result of our new solution command.

We have a solution, but there's nothing in it. We need some projects to make the solution useful.

Creating a command-line project

Next, we'll add a console application to the current solution folder. First, **`cd`** into the **`BicycleSolution`** folder. Going forward, the commands I'm giving you are based on the assumption that your present working directory is the **`BicycleSolution`** folder.

To add the console app to the solution, type this command into the Terminal window:

```
dotnet new console -o BicycleConsoleApp
```

This creates a new console app. Again, **`-o`** is for output. This tells the command what to call the project. The **`dotnet`** command generates the boilerplate files for the console app project, as shown in *Figure A1.27*:

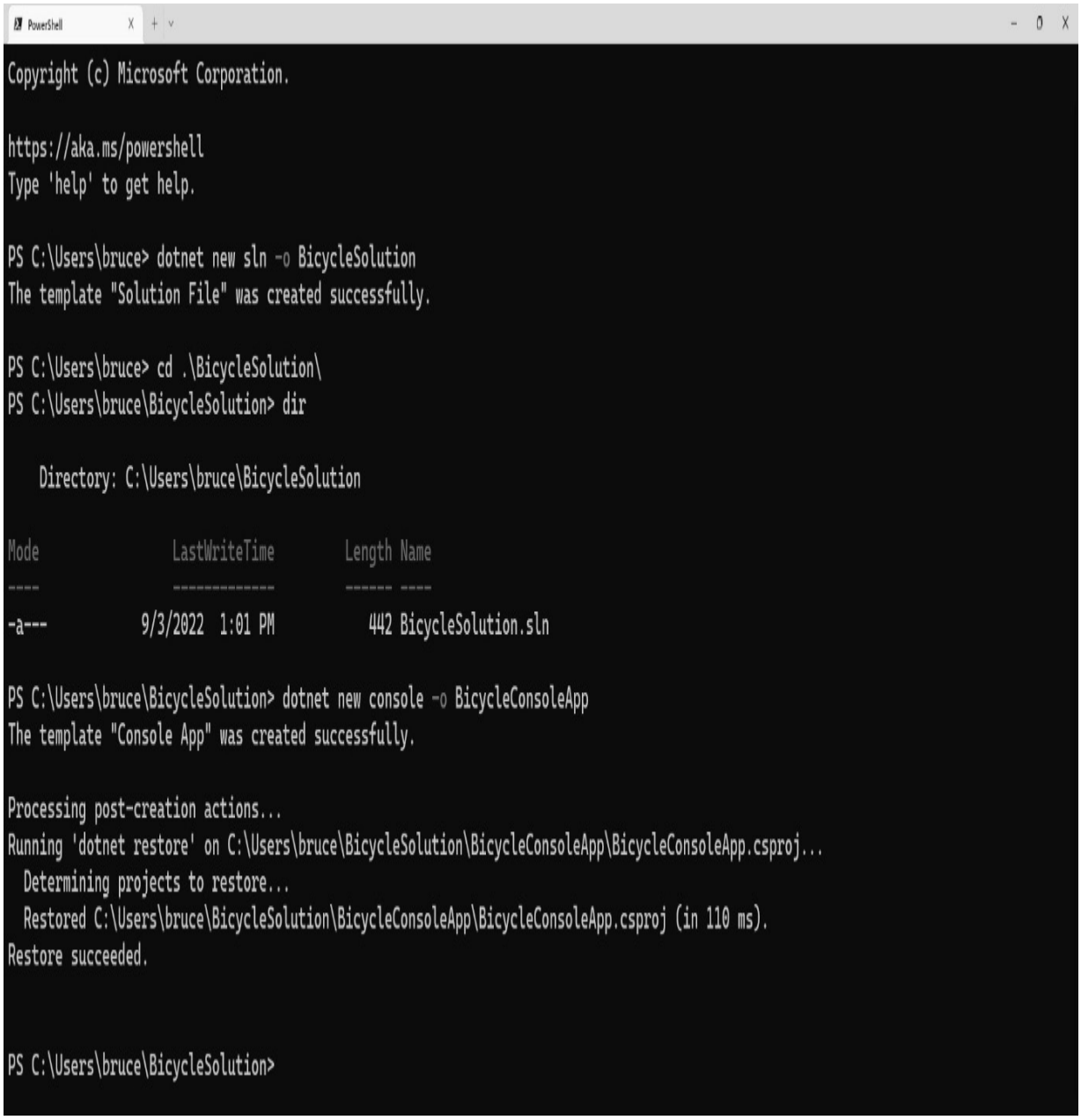


Figure A1.27: Our command generated your file structure for the command-line project.

Once you have the app created, you need to add it to your solution using this command:

```
dotnet sln add BicycleConsoleApp/BicycleConsoleAp
```

The result of the command can be seen in *Figure A1.28*:

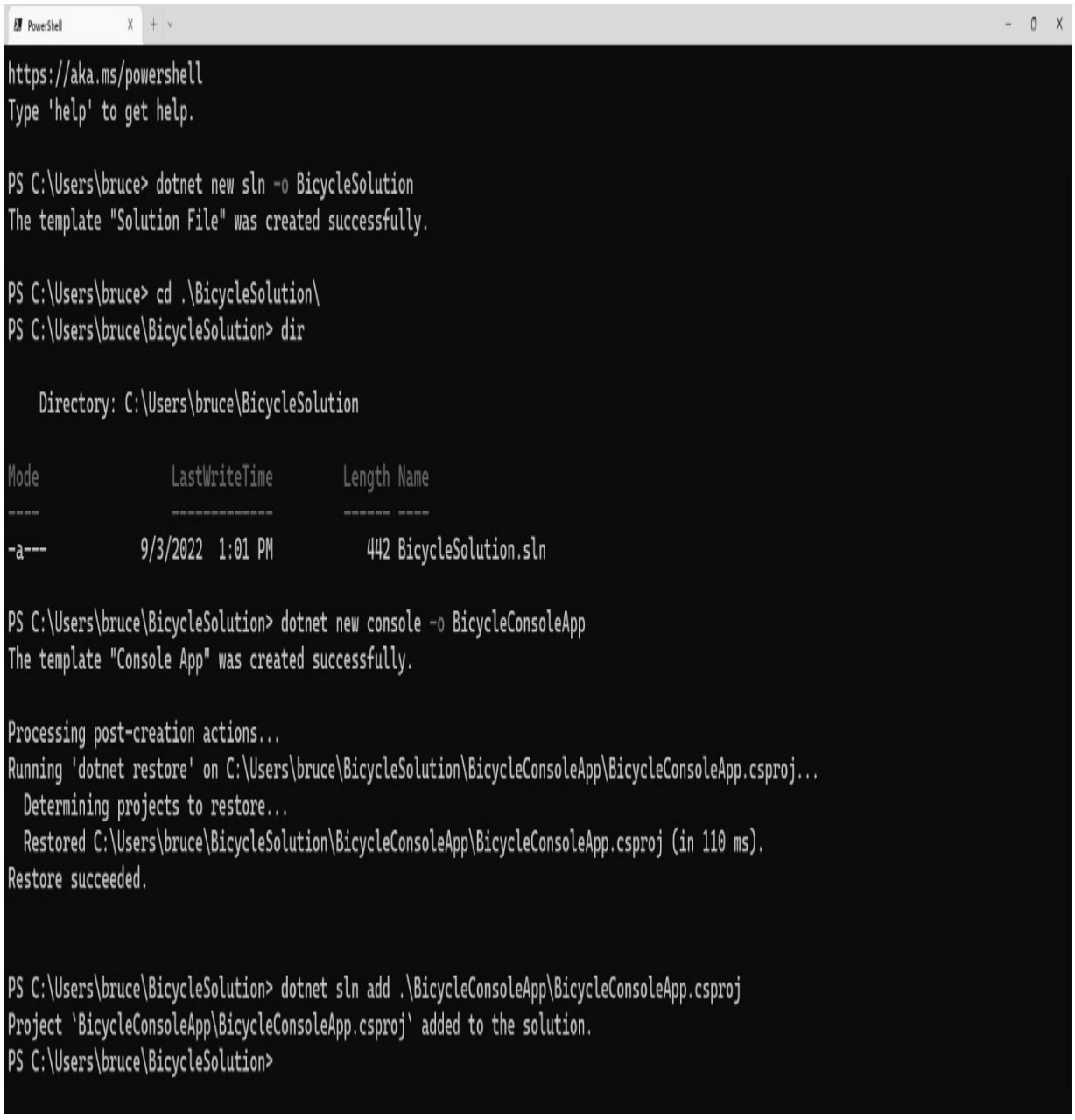


Figure A1.28: The console app was successfully added to the solution.

There won't be a visible change to the folder structure. The command alters the **`BicycleSolution.sln`** file to include the **`BicycleConsoleApp`** project. Your command-line project is ready to use.

Creating a library project

Library projects are used to house reusable code designed to be accessible between projects. In *Chapters 3–5* of this book, I use a library for common classes like those that model the basic parts of the bicycle classes we're building in those chapters. In the real world, you generally put your business logic in a library. When you do this, you can leverage that logic in a web application, a mobile application, and a desktop application without repeating your code in three places.

To create a library project in your solution folder, type the following command into your Terminal window:

```
dotnet new classlib -o BicycleLibrary
```

As before, the command creates a new project within the current solution. The **`-o`** switch tells the command what to name the output, which, in this case, is the name of the library project. Having created

the library project, you need to add the project to your solution using this command:

```
dotnet sln add BicycleLibrary/BicycleLibrary.cspr
```

Use the **`dir`** command to confirm that your folder structure looks like mine in *Figure A1.29*:

Figure A1.29: The project structure after creating the solution, the console app, and the library project.

Now that you have the library project, you need to set up a reference between the console app and the library project.

Linking the library to the console project

To link the console project to the library project, type in this command:

```
dotnet add BicycleConsoleApp/BicycleConsoleApp.cs
```

The library is linked to the console app project. We've spent a lot of time in the Terminal window. It's time to start working in VS Code.

Launching VS Code and adding the C# extension

Adding a class or an interface is easy in VS Code. You can launch VS Code from the Terminal window we've been using by typing in the following:

```
code .
```

You read it correctly: type in **`code`**, add a space and a period, and then press *Enter*. This command launches VS Code with the solution folder loaded. You'll be greeted by your usual security warning, as shown in *Figure A1.30*:

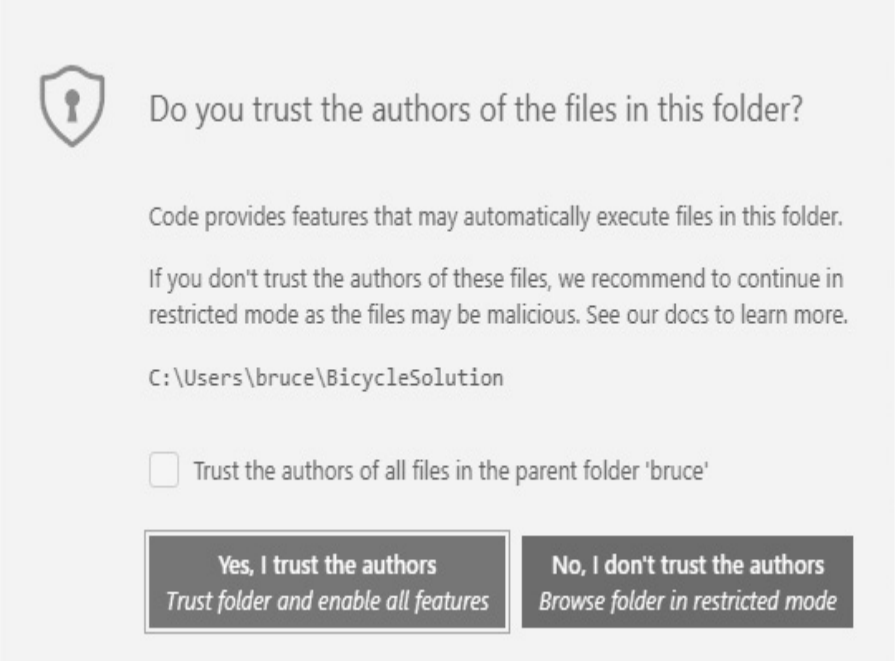


Figure A1.30: Do you trust the files you just made?

Assuming you trust your own work, click on **Yes**. If you don't, briefly close this book and ponder deeply on this. Consider whether it's better to live in a beautiful boring secure world where no harm will ever come to you; a world filled with beautiful flowers whose aroma reminds you of only the best moments of your life. Or would you rather live in a world filled with high adventure and the possibility of total ruin? Pirate ships are safe hiding in port. But they are not meant to stay there! This is your moment! Go for it! Click on the **Yes, I trust the authors** button! I promise that if you do, you'll turn a corner in your life. Grow a spine already and click on the button! Side note: my attorney wants me to remind you the author is not responsible for malware, computer damage, damage to your reputation, or pirate attacks resulting from clicking on this button.

With your personal issues resolved, if you had them, let's continue.

I have mentioned that VS Code isn't considered a full-fledged IDE. It is meant as a general-purpose coding tool and doesn't assume anything about how you will use it. As such, VS Code doesn't come with C# support baked in. Naturally, Microsoft has a plugin to help you with C#. You'll probably be prompted to install the plugin as soon as the project opens. In case you aren't, you can install it by following the steps shown in *Figure A1.30*:

Figure A1.30: Microsoft makes a free extension for VS Code that makes working with C# more enjoyable.

First, click on the **Extensions** button in the menu on the left-hand side of the interface (1). Then, search for this extension in the search

bar (2):

```
@id:ms-dotnettools.csharp
```

This will narrow the list to a single extension. Click on it, and then click on the install button. VS Code is now fully aware of the C# language and the structure of your project.

Adding a class or interface

Right-click on the **`BicycleConsoleApp`** folder in the explorer view and click on **New File**, as shown in *Figure A1.31*:

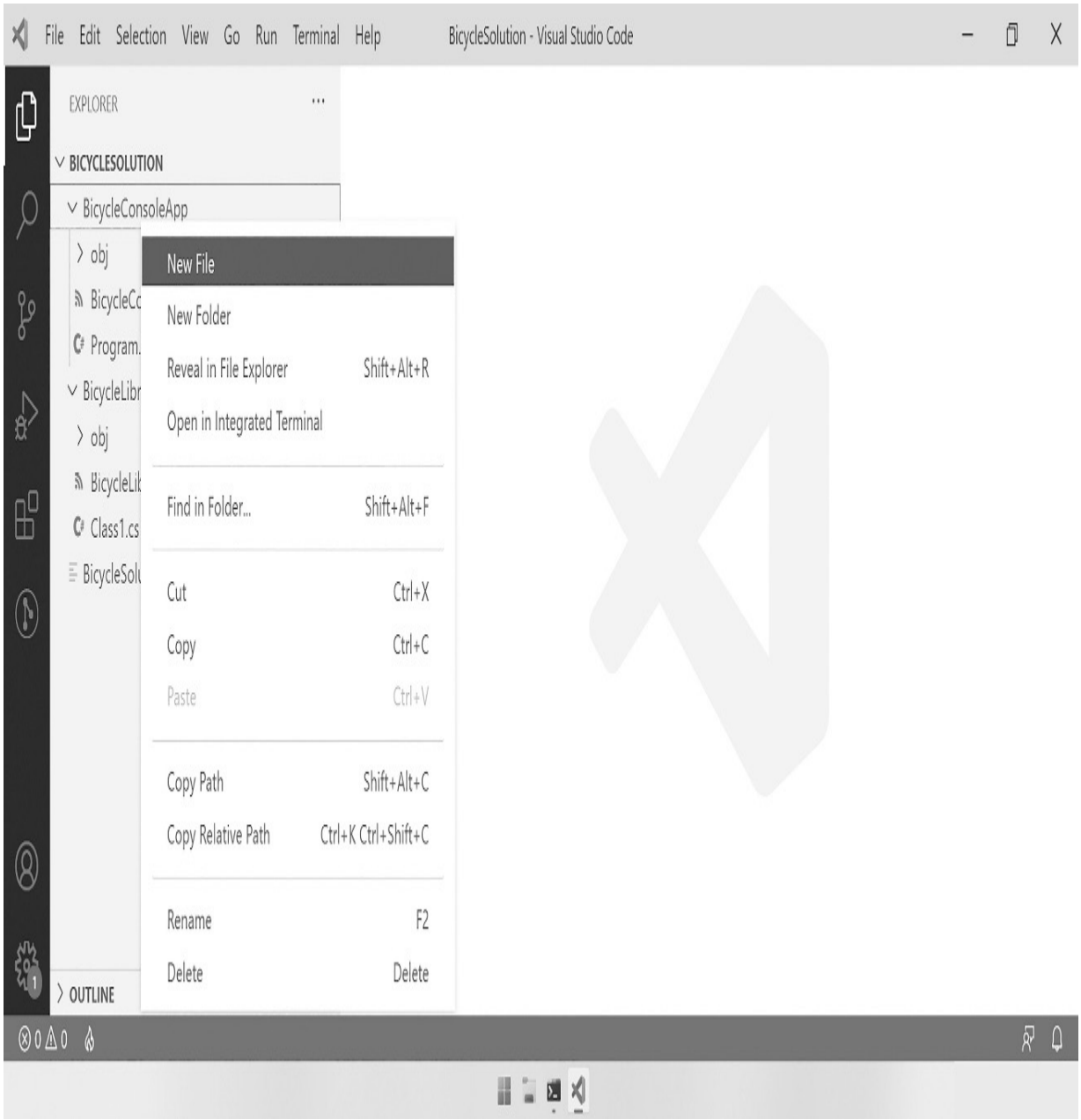


Figure A1.31: Right-click on the explorer area, and click on New File to create a new file.

Once you click on **New File**, just type in the name of the file you want to create. Be sure to include the `.cs` extension. For example, I'm call-

ing my file **`Class1.cs`**. If you're creating an interface, it is standard practice to begin the filename with a capital I (as in "Interface").

Adding code to the BicycleLibrary project

Let's add a method to the **`BicycleLibrary`** project so that we can verify it is linked and working correctly.

Find the **`Class1.cs`** file in the **`BicycleLibrary`** project. This file was generated for you when you created the project. Click on the file, and it will open in the editor. Add the following code:

```
namespace BicycleLibrary;

public class Class1
{
  public string SayHello()
    {
        return "Hello from the Bicycle Library!";
    }
}
```

Be sure to save the file! Compare your project with mine, as shown in *Figure A1.32*:

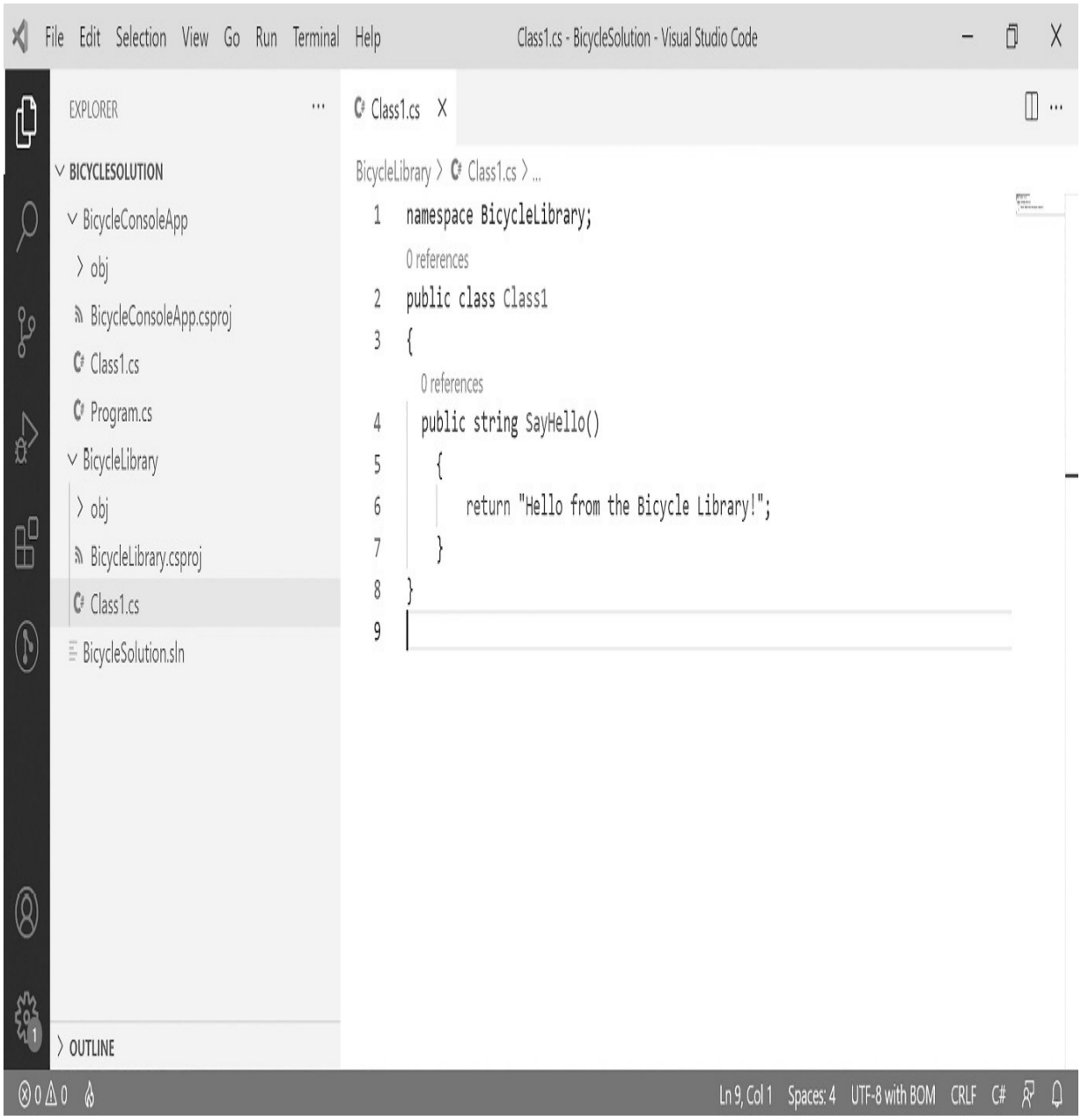


Figure S1.32: The library code has been added to Class1.cs in the BicycleLibrary project.

Next, let's switch to the **`Program.cs`** file in the **`BicycleConsoleApp`** project. Replace the line **`dotnet`** generated with the following:

```
using System;
using BicycleLibrary;
var test = new Class1();
Console.WriteLine(test.SayHello());
```

As you type, IntelliSense will show you the library method we added to the **`BicycleLibrary`** project as if it was within the **`BicycleConsoleApp`** project. Check your work against *Figure A1.33*:

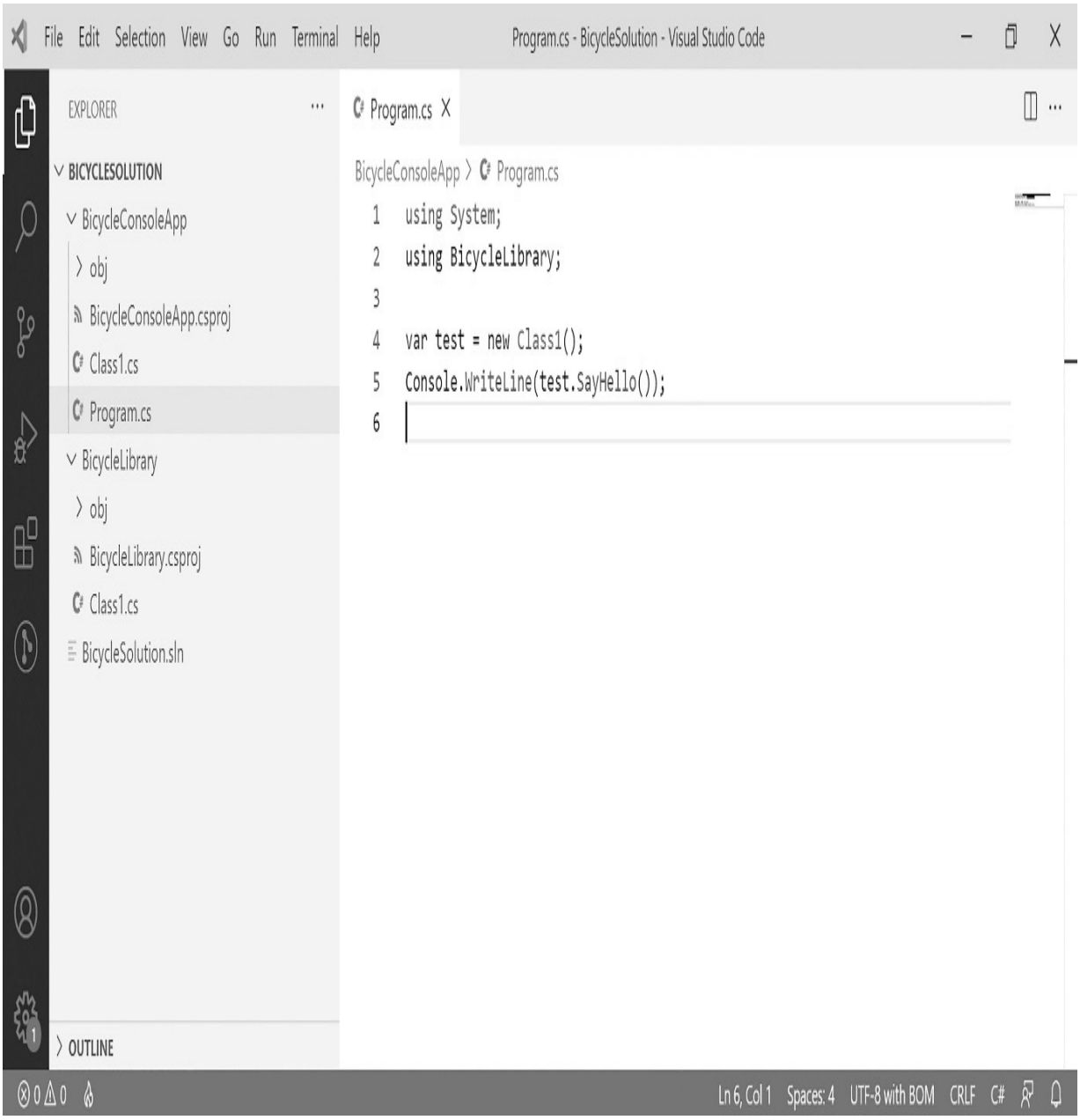


Figure A1.33: The updated Program.cs file with our test code.

Our mission has been accomplished! There is just one thing left to do: build the project.

Building and running the console project

We could switch back to our Terminal window, but VS Code has an integrated Terminal window. Using it is more convenient than constantly switching windows.

To activate it, use the keyboard shortcut *Ctrl* + `(control + backtick). Additionally, you can click on **View** and then **Terminal** from the main menu, as shown in *Figure A1.34*:

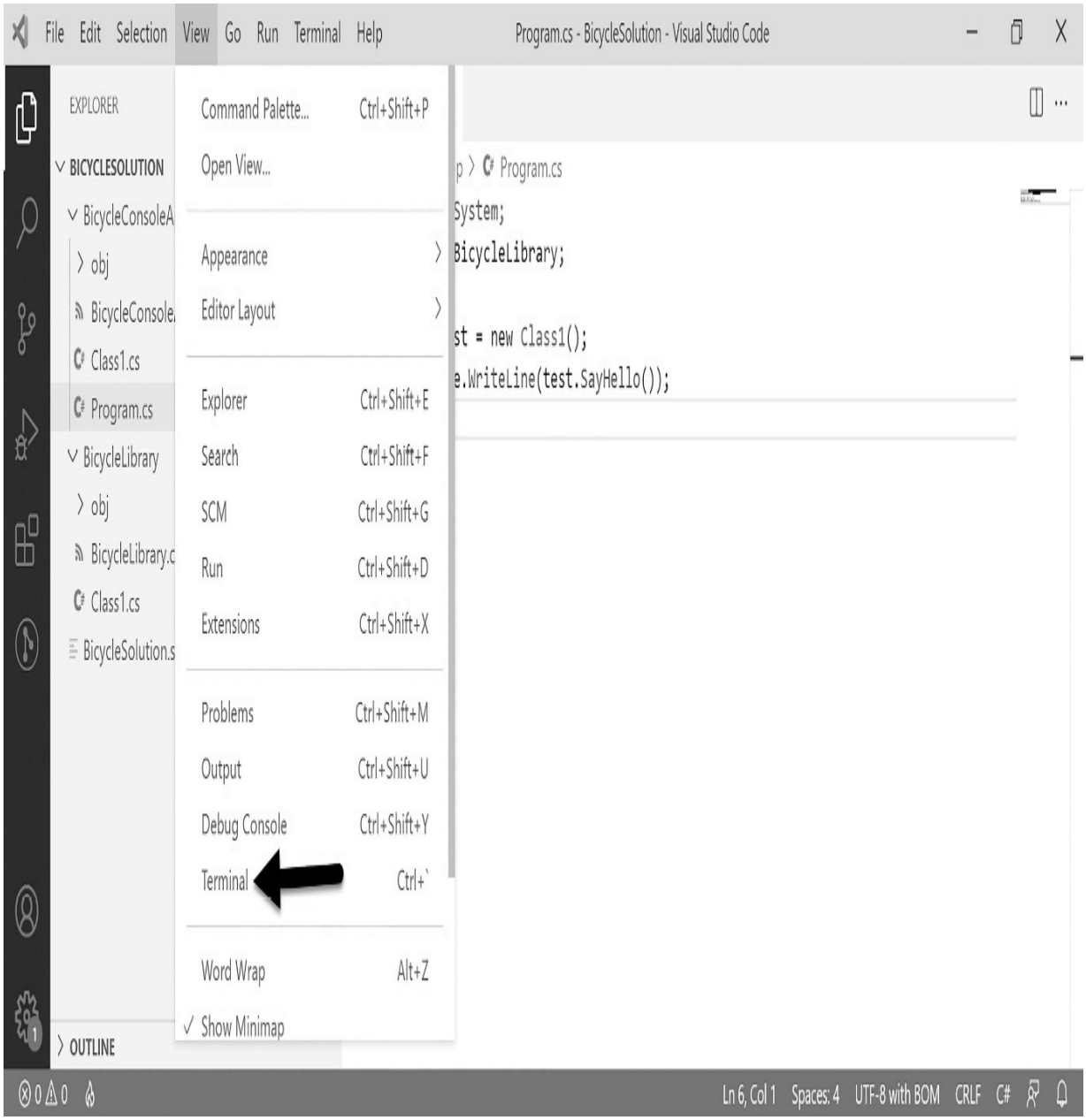


Figure A1.34: You can open the integrated Terminal window in VS Code using this menu item, or Ctrl + ` as a keyboard shortcut.

Running the project will build the project and then execute the **`BicycleConsoleApp`** project's executable file, which consists of the

code in the **`Program.cs`** file. To run the program in the Terminal window, enter this command:

```
dotnet run —-project .\BicycleConsoleApp\BicycleC
```

You'll see the program being built, and the command-line project will run. You should see the "Hello from the Bicycle Library!" message in the Terminal window just like you see it in *Figure A1.35*:

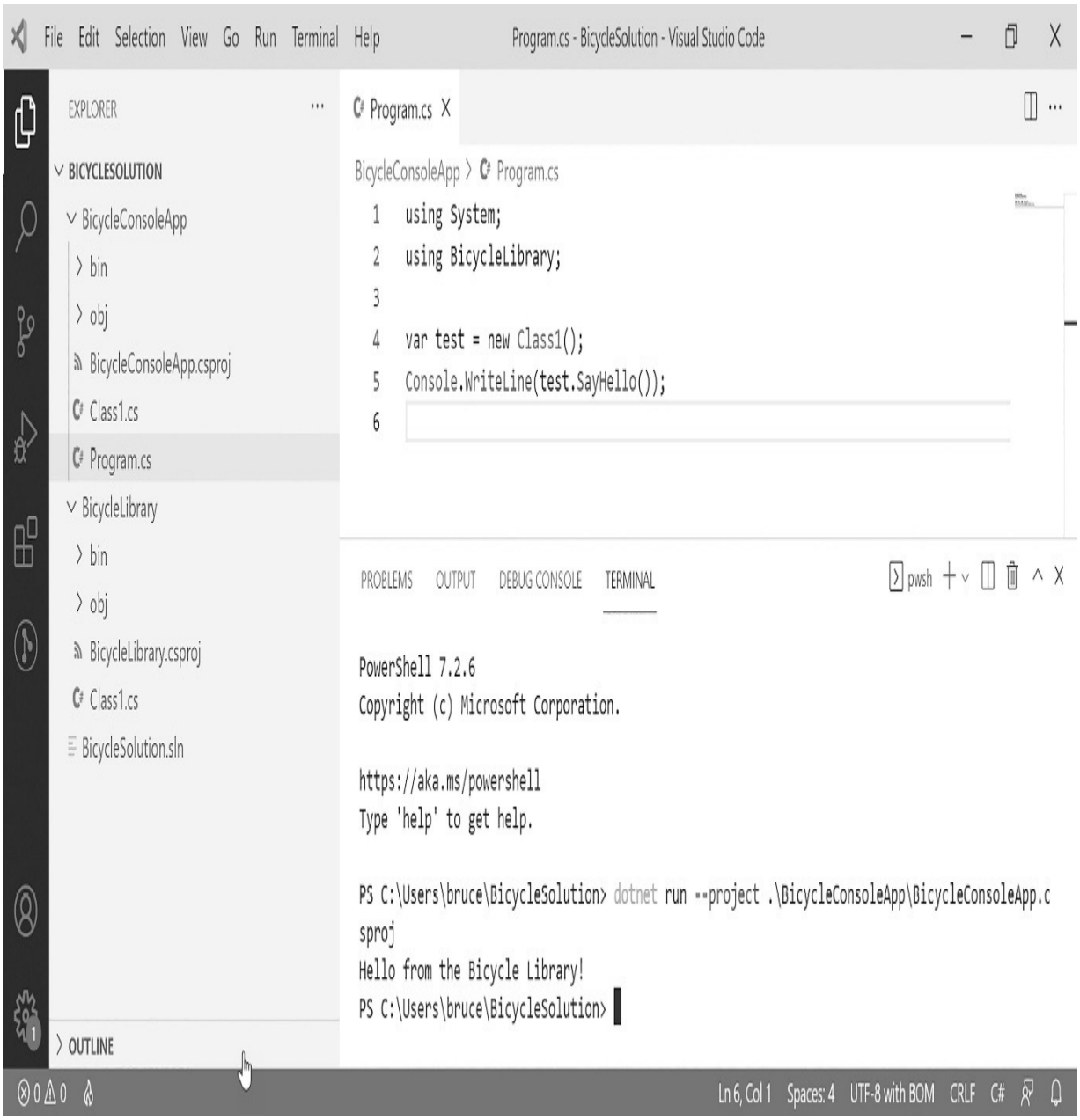


Figure A1.35: This is the result of our test run.

This concludes our brief tour of VS Code. If you're interested in a video walk-through that coaches you through setting up a VS Code development environment in Windows, you'll find links for this at https://csharppatterns.dev.

Rider

JetBrains Rider is the most recent addition to the world of C# IDEs. JetBrains is known for making amazing IDEs for Java (*IntelliJ Idea*), Python (*Pycharm*), PHP (*PHPStorm*), and JavaScript (*WebStorm*) among others. They also built Google's IDE for Android Development, *Android Studio*. In addition to IDEs, the company has a very popular add-in product for Visual Studio called *Resharper*. The tools in *Resharper* give you many similar features you'll find in *Visual Studio Enterprise* at a fraction of the cost.

I used Rider to create this book, mainly because of the code formatting and refactoring tools available in the program. Also, I use several of their other IDEs regularly, and I have my keyboard shortcuts configured to be common in every language IDE I use. This is purely a personal preference. I knew I could produce this book more quickly with Rider.

Rider is not a free product. There is a lower-cost version available for solo developers, and the same product is sold at a higher rate for use in companies. Check out https://www.jetbrains.com/rider for more information if you're interested in using this tool.

Since I used Rider to create the book, I think it would be strange if I didn't at least show you around. Let's do this exercise one more time

using Rider.

Creating a command-line project

When you launch Rider, you are greeted with the **Welcome** screen dialog. Click on the **New Project** button in the upper-right portion of the dialog box, as shown in *Figure A1.36*:

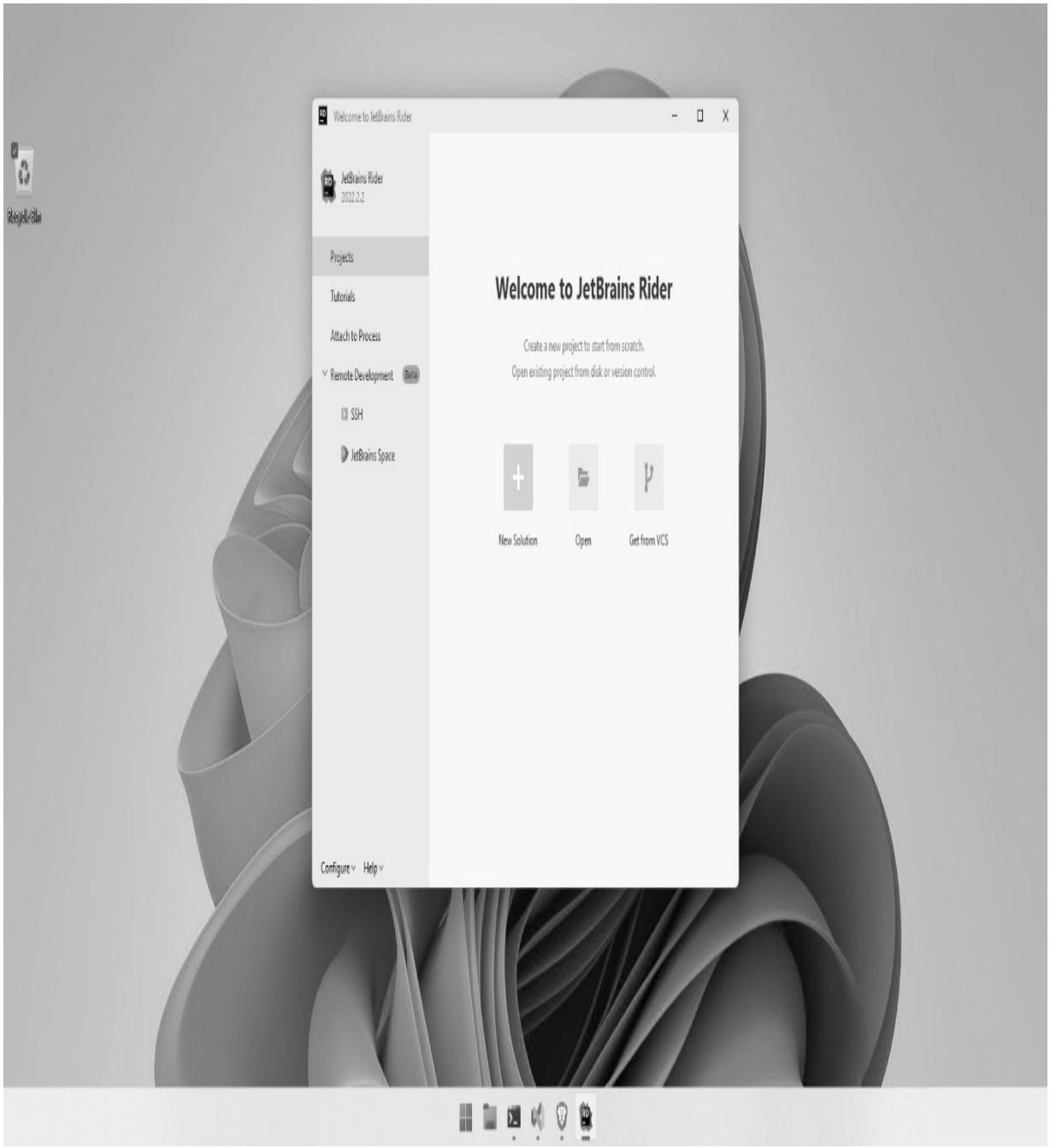


Figure A1.36: The welcoming screen in Rider with the New Project button highlighted.

Clicking on the button gives you another dialog, as shown in *Figure A1.37*:

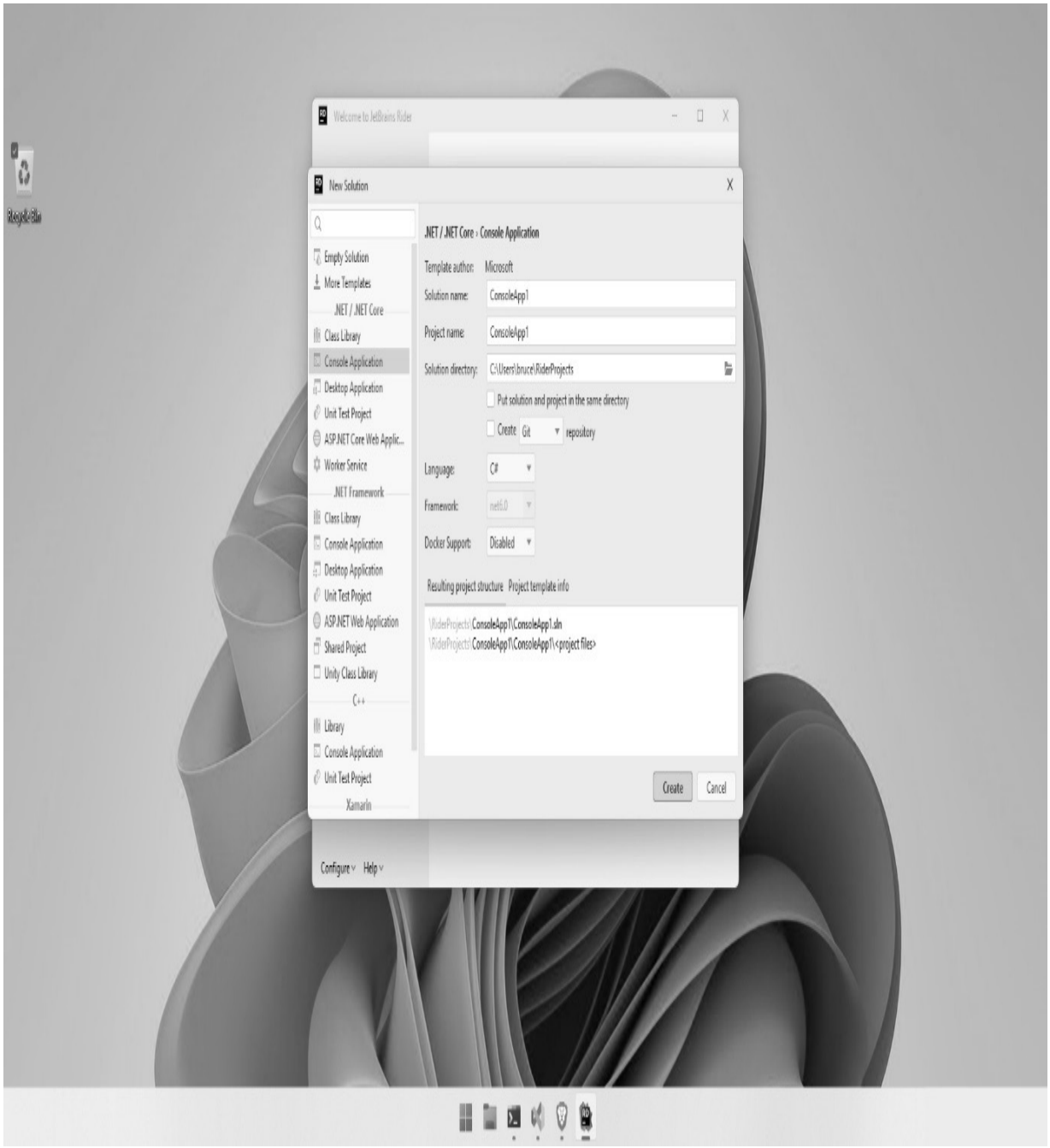


Figure A1.37: Project creation in Rider.

With Rider, everything is on one screen. Find the **Console Application** template in the list on the left-hand side and click on it. Set the solution name to **`BicycleSolution`** and **`ProjectName`** to

`BicycleConsoleApp`. Click on the **Create** button and your project will be generated. The full IDE will appear, and you'll find your project hierarchy in the explorer panel on the left-hand side of the IDE window.

Adding a class or interface

Adding a class is the same as it is in the other two IDEs we've talked about so far. Right-click on the **`BicycleSolution`** project. A context menu appears, as shown in *Figure A1.38:*

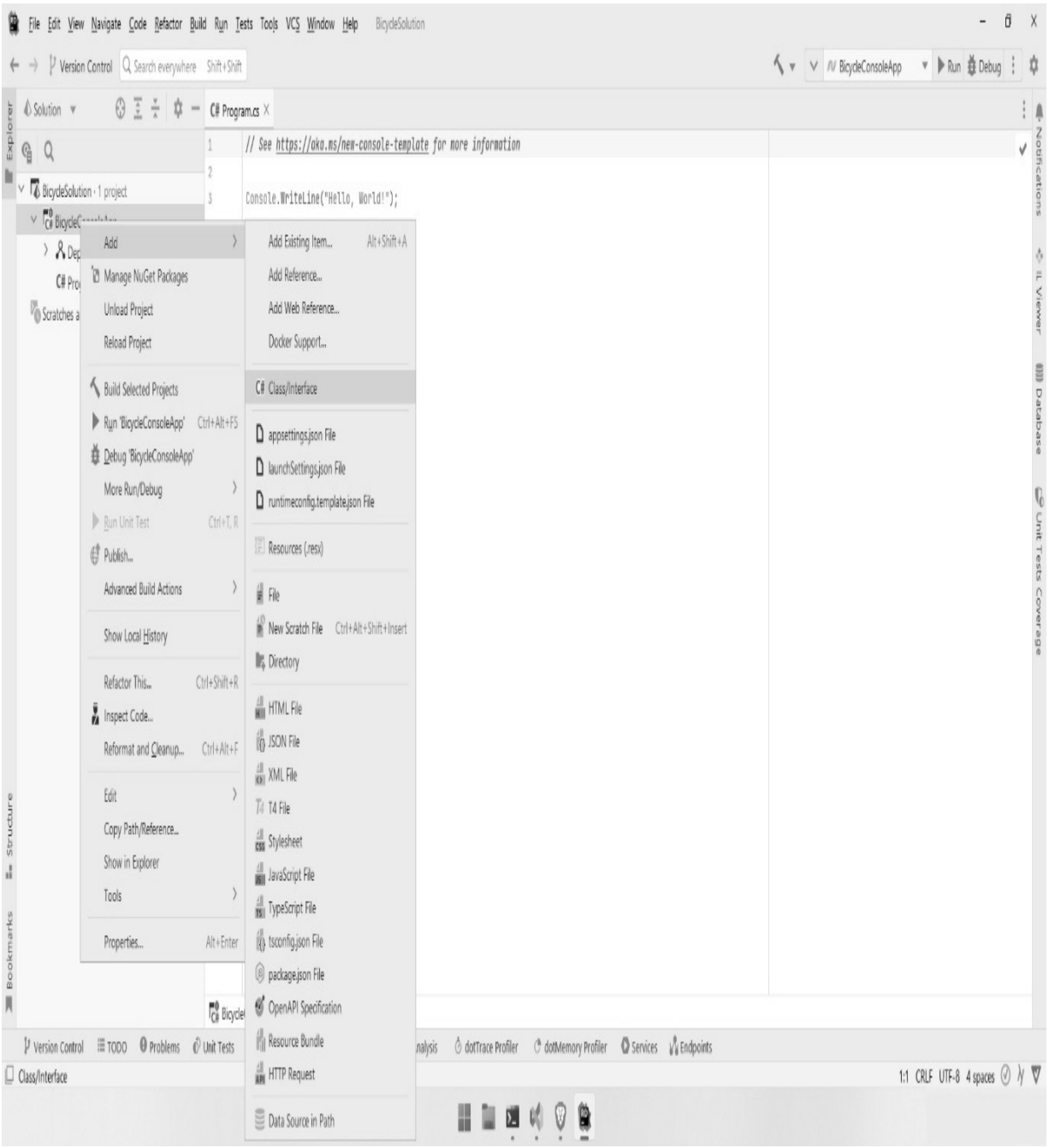


Figure A1.38: The ability to add a class or interface to your project can be found in the context menu when you right-click on the project.

Hover over the **Add** menu item, and then find **Class/Interface** in the sub-menu. Click on it. You will see a small dialog asking you to name the class or interface. There are a few more possibilities evident in the list, but this book is only concerned with classes and interfaces. Choose the **Class** option and name your class whatever you would like. As soon as you press *Enter*, you'll see the class file added to your project.

When you need an interface, follow the same procedure, but instead of **Class**, select **Interface**. Don't forget that in C#, the name convention for interfaces stipulates naming the interface starting with the letter I (as in "Interface").

Next, we'll look at adding a library project to our solution so that we can maximize code reuse between projects.

Creating a library project

Library projects are used to house reusable code designed to be accessible between projects. In *Chapters 3–5* of this book, I use a library for common classes like those that model the basic parts of the bicycle classes we're building in those chapters. In the real world, you generally put your business logic in a library. When you do this, you can leverage that logic in a web application, a mobile application, and a desktop application without repeating your code in three places.

Right-click on the **`BicycleSolution`** in the explorer menu. Hover over the **Add** option and select **New Project...**. You can see this in *Figure A1.39*:

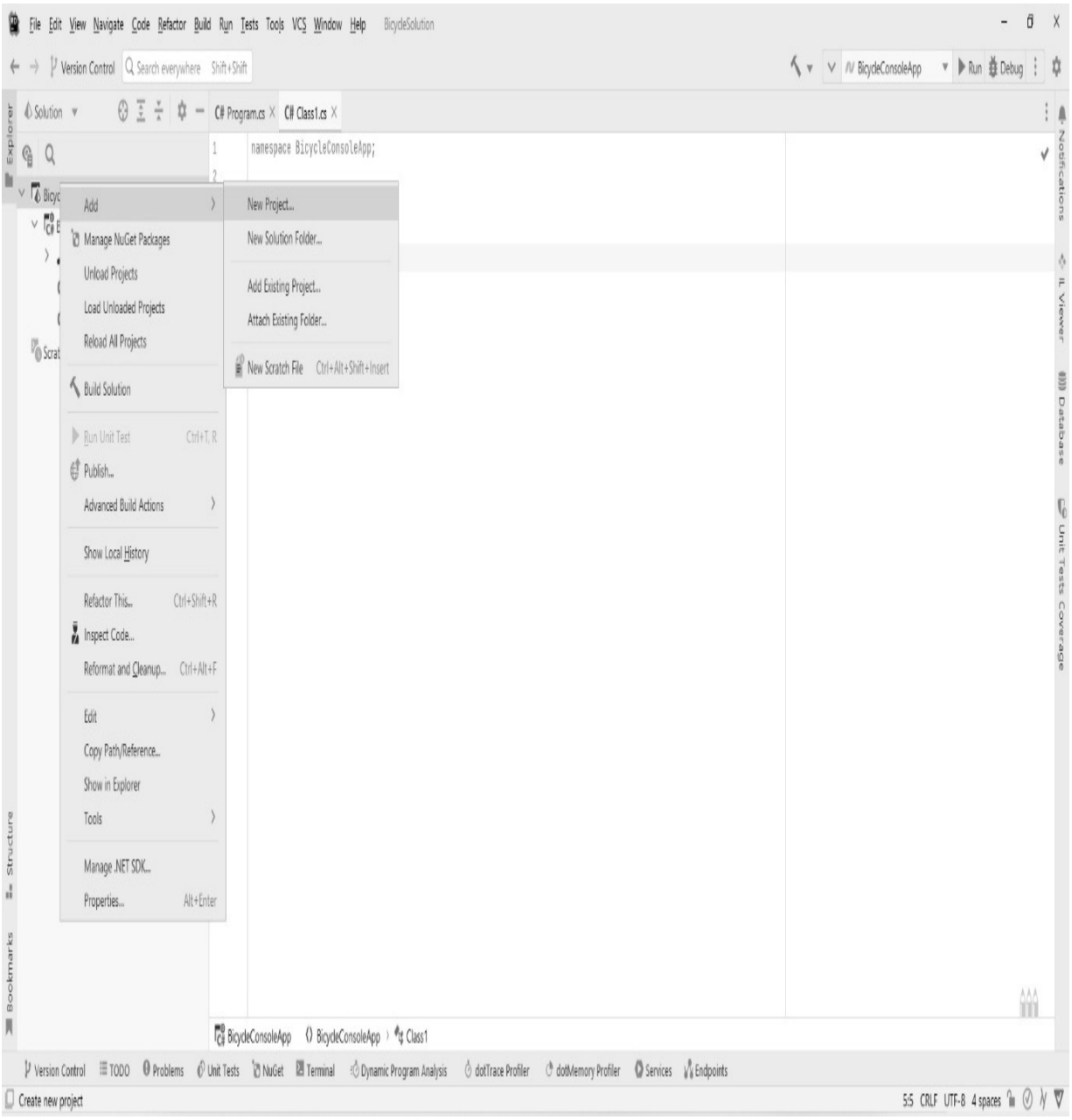


Figure A1.39: Adding a project to the solution in Rider is done by right-clicking on the solution, then hovering over Add, and

then clicking on New Project.

You get the same project dialog we started with. Locate the **Class Library** template and click on it. Then, fill in the name for the project as **`BicycleLibrary`**. Your project should match mine, as shown in *Figure A1.40*:

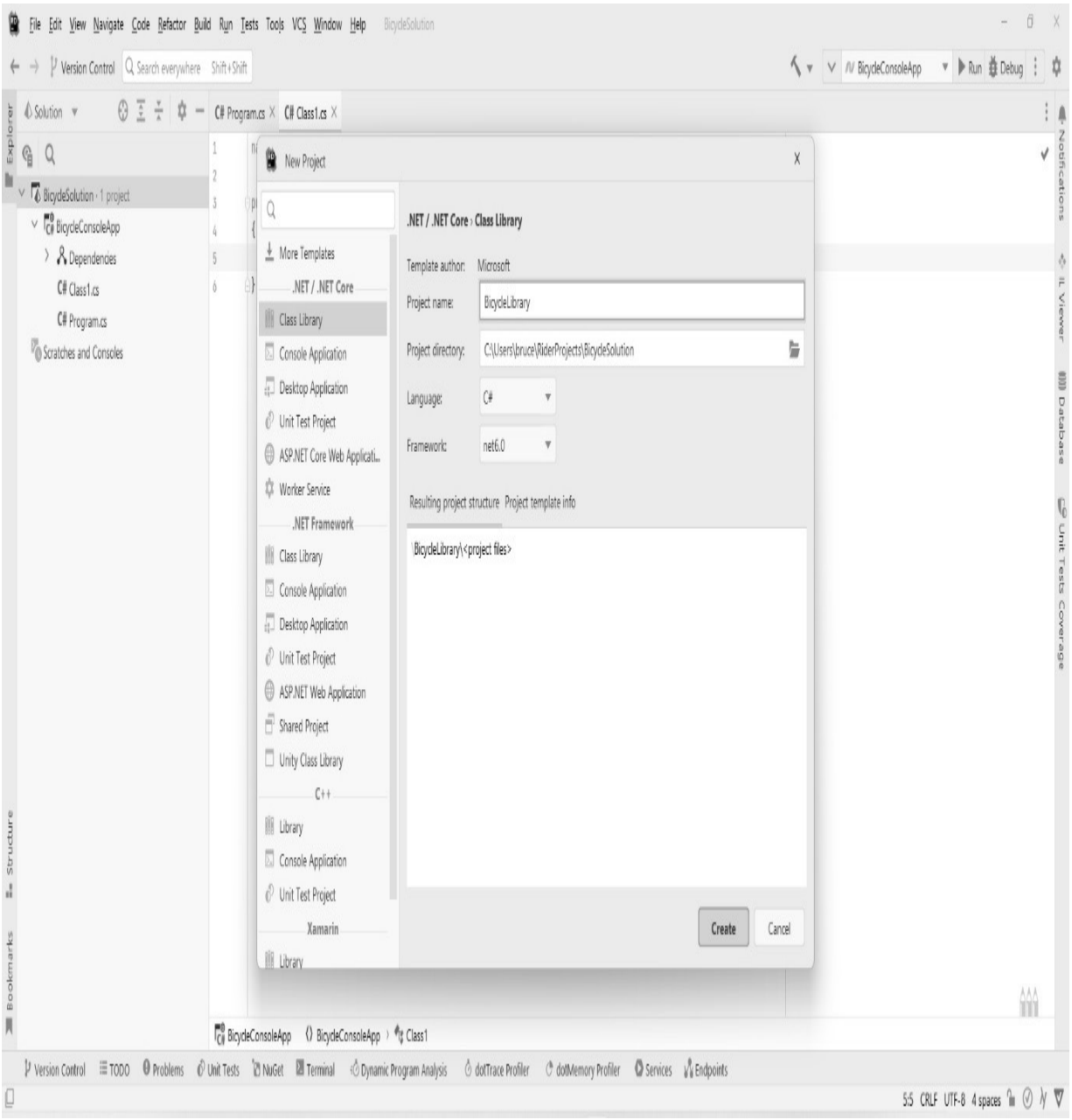


Figure A1.40: Set up your new library project as shown.

Click on the **Create** button, and you'll find a new project added to the solution. Your explorer pane should look something like *Figure A1.41*:

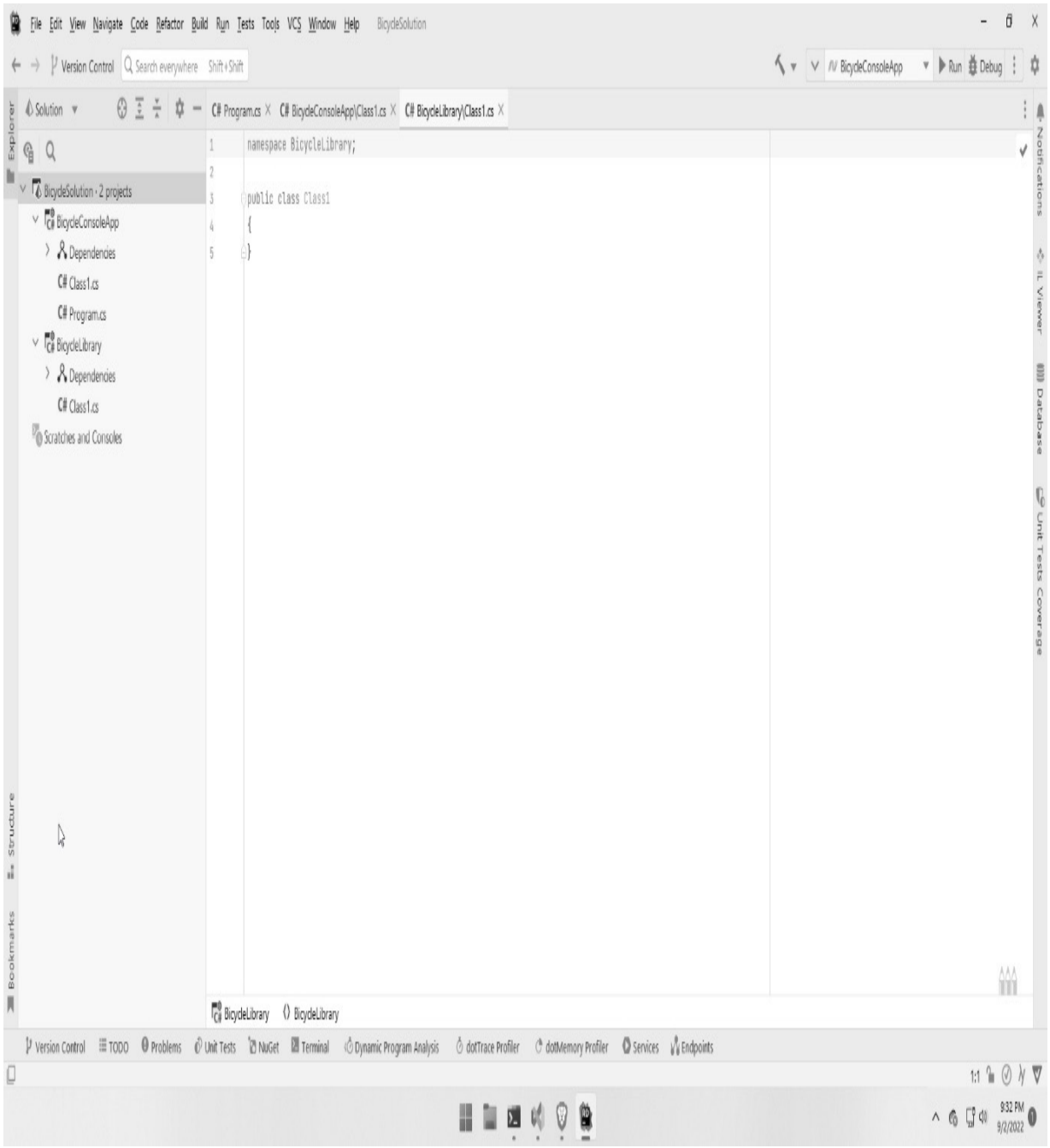


Figure A1.41: The explorer pane shows both projects in our solution.

Before we can use the library in our console app project, we need to link the two projects with a reference.

Linking the library project to the console project

In the explorer pane, locate the **`BicycleConsoleApp`** project and right-click on the **`Dependencies`** item in the project hierarchy. A context menu appears, as shown in *Figure A1.42*:

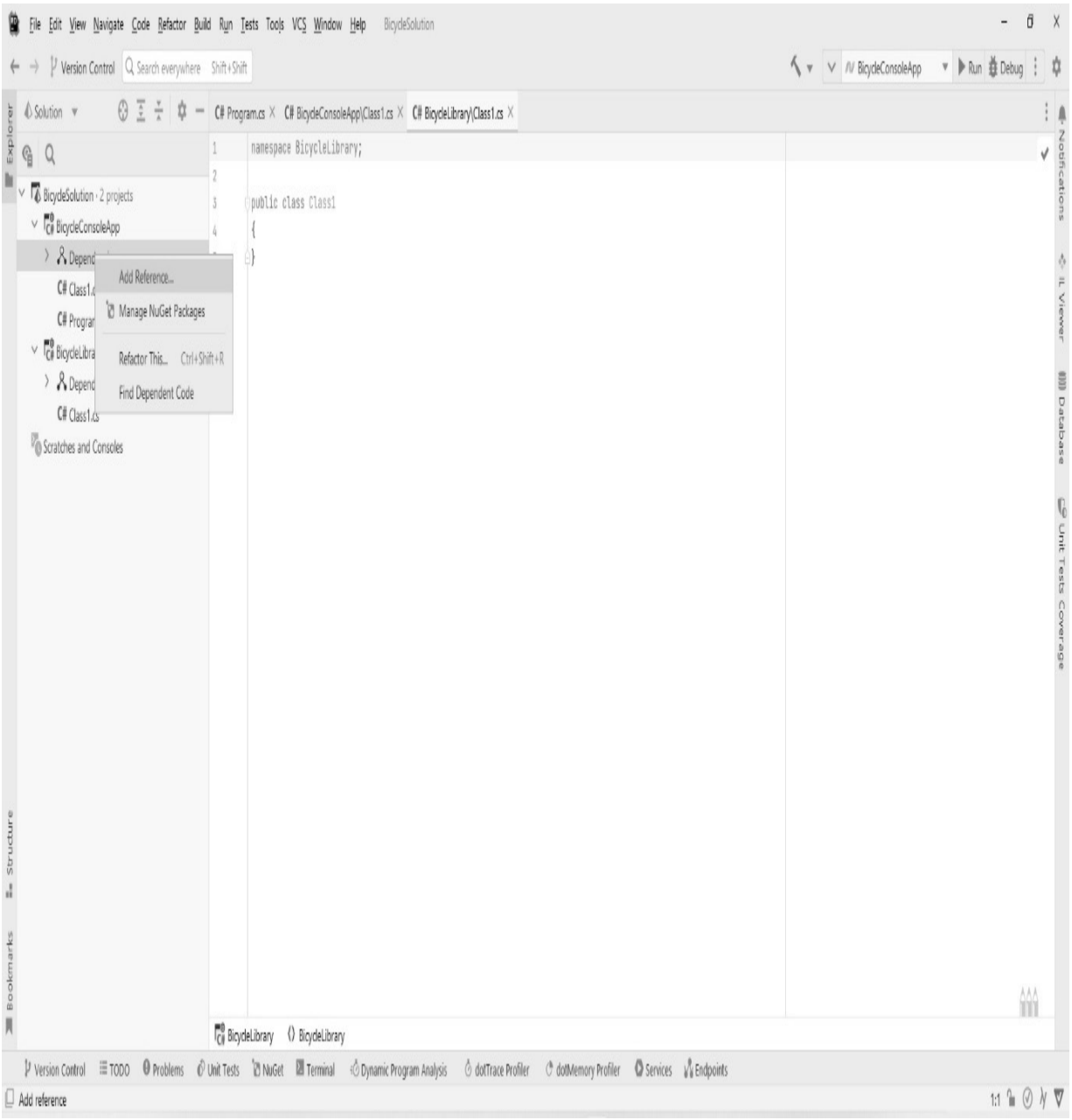


Figure A1.42: The context menu is displayed when you right-click on the Dependencies item in the explorer pane.

Click on **Add Reference**. A dialog appears like the one in *Figure A1.43*:

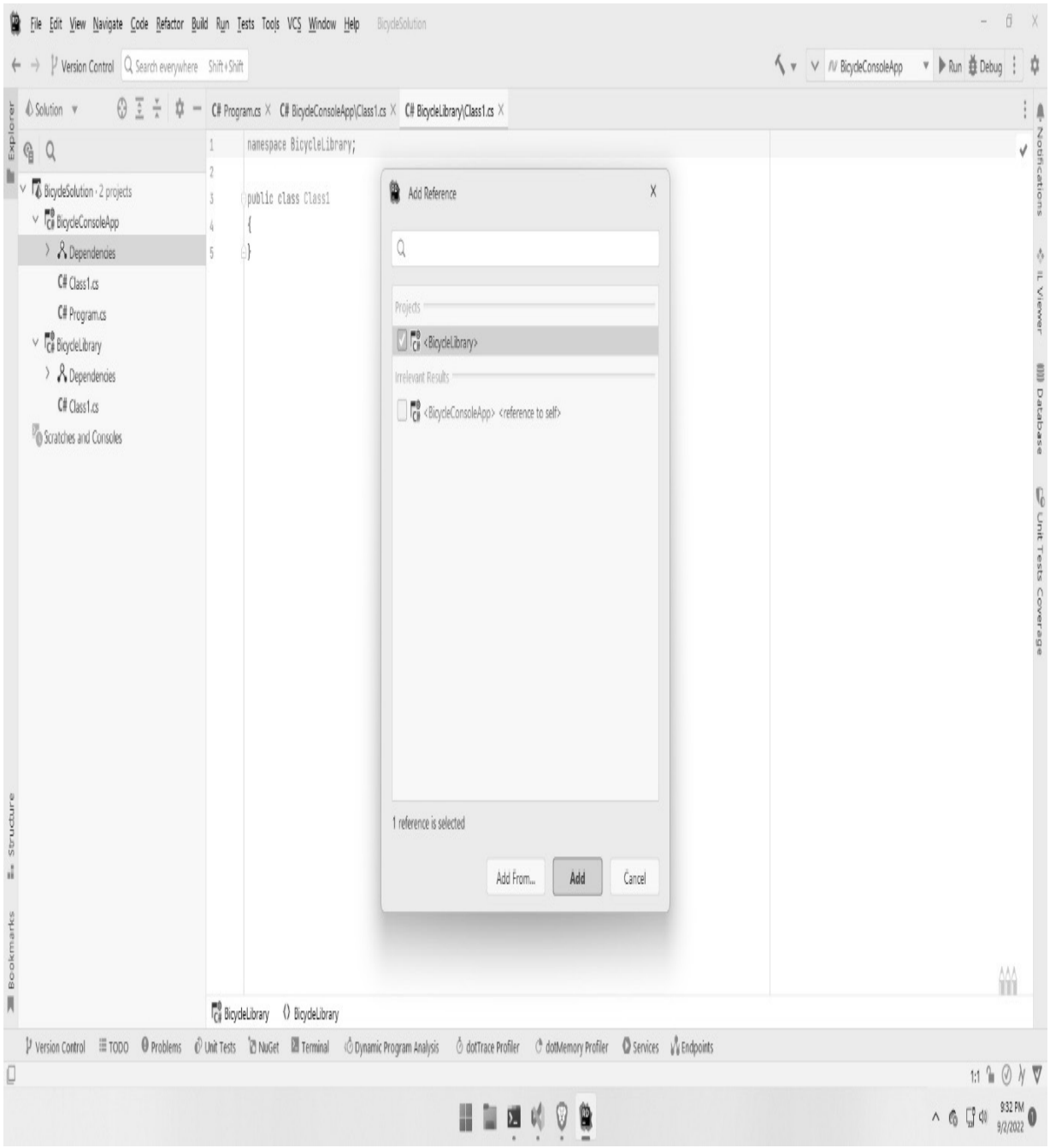


Figure A1.43: The Add Reference dialog in Rider.

You should see the library project listed. Check the box next to the library, and then click on **Add**. You can now reference the library project from your console app project. Let's try it out.

Testing the linked library

Open the **`Class1.cs`** file in the **`BicycleLibrary`** project. Add this code:

```
namespace BicycleLibrary;
public class Class1
{
  public string SayHello()
    {
        return "Hello from the Bicycle Library!";
    }
}
```

That's the third time I've typed in the same code. Thank goodness the English language doesn't require authors to be DRY.

Next, open the **`Program1.cs`** file in the **`BicycleConsoleApp`** project. Change the code in that file to this:

```
using System;
using BicycleLibrary;
var test = new Class1();
Console.WriteLine(test.SayHello());
```

As you type the code, you should see the code hints showing you that you have access to the library. Once the code has been entered, we're ready to run a test.

Building and running the console project

As with Visual Studio, you're looking for a big green arrow that indicates the **Run** button in the toolbar. You can see this button highlighted, alas, not in color, in *Figure A1.44*.

When you click on the **Run** button, your project builds and runs with an attached debugger just as it does in Visual Studio. Unlike Visual Studio, your program runs within Rider's integrated Terminal just like it did in VS Code. You can see the result of my run in *Figure A1.44*:

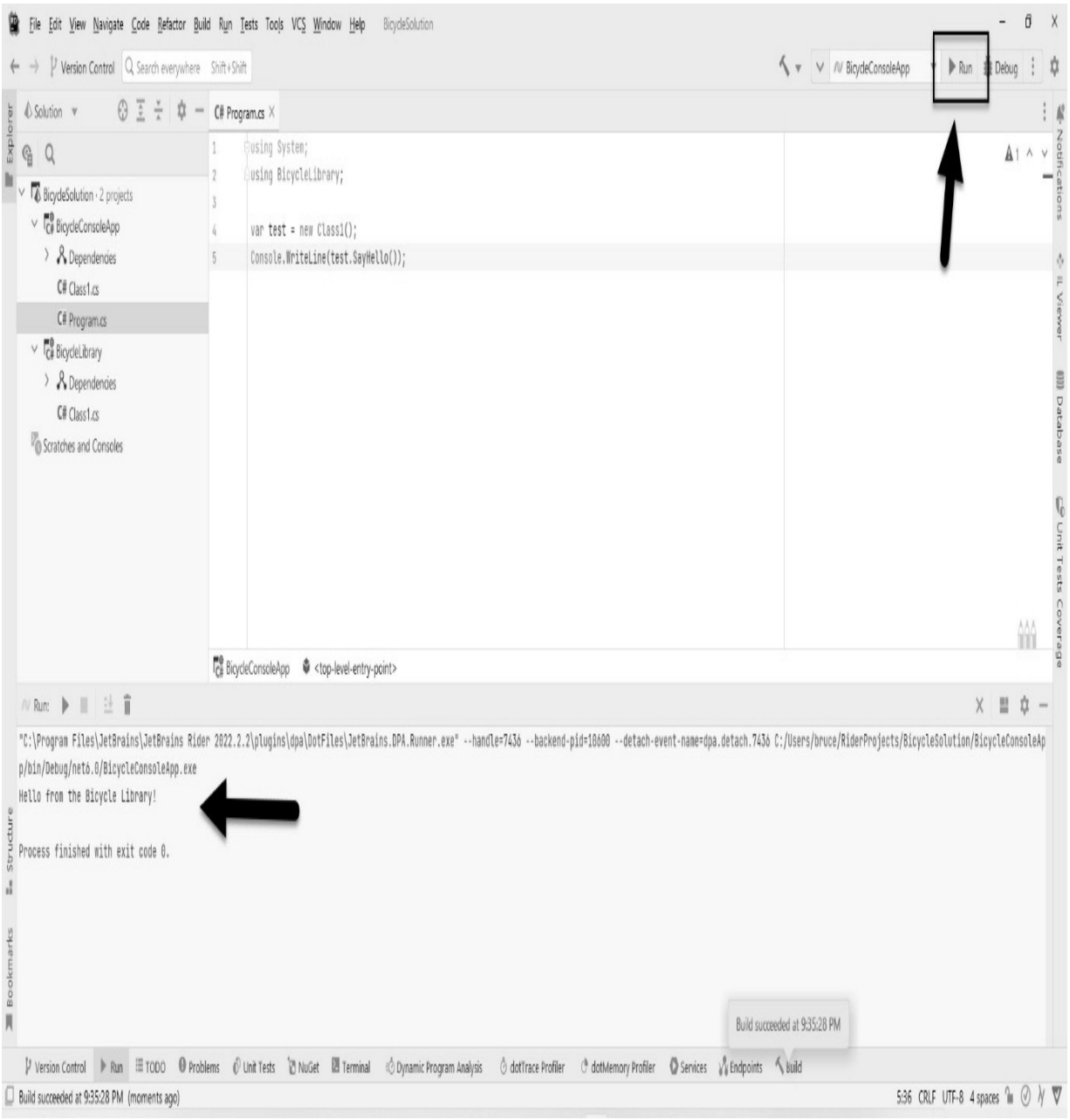


Figure A1.44: The completed run of our test program in Rider; the Terminal window has been pointed out.

This concludes our brief tour of *Rider*. If you're interested in a video walk-through that coaches you through setting up a *Rider* develop-

ment environment in Windows, you'll find links at https://csharppatterns.dev.

Summary

This appendix was meant to be a short summary of working with C#. I almost didn't include it in the book. There are many other places you can get training and orientation for the language. I felt, though, having taught C# for decades that I might be able to more concisely, yet thoroughly, get you up to speed if you're coming from a different language, have limited experience with C#, or if it's been a while and you're a little rusty.

Along the way, we learned the following:

- C# is a standardized general-purpose, strictly OOP language.
- C# uses a strong, static type system.
- C# is designed with features designed to limit the most common problems found in C and C++ development such as bounds checking, effortless memory allocation, and automated garbage collection.
- C# supports many different kinds of numeric types, including signed and unsigned variants.
- How to make auto-implemented properties and methods and work with encapsulation.

- How to use the basic building blocks of OOP in C#, including inheritance and working with interfaces.

At the end of all that we were treated to a basic tutorial on the three most popular IDE choices for C#: Visual Studio, VS Code, and Rider. I called this appendix concise. I wouldn't call it short, but consider the size of most C# books that only orient you in Visual Studio, and I think you'll agree this is a bargain. If you have found yourself wanting more, head over to https://csharppatterns.dev where I have links to additional resources.

Further reading

- *C# Object-Oriented Programming for Beginners in C# and .NET* by Praveenkumar Bouna.
- *Visual Studio Code for C# Developers* by Praveenkumar Bouna.
- *Hands-On Visual Studio 2022* by Hector Uriel Perez Rojas and Miguel Angel Teheran Garcia.
- https://csharppatterns.dev has links to additional resources.

Appendix 2: A Primer on the Unified Modeling Language (UML)

You don't have to look hard to realize that designing software is a lot like designing anything else. In *Chapter 1, There's a Big Ball of Mud on Your Plate of Spaghetti*, we talked about the underpinning of software patterns coming from a pioneer in the field of architecture – not software architecture, but the traditions, engineering, and design practices involved in the architecture of buildings and cities. In 1977, Christopher Alexander documented a pattern language designed to form the basis for the best practices for building towns. His book described 253 patterns that were presented as the paragon of architectural design. The book broke everything down into objects.

Object-Oriented Analysis and Design (**OOAD**), a practice that is adjunctive to **Object-Oriented Programming** (**OOP**), involves the design of an object structure separate from the exercise of writing code. This is usually the job of a software architect or a senior developer. Software architects are similar to architects of buildings: they design the structure of an application. This can be done before the implementation team, which is the team that is going to build what the architect designs, picks the programming language for a project.

OOAD makes use of a set of documentation conventions codified as the **Unified Modeling Language** (**UML**). It sounds like a programming language, but it isn't. Instead, it is a standard for creating diagrams that explains the structure and relationships between components in a software system.

Think of it this way. If you are a musical composer, you can write your notes in a musical score. You can do this without ever touching an instrument. If you are skillful in composition, you can write entire orchestral works on paper using sheet music notation. You can then hand your sheet music to an orchestra and a conductor, and assuming the orchestra comprises competent musicians, they should be able to play your music. Architects are like composers. Programmers are musicians, and the team lead or lead developer is the conductor.

Throughout this book, UML class diagrams are used to convey the structure of the pattern code to be implemented in our real-world projects. If you've never worked with the UML before, you're going to need a short primer. This appendix is designed to be that primer. It isn't a substitute for a graduate-level OOAD course, or even for reading a whole book on UML. I'm just going to cover the diagram conventions for the one type of diagram we use throughout the book so that you can understand the diagrams, and how they are converted to code.

Again, UML isn't a coding language; it's a specification for diagramming your code. There are 14 types of diagram in the UML 2.5 specification. We only need one; the most common among the different diagrams is the *class diagram*.

Technical requirements

There are many tools for drawing UML diagrams, and although I have presented many such diagrams, I've taken the tool used to make them for granted. In the real world, this exercise often happens on a whiteboard. A whiteboard is fine for ephemeral drawings that get erased later. For this book, my diagrams need to be a bit more permanent, so here's what I'm using:

- A computer running the **Windows** operating system. I'm using **Windows** 10. Honestly, this doesn't matter, since diagramming tools are plentiful for all operating systems, and there are many that will work in your browser.
- A diagramming tool. I'm using **Microsoft Visio**.

There are quite a few UML tools on the market. Here is a short list of tools I've used over the years:

- **Microsoft Visio**
- **StarUML**

- **Altova UModel**
- **Dia**
- **Umbrello**
- **Umlet**
- **Omnigraffle** (**Mac** only)

There are many more on the web. I tend to prefer apps that run on my computer versus browser-based solutions.

The structure of a class diagram

Class diagrams consist of several types of structures, and a set of connecting lines that express the relationship between the structures. The structures are as follows:

- A class (obviously)
- An interface
- An enumeration
- Packages (expanded and collapsed)

Connecting lines between the structures show how those structures are related. The relationships we can express include the following:

- Inheritance
- Interface realization

- Composition
- Association
- Dependency
- Aggregation
- Directed association

The last possible element on a UML diagram are the notes. Notes are just what you think they are. Sometimes, an architect needs to add a little more information than what the standard UML allows. Notes let you do that. You shouldn't use them to write an epistle. Short notes in implementation logic are what you see most commonly.

To understand the patterns in this book, you need to understand class diagrams. Every pattern in this book is expressed with at least one UML diagram. The most important patterns are covered with two diagrams each.

Let's take a look at the parts of a UML diagram. They aren't that hard to decipher once you realize the meaning of all the diagram's pieces. Let's start with the structures.

Classes

A class diagram is a visual representation of an object's class. Let's say our program requires us to model a circle like the one in *Figure A2.1*:

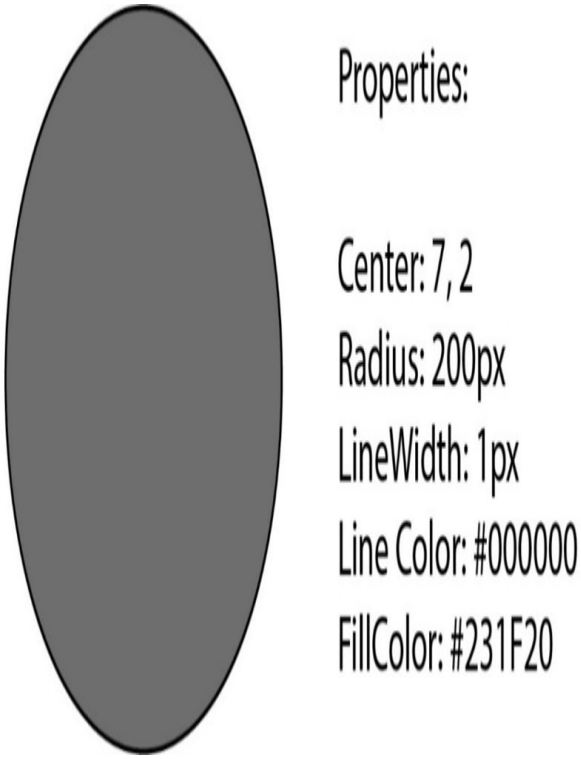


Figure A2.1: This is not a UML diagram of a circle.

If we want to model this in UML, the class diagram would look like *Figure A2.2*.

Figure A2.2: This is an example of a UML diagram. The class name, Circle, is at the top, followed by a list of properties, a dividing line, and a list of methods. The + indicates that they're all public.

That's not too difficult, right? It is a box split into three sections.

The top section contains the name of the class. The class is simply the codified version of what I put in my UML diagram. Understanding these details is central to the coverage in *Chapter 3, Getting Creative with Creational Patterns,* of creation patterns. These are patterns that revolve around object creation.

The middle box in *Figure A2.2* contains a list of the properties that are members of the class. Note they do not have types. Some people put types in the diagram. I was taught not to because that is an implementation detail. The architect simply designs; they don't constrain the builder any more than necessary. The programmer that implements the diagram picks the types. If it happens that you are the architect and the developer, feel free to specify the types if that is helpful.

Beneath the list of properties, we draw a line to separate the properties section from the next section. Visio made this a dashed line for my drawings. The dashed line has no significance. It can be solid or whatever line style that appeals to you.

The bottom box in *Figure A2.2* is a list of methods. It doesn't include details on how to implement the methods, or what they do – just the names, and any arguments.

You might have noticed there are + signs in the diagrams. These refer to your access modifiers. As with types, some architects put them in and some don't. Others put them in when it is vitally important – when the class just won't work unless that detail is honored. I succumbed to the temptation to make a cute diagram to show you what the access modifiers look like in *Figure A2.3*.

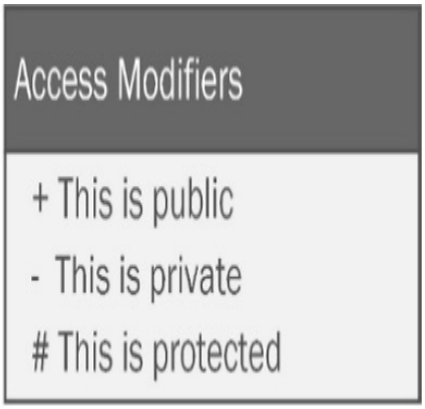


Figure A2.3: Access modifiers.

The plus sign (**+**) denotes a public property or method. The minus sign (**-**) denotes a private property or method, and the hashtag (**#**) indicates that the property is protected. C# supports a few more access modifiers, such as **`internal`**, but they are unique to C#, so UML won't have a symbol for them. If you don't understand these access modifiers, refer to *Appendix 1*.

That's it! UML class diagrams are very simple in nature. We're not done yet though. UML diagrams can show different levels of detail. This really depends on your preferences and the audience for your diagrams. When I studied OOAD in graduate school, my instructor wanted me to put as few implementation details in the diagram as I could. This habit has stuck with me. I've seen other people diagram everything to the smallest detail. I've never been one for micro-managing, so I'm going to stick with leaving as much as possible up to the developer.

Abstract classes

Abstract classes are classes we aren't allowed to instantiate directly. You can only instantiate subclasses of an abstract class. Diagramming them requires one tiny change to the diagram. Here's an example in *Figure A2.4*.

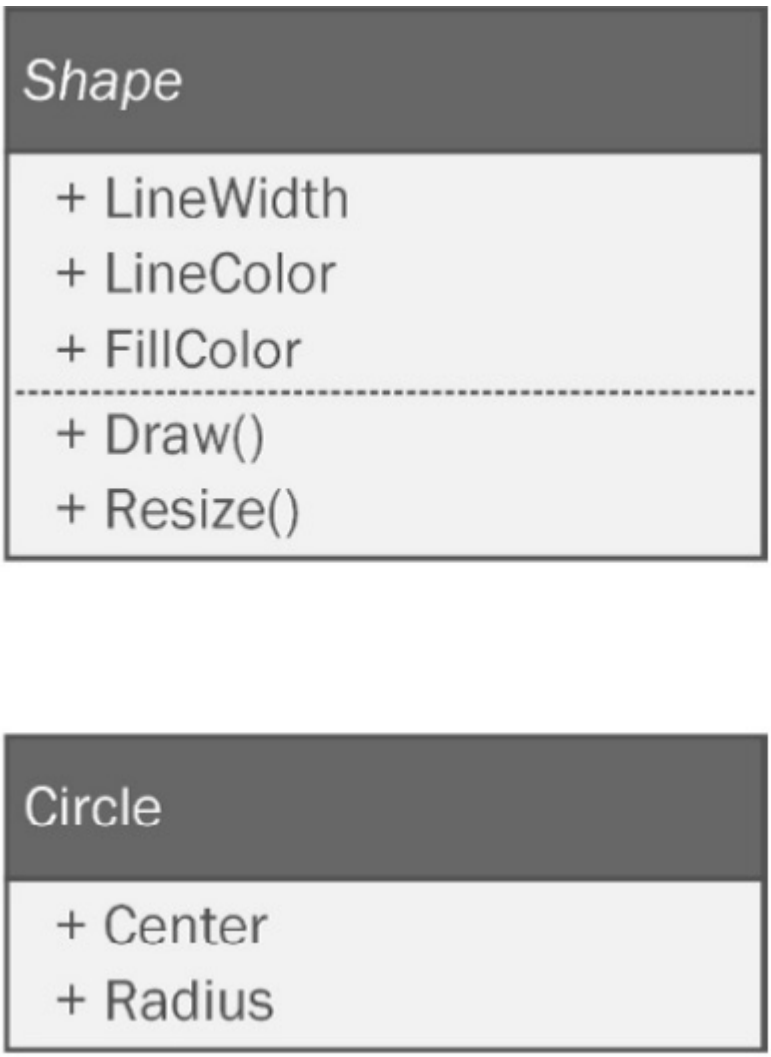


Figure A2.4: When diagramming abstract classes, type the name of the class in italics.

Here, we can see that the **`Shape`** class is abstract and the **`Circle`** class is not. To specify a class as abstract, you need to type the name of the class in the top box in italics. Watch out for this! Sometimes, it can be hard to see, especially if your diagramming tool uses a cute font that makes italics difficult to spot.

Diagramming required types

There are times when you absolutely must define a property type, method argument, or return type because the implementation absolutely requires it. In those cases, you can specify the types in a diagram, as I have in *Figure A2.5*.

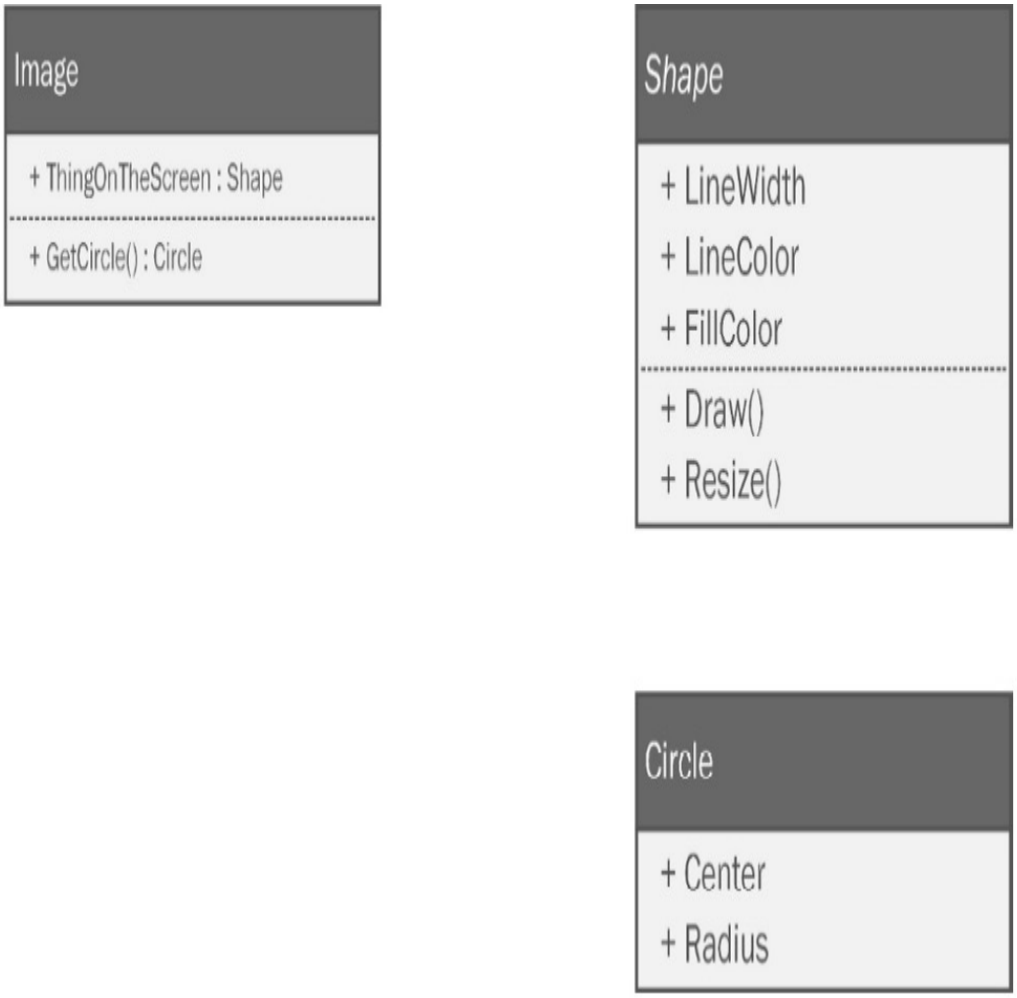


Figure A2.5: You can specify particular types using a colon followed by the type.

The **`Image`** class has a property here that absolutely must be of type **`Shape`**. It's represented with a colon separating the property name and its type. Likewise, I can specify a return type for a method. The **`Image`** class has a method called **`GetCircle()`** that must return **`Circle`**. It is similarly specified with a colon and the type after the method signature is defined.

Constructors

A constructor is a special method that is called when an object is instantiated. *Appendix 1* covers this in detail if you're not sure what I mean. A constructor must have the same name as the class where it is defined. For a lot of people, just seeing a method with the same name as the class is enough. It can also be drawn more formally, as shown in *Figure A2.6*.

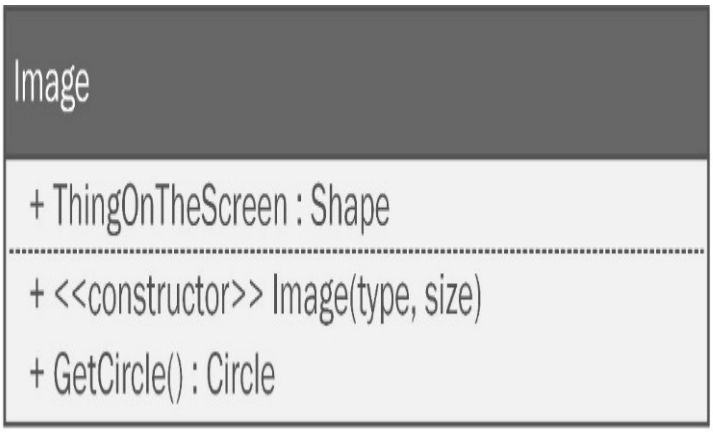


Figure A2.6: Constructors in a UML diagram are difficult to miss.

You can't miss it. It's the method with the obnoxious **<<constructor>>** in front of it. Since obnoxious things are very pedagogical, this is the convention I'll be using.

Don't diagram every tiny detail

In the world of web design, designers use a trick when dealing with their clients that has been around for centuries. When working with initial designs, the objective is to get the layout right and not focus on copy. The difficulty arises when you try to put fake copy in a web page design. Your clients will focus on what the copy says, and they'll try to word-smith the text on the page. The objective is to get buy-in on the design, but that can't happen because all the stakeholders are wrapped up in what the placeholder text says.

To get around this, designers use *lorem ipsum* text. *Lorem ipsum* are words from a famous book on ethics attributed to the Roman statesman and poet Marcus Tullius Cicero. The text looks like this:

“Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.”

This text is placed in a design draft as a placeholder for the actual copy. Since the text is unintelligible, anyone looking at the design with this text in place isn’t tempted to critique the content. They focus on the design.

In UML, when you want to do this, you can use empty, or nearly empty, classes to encourage stakeholders, namely the developers, to focus on the pattern design instead of what you’ve named the properties.

I do this a lot within this book. The projects aren’t real, and the code doesn’t really matter beyond being pedagogical. We’re here to learn the patterns. You can expect to see a lot of diagrams like *Figure A2.7*.

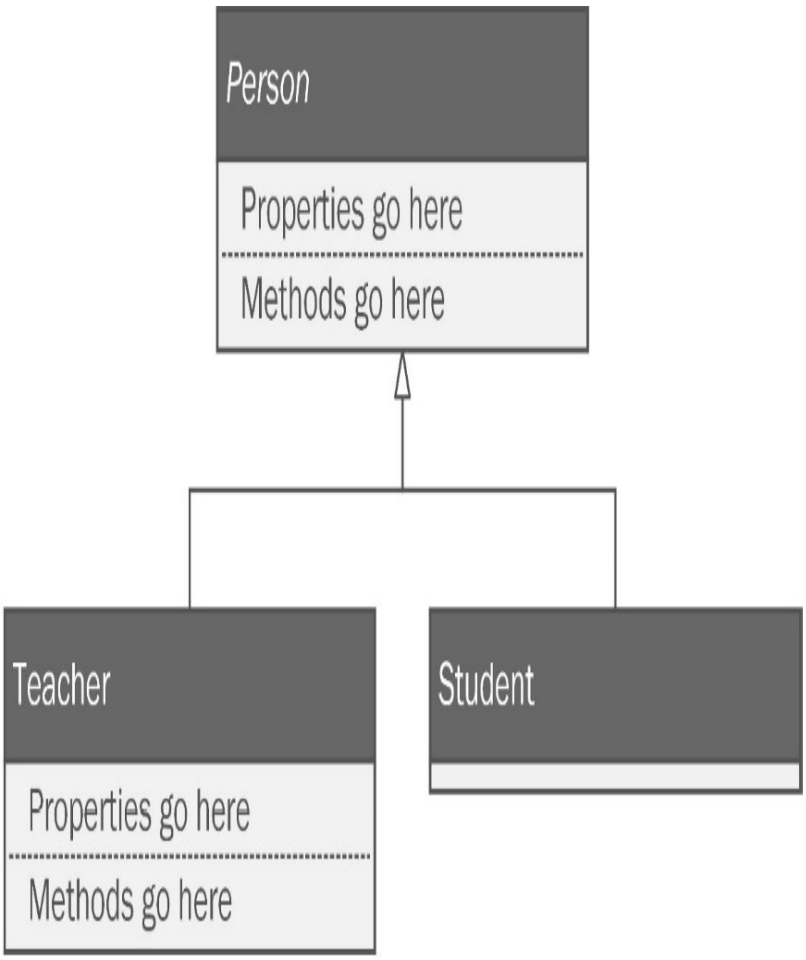


Figure A2.7: When designing with patterns, it is normal to see classes without all the details filled in. The point is to design and critique the structure, not the content.

In the **`Person`** and **`Teacher`** classes, I used placeholders. In the **`Student`** class, I didn't specify anything. Both notations are valid. When you have the structure down, you can choose to either fill in the missing properties, methods, and relationships or leave them as implementation details for the developers.

If you choose the former, be careful. When you're first getting used to working with UML, it is tempting to use it as a tool to specify minutiae. Resist this temptation. If you don't, your diagrams will become so complicated that they lose their meaning.

Interfaces

Interfaces are extremely important to OOAD and pattern diagramming. Drawing them is not very different from drawing classes. They look the same except for an obnoxious **<<Interface>>** expressed in the title box on the diagram. You can see one in *Figure A2.7*.

Figure A2.7: An interface in UML.

Some languages only allow you to implement methods in interfaces. Interfaces in C# can specify public properties in addition to the usual public methods, so it is legitimate to see properties defined on an in-

terface. To reiterate: all properties in methods in an interface have to be public. You'll only ever see + symbols for the access modifiers.

Enumerations

Enumerations refer to a finite set of values that rarely change. For example, the list of states in the United States hasn't changed in 70 years. The days of the week haven't changed in thousands of years. These are good candidates for enumerations. Let's look at a quick example:

```
public enum DaysOfWeek { Mon, Tue, Wed, Thur, Fri
```

They can be used as types, with the effect of preventing spurious data from being passed into a property or variable:

```
var dayOff = DaysOfWeek.Wednesday;
```

This is better than just making it a string:

```
var dayOff = "Wednesday";
```

I set the variable to "**`Wednesday"`**, but as my **`enum`** reveals, my program is expecting a very specific value. The **`"Wednesday"`** string isn't right. I

could easily set any value as long as it is a string. If my class needs a day of the week, and I use an enumeration as the type, it would be impossible to pass in *Bruceday* as a day of the week. *Bruceday* only occurs in my house where every day is a beautiful, beautiful *Bruceday*. I've been trying to standardize it as an internationally recognized day off for years. I figured it was a lock when country superstar Alan Jackson recorded, "It's Bruceday Somewhere." At the last minute, he changed the title to "It's Five O'Clock Somewhere." I think it was a mistake. Apparently, you have to be a Nordic or Greek god, a main sequence star, or a planetary body before you are taken seriously.

An enumeration, like interfaces, features an obnoxious notation added to the title box, as seen in *Figure A2.8*.

Figure A2.8: An enumeration in UML.

When you draw an enumeration, the properties represent the values that go into the enumeration. Enumerations don't have methods, so there won't be anything in the lower part of the diagram. There aren't any access modifiers either. Just the contents are shown.

Packages

Packages are collections of the preceding structures. In C#, you would call it a namespace. There are two ways to draw packages: expanded and collapsed. Expanded packages are drawn with a rec-

tangle around the classes and structures that belong to that namespace. They are used to show the contents of the package.

Collapsed packages don't show the contents. They are just a representation of the namespace. You can see an example of both expanded and collapsed packages in *Figure A2.9*.

Figure A2.9: Expanded and collapsed packages in UML.

I only used the expanded package once in the book to express a dependency on a third-party library.

Connectors

In business, one of the most important factors to success is the relationships between the people you work with. The same can be said of a system of classes, interfaces, enumerations, and packages. In addition to the structures in a class diagram, the diagrams perhaps more importantly express the relationships between the structures, using a standardized set of lines and symbols. These lines connect structures together. Let's look at the relationships expressed in UML class diagrams.

Inheritance

Inheritance between two classes is presented with a solid line with an open triangular arrow on one end. For example, if I have a class called **`Person`**, and another class called **`Student`** that inherits from **`Person`**, the diagram would look like *Figure A2.10*.

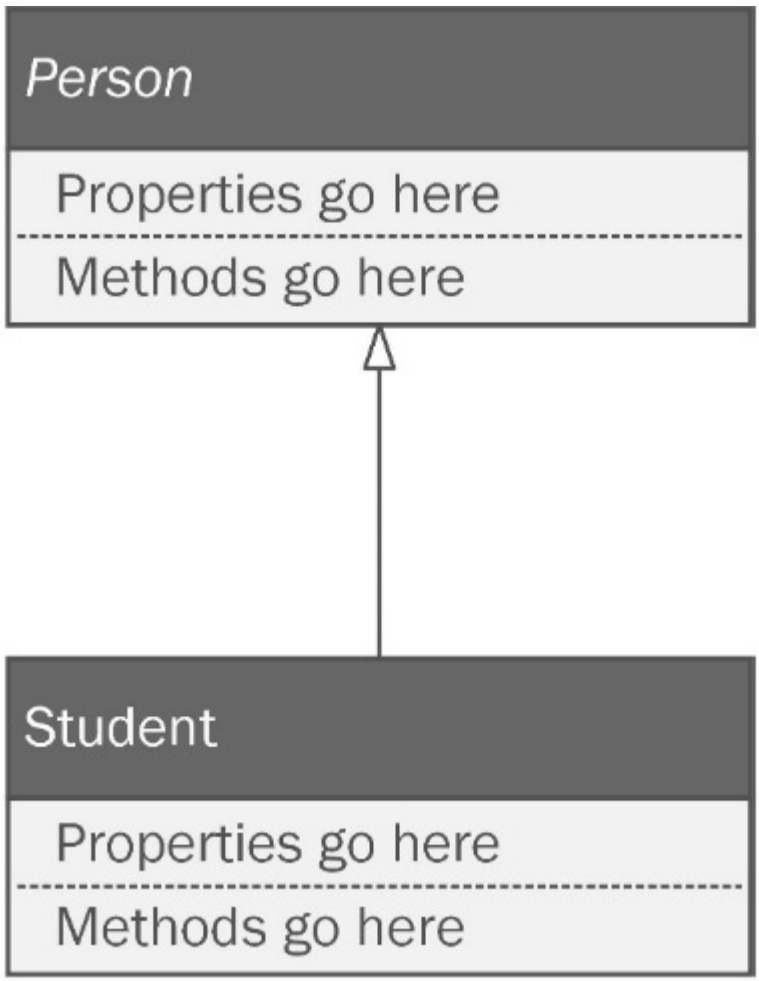


Figure A2.10: Inheritance is represented with a solid line with an empty triangular arrow pointing to the based class.

The arrow should point from the inheriting class to the base class. You are expressing that a student **IS A** person. As a best practice, the base classes are always drawn above the subclasses, so the arrows should always be pointing up.

Interface realization

We discussed interfaces in *Appendix 1* if you're not sure what these are about. Interfaces are the most flexible way to pass around dependencies without tightly coupling two objects together. This makes them very important to our study of patterns.

When a class implements an interface, you draw a dashed line from the class implementing the interface to the interface itself. The end of the line has a hollow arrow pointing to the interface, as shown in *Figure A2.11*.

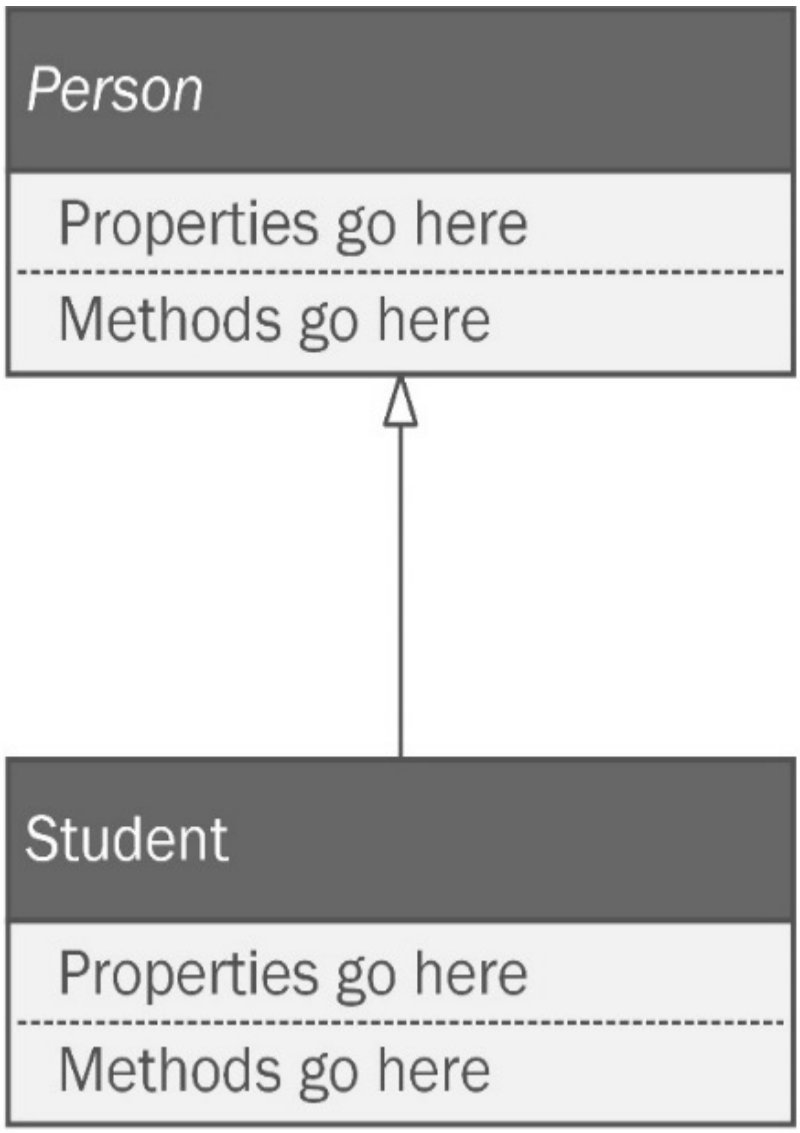


Figure A2.11: An interface realization line is dashed, with a hollow arrow pointing from the implementing class to the interface.

Composition

Composition refers to creating an object out of other objects. You are expressing a relationship by saying **HAS A**. To make a car object, you

might need an engine object and a hood ornament object. You would say a car *has a* hood ornament, and you would be expressing composition. See *Figure A.12* for an example of composition.

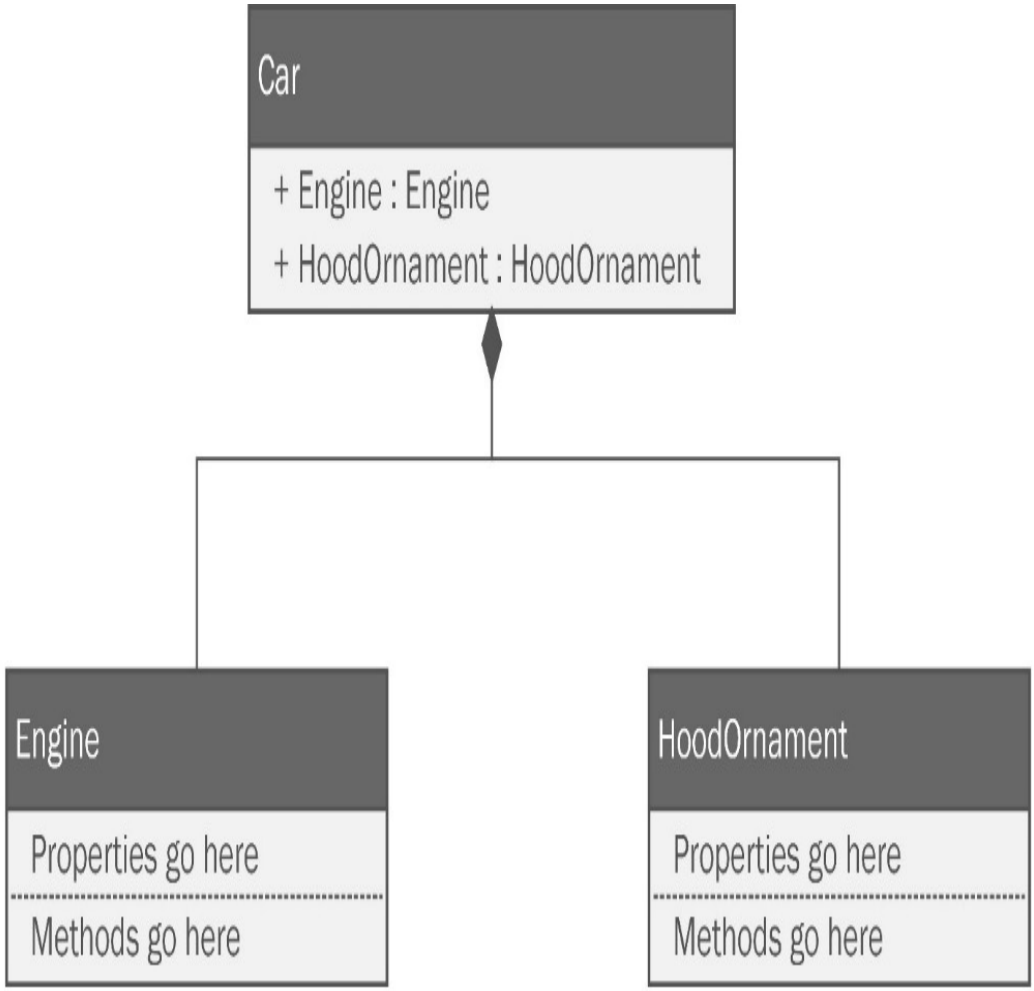


Figure A2.12: Composition in UML is shown using a solid line, with a diamond on the end of the containing class.

In UML, composition is expressed as a solid line drawn between classes, with a filled diamond on one end. The diamond should be pointing to the class that has whatever you're putting into it. In our

Car example, a car has an engine, so the line would go from the engine to the car and the diamond would be pointing to the `Car` class.

Association

Association between two classes indicates that they interact with each other. A `driver` object interacts with a `car` object. The driver puts gas in the car. The car transports the driver to another place. To draw an association in UML, you use a solid line with no arrow, as seen in *Figure A2.13*.

Figure A2.13: Associations show objects that interact with each other.

Aggregation

Aggregations are a type of association involving multiplicity. A `warehouse` object will have a relationship with inventory items expressed as "An order has many inventory items." Don't confuse it with composition. Not only does it seem like the same thing, but the symbols are

also nearly identical. Aggregation is an empty diamond on a solid line, as seen in *Figure A2.14*.

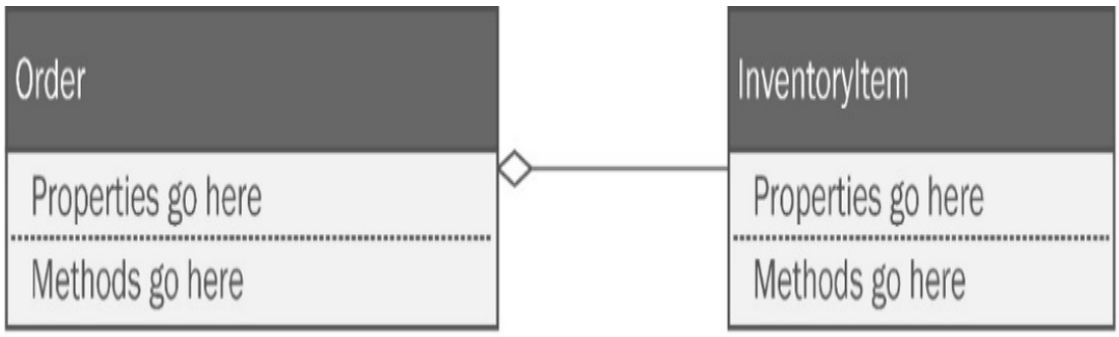


Figure A2.14: Aggregation is an empty diamond on a solid line.

Aggregation and composition are different. Composition refers to the construction of an object from other objects. Aggregation refers to a relationship where there is a one-to-one, one-to-many, or many-to-many relationship between objects, but the objects can stand alone. If we had a **`CollegeCourse`** object, we could express aggregation with **`student`** objects. There are many students in one class. If the college cancels the class, the students don't instantly vanish. You can have a student without a class.

Directed association

A directed association shows a container relationship. A **`spaceship`** object is related to **`passenger`** objects in this way. Passengers are contained within the spaceship. Note that the spaceship is not com-

posed of passengers, so it's not the same thing as composition. A directed association can be diagramed as seen in *Figure A2.15*.

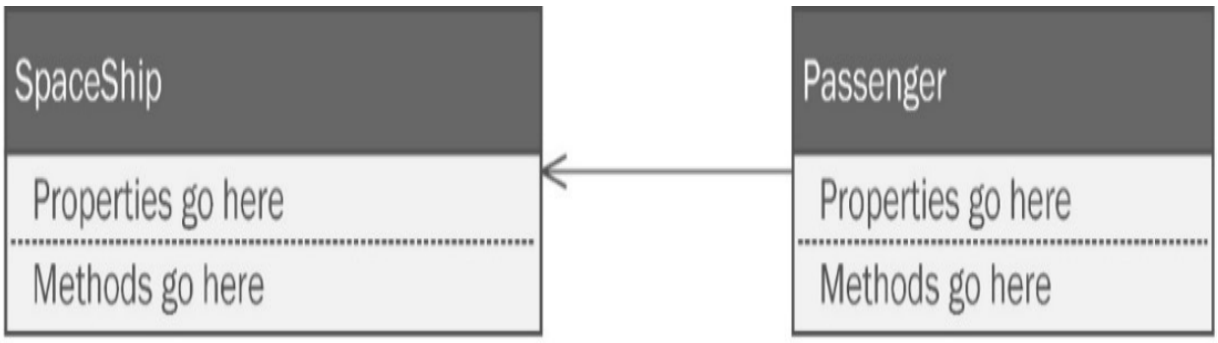


Figure A2.15: Directed association between two classes.

The line is solid and the arrowhead is two lines rather than a triangle.

Dependency

When one class depends on another, you will find yourself changing both classes when modifications are needed. If you change one, you have to change the other. Imagine an electrical plug and an electrical socket. They are truly designed to be physically dependent on one another. If you had to change the design for the plug, you'd probably have to change the socket design too, and vice versa. A dependency relationship between two classes is something we usually strive to avoid, even though they are unavoidable. That might sound very Zen, but I'm referring to a relationship between two classes. A dependency is expressed using a dashed line with a non-triangular arrowhead on the end, as seen in *Figure A.16*.

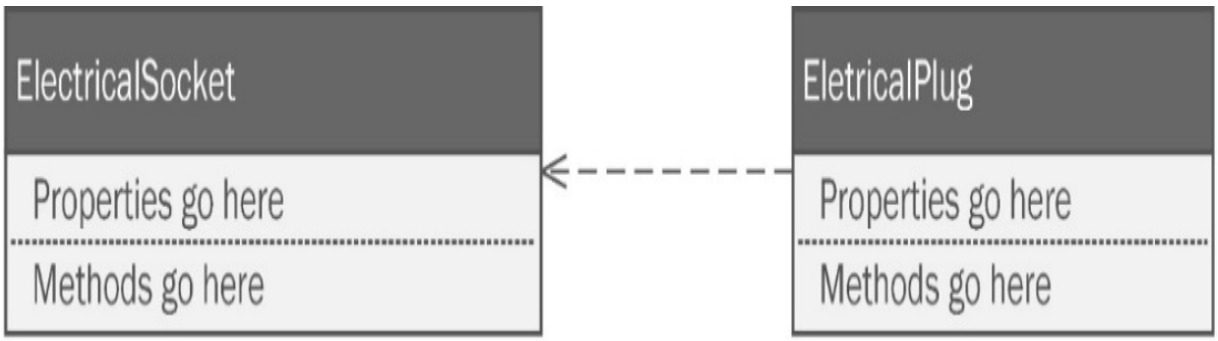


Figure A2.16: Dependencies between classes are drawn with a dashed line and an angular arrowhead pointing from the dependent to the dependence.

These relationships can be vocalized as A **depends on** B changing as little as possible. I added the last part to remind me of the SOLID principle discussed in *Chapter 2, Prepping for Practical Real-World Applications of Patterns in C#* . The O in SOLID refers to the open-closed principle. Classes should be open for extension but closed for modification. Dependencies are unavoidable, but when you see them in the diagram, scrutinize them and implement them with care.

Notes

Notes are just what you think they are. Sometimes, an architect needs to add a little more information than what standard UML allows. Notes let you do that. You can see an example in *Figure A2.17*.

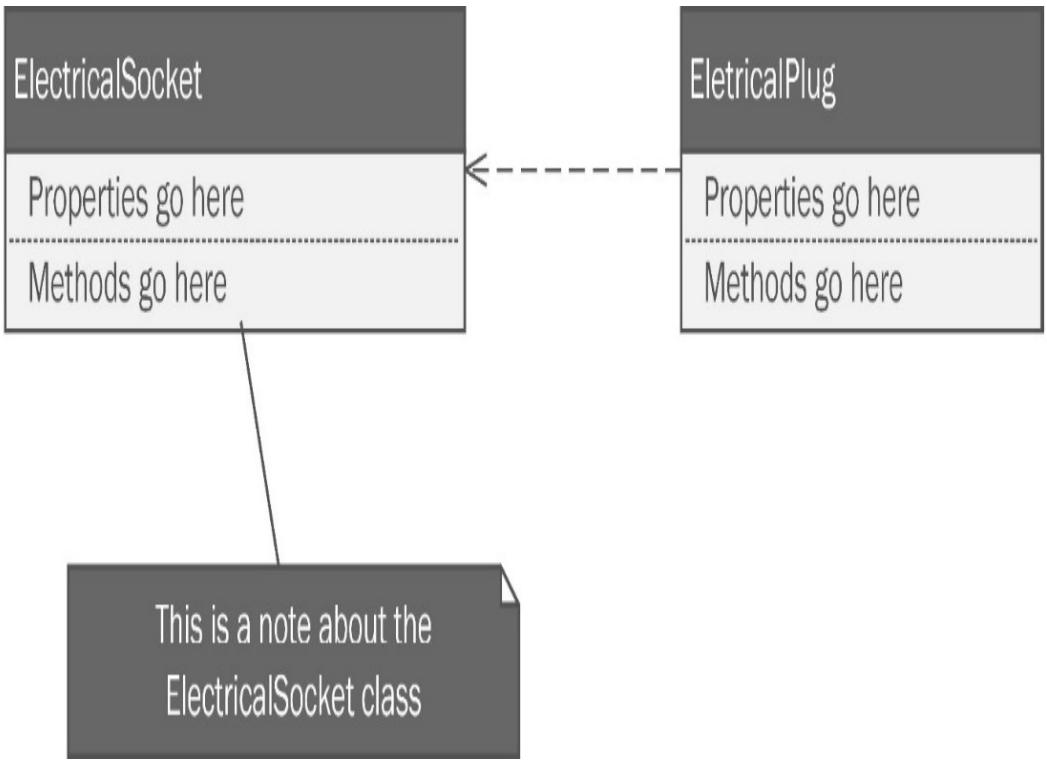


Figure A2.17: This is what a note looks like in UML.

You shouldn't use them to write an epistle. Short notes in implementation logic are what you see most commonly.

Best practices

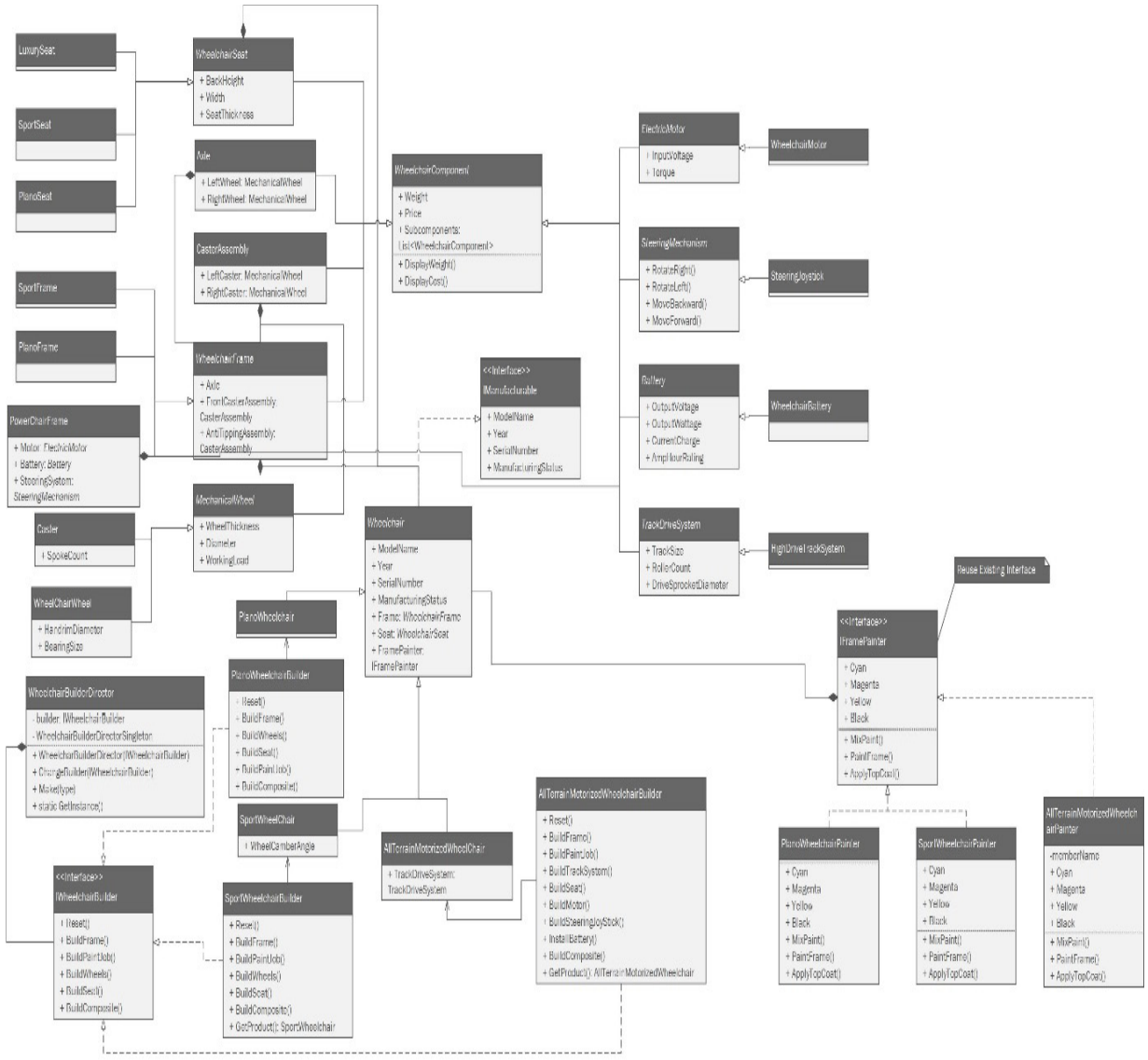
UML class diagrams are easy to create and understand, or at least they should be. Unfortunately, many diagrams fall victim to the same forces I covered in *Chapter 1*. They can be a Golden Hammer, and they can grow uncontrollably to become too complicated to be useful. Software succumbs to these forces slowly over time. Sometimes, it takes years to make a repository full of spaghetti. Diagrams tend to

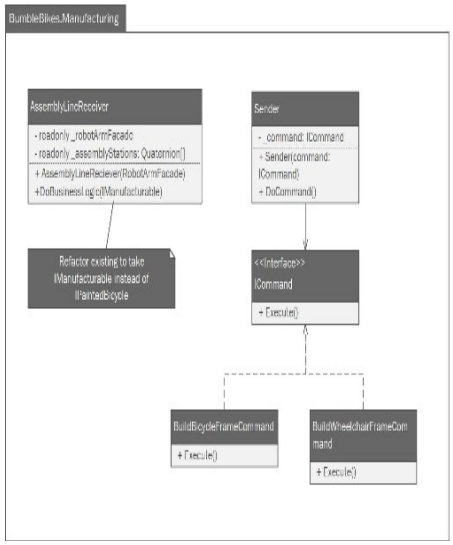
fall apart over the space of days. UML is a plan. If your plan is a messy disaster, how can your software be anything else?

To curb the tide of these destructive forces, I'm going to give you four hints to keep your diagrams useful, easy to read, and hopefully, keep you out of analysis paralysis.

Less is more – don't try to put everything in one big diagram

UML diagrams are meant to be used by the development team, but they are often shared with other project stakeholders. If you go to a meeting with a diagram that looks like the guidance system schematic for the Patriot missile system, you shouldn't expect to be well received. The developers won't appreciate you and management will not think highly of you. The UML for the project in *Chapter 6, Step Away from the IDE! Designing with Patterns Before You Code,* and *Chapter 7, Nothing Left but the Typing: Implementing the Wheelchair Project*, could easily become way too big if you try to diagram the entire project in one diagram. Check out *Figure A2.18*. Even if I printed it on a plotter 3 feet wide by 2 feet tall, the diagram would still be hard to read.


LuxurySeat
SportSeat
PianoSeat
WheelchairSeat
+ BackHeight
+ Width
+ SeatThickness
Axle
+ LeftWheel: MechanicalWheel
+ RightWheel: MechanicalWheel
CasterAssembly
+ LeftCaster: MechanicalWheel
+ RightCaster: MechanicalWheel
SportFrame
PianoFrame
WheelchairFrame
+ Axle
+ FrontCasterAssembly: CasterAssembly
+ AntiTippingAssembly: CasterAssembly
PowerChairFrame
+ Motor: ElectricMotor
+ Battery: Battery
+ SteeringSystem: SteeringMechanism
MechanicalWheel
+ WheelThickness
+ Diameter
+ WorkingLoad
Caster
+ SpokeCount
WheelChairWheel
+ HandrimDiameter
+ BearingSize
WheelchairComponent
+ Weight
+ Price
+ Subcomponents: List<WheelchairComponent>
+ DisplayWeight()
+ DisplayCost()
<<Interface>>
IManufacturable
+ ModelName
+ Year
+ SerialNumber
+ ManufacturingStatus
Wheelchair
+ ModelName
+ Year
+ SerialNumber
+ ManufacturingStatus
+ Frame: WheelchairFrame
+ Seat: WheelchairSeat
+ FramePainter: IFramePainter
PianoWheelchair
SportWheelChair
+ WheelCamberAngle
AllTerrainMotorizedWheelChair
+ TrackDriveSystem: TrackDriveSystem
ElectricMotor
+ InputVoltage
+ Torque
WheelchairMotor
SteeringMechanism
+ RotateRight()
+ RotateLeft()
+ MoveBackward()
+ MoveForward()
SteeringJoystick
Battery
+ OutputVoltage
+ OutputWattage
+ CurrentCharge
+ AmpHourRating
WheelchairBattery
TrackDriveSystem
+ TrackSize
+ RollerCount
+ DriveSprocketDiameter
HighDriveTrackSystem
Reuse Existing Interface
<<Interface>>
IFramePainter
+ Cyan
+ Magenta
+ Yellow
+ Black
+ MixPaint()
+ PaintFrame()
+ ApplyTopCoat()
PianoWheelchairPainter
+ Cyan
+ Magenta
+ Yellow
+ Black
+ MixPaint()
+ PaintFrame()
+ ApplyTopCoat()
SportWheelchairPainter
+ Cyan
+ Magenta
+ Yellow
+ Black
+ MixPaint()
+ PaintFrame()
+ ApplyTopCoat()
AllTerrainMotorizedWheelchairPainter
-momborName
+ Cyan
+ Magenta
+ Yellow
+ Black
+ MixPaint()
+ PaintFrame()
+ ApplyTopCoat()
WheelchairBuildorDirector
- builder: IWheelchairBuilder
- WheelchairBuilderDirectorSingletor
+ WheelchairBuilderDirector(IWheelchairBuilder)
+ ChangeBuilder(IWheelchairBuilder)
+ Make(type)
+ static GetInstance()
<<Interface>>
IWheelchairBuilder
+ Reset()
+ BuildFrame()
+ BuildPaintJob()
+ BuildWheels()
+ BuildSeat()
+ BuildComposite()
PianoWheelchairBuilder
- Reset()
- BuildFrame()
- BuildWheels()
- BuildSeat()
- BuildPaintJob()
- BuildComposite()
SportWheelchairBuilder
+ Reset()
+ BuildFrame()
+ BuildPaintJob()
+ BuildWheels()
+ BuildSeat()
+ BuildComposite()
+ GetProduct(): SportWheelchair
AllTerrainMotorizedWheelchairBuilder
+ Reset()
+ BuildFrame()
+ BuildPaintJob()
+ BuildTrackSystem()
+ BuildSeat()
+ BuildMotor()
+ BuildSteeringJoyStick()
+ InstallBattery()
+ BuildComposite()
+ GetProduct(): AllTerrainMotorizedWheelchair


BumbleBikes.Manufacturing
AssemblyLineReceiver
- readonly _robotArmFacade
- readonly _assemblyStations: Quaternion[]
+ AssemblyLineReciever(RobotArmFacade)
+DoBusinessLogic(IManufacturable)
Refactor existing to take IManufacturable instead of IPaintedBicycle
Sender
- _command: ICommand
+ Sender(command: ICommand)
+ DoCommand()
<<Interface>>
ICommand
+ Execute()
BuildBicycleFrameCommand
+ Execute()
BuildWheelchairFrameCommand
+ Execute()

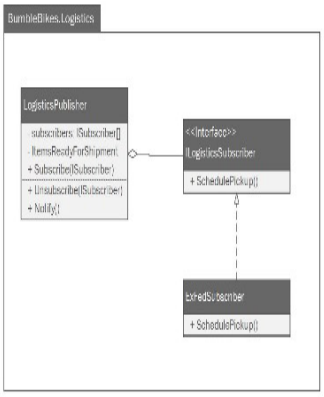

BumbleBikes.Logistics
LogisticsPublisher
subscribers: ISubscriber[]
- ItemsReadyForShipment:
+ Subscribe(ISubscriber)
+ Unsubscribe(ISubscriber)
+ Notify()
<<Interface>>
ILogisticsSubscriber
+ SchedulePickup()
ExtredSubscriber
+ SchedulePickup()

Figure A2.18: Don't do this! This diagram covers the entire project presented in *Chapter 7*. It is too big, too complicated, and it won't fit on one page.

It's better to break the system down into small parts and diagram those. You'll see this done in *Chapter 7* . Make diagrams of small systems and keep them simple. As a rule, I try to keep to one or two pages at most. If the diagram won't fit on two standard sheets of paper, it is probably too big, and you will need to find a way to break it down into smaller pieces.

Don't cross the lines

In the classic movie *Ghostbusters*, the heroes of the story invent a dangerous, high-tech proton accelerator that fires a stream of energy capable of trapping and holding ghosts. In the movie, the chief scientist who invented the technology warns his comrades to never cross the streams, meaning they should never let the bolts of energy from two proton packs cross one another. Try to imagine all life as you know it instantly coming to an end as all the atomic particles in your body explode at the speed of light. They called it a *total protonic reversal*. That's bad.

Now, apply the same caution to crossing lines in your UML diagrams. Crossed lines are ambiguous. The best you can do is to make it look

like a circuit diagram and have one line hop over another. You can see an example in *Figure A2.19*.

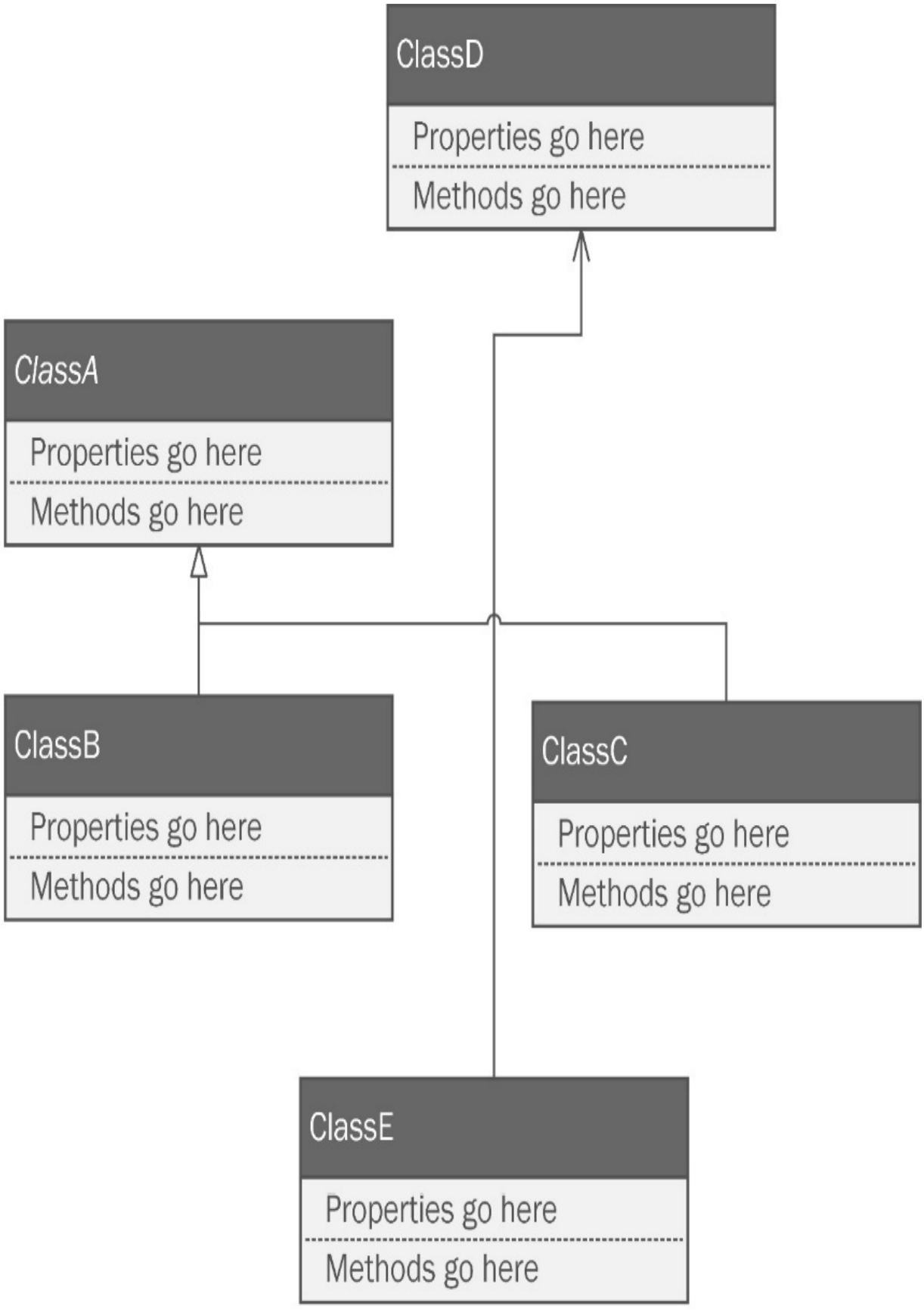

ClassD
Properties go here
Methods go here
ClassA
Properties go here
Methods go here
ClassB
Properties go here
Methods go here
ClassC
Properties go here
Methods go here
ClassE
Properties go here
Methods go here

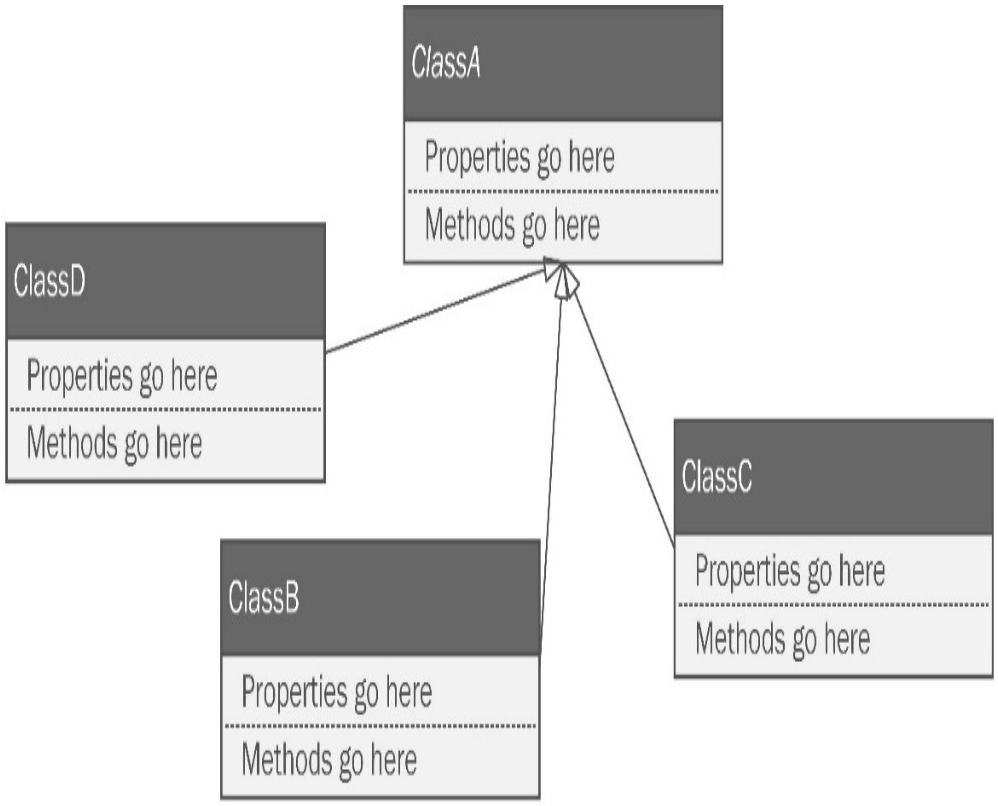
Figure A2.19: The connector between Class D and Class E crosses the line between Class C and Class A.

You're not sure where they originate and where they terminate. If there is no way to draw a class diagram without crossing lines, it is probably too big.

This doesn't apply if the lines are going to the same place. Class B and Class C are both inheriting from Class A. Technically, their lines join rather than cross. When this happens, I make an effort to join the lines together so that they look like one line.

The most direct path for lines leads to a mess

This best practice is baked into the better UML diagramming tools you'll find online. Even Visio does it by default. Drawing straight lines between classes leads to a messy diagram. You can see what I mean in *Figure A2.20*.


ClassA
Properties go here
Methods go here
ClassD
Properties go here
Methods go here
ClassC
Properties go here
Methods go here
ClassB
Properties go here
Methods go here

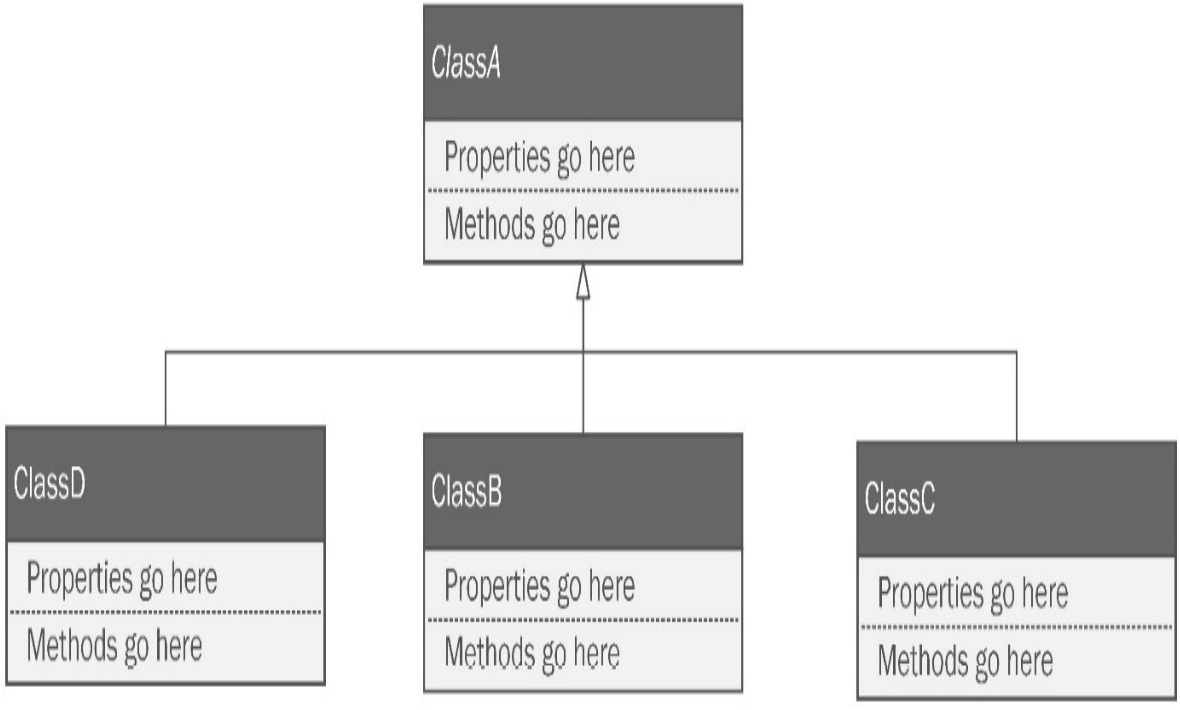

ClassA
Properties go here
Methods go here
ClassD
Properties go here
Methods go here
ClassB
Properties go here
Methods go here
ClassC
Properties go here
Methods go here

Figure A2.20: Circuitous lines with right angles instantly make a diagram look better.

It is better to draw a more circuitous line that is easy to follow and, as we already noted, doesn't cross other lines.

Parents go above children

When you are diagramming any sort of inheritance or interface implementation, you should always draw the parent element above, or at least higher than, the child element, as depicted in *Figure A2.21*.

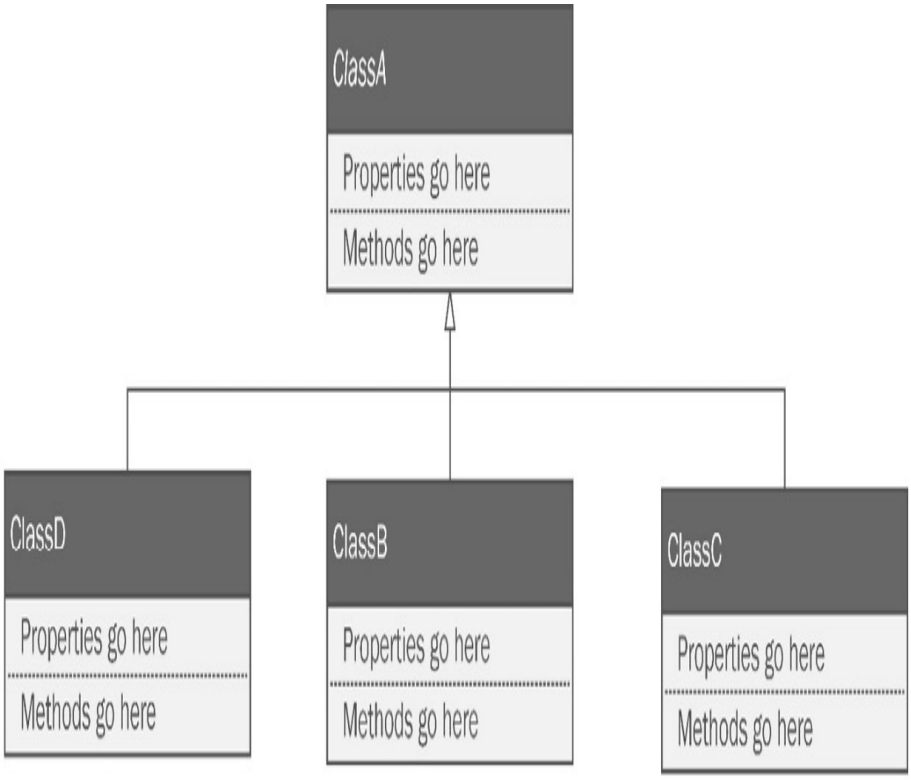


Figure A2.21: Parent elements should always be above child elements, which means your arrows should always point up.

This makes your diagram easier to follow as long as you are consistent.

Keep your diagrams neat

Spend time keeping your diagrams tidy. I'd give you the same advice for your code. Clean up things you aren't using anymore. Correct your spelling. Details matter! I would add that you should pick fonts that are easy to read. Fonts in your diagram should show italics very well so that you can always see abstract classes.

Summary

This appendix covered the UML. While it sounds like a programming language, it is really a standard way of drawing diagrams to represent structures and patterns of code. We only need 1 of the 14 recognized UML documents to get through this book, but the one we do use is used quite extensively.

The class diagram shows the structures of a system along with the relationships between those structures. The structures may be classes, interfaces, enumerations, and packages. Common relationships include inheritance, interface realization, composition, association, and more. Notes can be added to a diagram to add more details but should be kept concise.

We learned the best practices for UML class diagrams, including keeping a diagram as legible as possible. Avoid clutter in the diagram that comes from trying to define every single possible relationship between structures. Focus instead on those that are vital to the implementation of the diagram. A good diagram isn't necessarily one that definitely conforms to the standards of UML. A good diagram is one that communicates quickly, with just enough detail to allow a developer to successfully implement the intent of the diagram, without micromanaging every detail.

Further reading

Baumann, Henriette, Patrick Grassle, and Philippe Baumann. *UML 2.0 in action: A project-based tutorial*. Packt Publishing, 2005.

Index

As this ebook edition doesn't have fixed pagination, the page numbers below are hyperlinked for reference only, based on the printed edition of this book.

A

abstract classes 351, 391

Abstract Factory pattern 97-103

abstract syntax trees (ASTs) 300

abstraction 40

access methods

getter 343

setter 344

access modifiers

about 341

adding 342

internal 342

private 342

private protected 342

protected 342

protected internal 342

public 342

Active Domain Object (ADO) pattern 312

ActiveX Data Objects (ADO) 312

Ada 329

Adapter pattern 292-294

Adobe Illustrator 341

aggregations 399

Agile development 260

aliases 333

ampere (amp) 229

application programmer interface-(API-) 140

arrays 325

artificial intelligence (AI) 296

aspect-oriented programming (AOP) 315

association 399

B

Behavioral patterns 173

bicycle

CAD design 74-79

initial design 79-87

Bicycle class 127, 128

bicycle project, patterns

behavioral patterns 233

creation patterns 232

structural patterns 232

Big Ball of Mud 9, 10

Big Ball of Mud, forces

change 12

complexity 12

cost 11

experience 11

scale 12

skills 11

time 11

visibility 12

bill of materials (BOM) 276

bit 332

blue screen of death (BSOD) 326

booleans 324

Bridge pattern

about 158-166, 242, 243

implementing 167

reviewing 160

use case 166

used, for painting wheelchair 283-287

broken windows theory 35

Builder interface 106

Builder pattern 103-112, 127, 235, 236

Builder pattern, Wheelchair

concrete component classes, adding 276

implementing 270-274

setting up 279-281

WheelchairComponents folder, refactoring 274, 275

Bumble Bikes factory 214-216

C

C#

auto-implemented properties 344

automated garbage collection 326

automatic bounds checking 324-325

background 321

design goals 321

detection, for uninitialized variables 324, 325

general-purpose language 321, 323

highly portable code 326-327

language mechanics 327-329

purely and fully object-oriented 323

type system 323, 324

variables 329, 330

Chain of Responsibility pattern 296, 297

character type 333

class

about 334-337

constructor 337, 338

decorating 131

instantiation 337

methods 339, 340

name 337

namespaces 336

parts 336

properties 339

using statements 336

class diagram

about 73, 388-391

abstract classes 391

constructor 392, 393

designing, without all details 393, 394

enumerations 395, 396

interfaces 394

packages 396

required types diagramming 392

structures 388, 389

clean code

characteristics 42

cognitive load, limiting 46

commenting 48

consistency, enforcing 46

consistency, establishing 46

style, enforcing 46

style, establishing 46

terse 47, 48

writing 41

writing, readable by humans 42-46

client-server 36

client-server pattern 310

cloning 291

cloud computing 36

CMYK color model 158, 283

code reviews

about 66, 67

functionality 68

overall design 67, 68

cognitive load 46

Collections 334

command-line programs 356

Command pattern

about 173, 244-246

applying 175

code, testing 180

coding 176-179

components 174, 175

Command Query Responsibility Segregation (CQRS) pattern 311

commercial off-the-shelf (COTS) components 140

Common Intermediate Language (CIL) 327

Common Object Request Broker Architecture (CORBA) 141

Composite pattern

about 148-152, 157, 237-241

finishing up 268, 269

implementing 152-156

composition 41, 398, 399

computer-aided design (CAD) 218

concrete component classes, Plano Wheelchair

about 276

axles 276

casters 276

frames 277

seats 277

wheels 278, 279

constructor

about 337, 338, 392, 393, 396

aggregations 399

association 399

composition 398, 399

dependency 400

directed association 400

inheritance 397

interface realization 397

Continuous Integration (CI) 46

controller layer 310

creatine kinase (CK) 215

Creational patterns 73

D

data 323

Data Access Object (DAO) 312

Data access patterns

about 311

Active Domain Object (ADO) pattern 312

demand cache 312

optimistic lock 313

ORM 311, 312

pessimistic lock 313

transaction 312

Data Definition Language (DDL) 20

Data Manipulation Language (DML) 22

Decorator pattern

about 127-136, 314

ConcreteComponent 129

ConcreteDecorator1 129

ConcreteDecorator2 129

Decorator 129

IComponent 129

working 136-138

decoupling 61

deep copying 290

demand cache 312

dependency 400

Dependency Injection (DI) 62

Dependency Inversion Principle (DIP) 61-65

directed association 400

Director class 106

Document Object Model (DOM) 300

domain-specific languages (DSLs) 290

Don’t Repeat Yourself (DRY) 43, 223, 272

duck typing 330

E

Electric Light Orchestra (ELO) 213

encapsulation

about 40, 340, 341

accessor logic, with backing variables 346, 347

enumerations 81, 395, 396

European Computer Manufacturers Association (ECMA) 321

F

Façade pattern

about 139-142

BuffingAPI 143

GrabbingAPI 143

implementing 143-147

RobotArmFacade class 143

WelderAttachmentAPI class 143

factory class 89

Factory Method pattern 91-97

first-in, first-out (FIFO) 182

FixComponent() method 264-267

Flyweight pattern 294, 295

frame 226

functional programming

versus OOP 323

G

Gang of Four (GoF) software design patterns 211, 323

Generics 334

god function 49

Golden Hammer 12-15

I

IDEs, for C# development

about 355

Rider 378

Visual Studio 356

VS Code 367

implicit typing 330

inheritance

about 40, 347, 397

working 347-351

Inkscape 341

instantiation 73, 337

intent 314

interface realization 397

interfaces

about 41, 351, 352, 394

defining 352-354

implementing 354, 355

Interface Segregation Principle (ISP) 59-61

Internet of Things (IoT) 64, 141

Interpreter pattern 290

introspection 365

inventory management systems (IMSs) 313

Inversion of Control (IoC) 62

iterative development 260

Iterator pattern

about 181-183

applying 184

coding 185-191

implementing 191, 192

J

JavaScript Object Notation (JSON) 28

Java Virtual Machine (JVM) 321

K

keeping it working 10

L

lasagna code 36-40

Liskov Substitution Principle (LSP) 55-58

M

Maverick wheelchair 218, 219, 225

Mediator pattern 300, 301

memento 313

Memento pattern 289, 302-304

methods 339

Microservices pattern 310

Microsoft Disk Operating System (MS-DOS) 356

Microsoft Intermediate Language (MSIL) 321

minimum viable product (MVP) 260

model layer 310

Model-View-Controller (MVC) pattern 310

Monodevelop 356

N

namespaces 336

No pattern implementation 87-89

notes 401

NuGet 21

numbers 324, 330

O

Object-Oriented Design (OOD) 49

object-oriented programming (OOP) language 327

versus functional programming 323

object pool pattern 112-117

Object-Relational Mapper (ORM) 21, 142, 310-312

objects 328

Observer pattern

about 192-194

applying 195

coding 195-199

Open-Closed Principle (OCP) 51-55, 400

optimistic lock 313

organic light-emitting diode (OLED) 216

original equipment manufacturer (OEM) 8

P

packages 396

patterns

about 30, 289

Adapter pattern 292, 293

adding 232, 233

Chain of Responsibility pattern 296, 297

designing with 217-220

design meeting 233, 234

first pass 220-224

Flyweight pattern 294, 295

Mediator pattern 300, 301

Memento pattern 302-304

need for 29-31

Prototype pattern 290, 291

Proxy pattern 298, 299

second pass 234, 235

state pattern 304-307

Visitor pattern 308, 309

patterns, elements

name and classification 313

problem description 314

solution description 314, 315

using, consequences 315

patterns, first pass

battery, for powered wheelchair 229

casters 227, 228

frame 226

motor, for powered wheelchair 228

seat 225, 226

steering mechanism, for powered wheelchair 229

track drive system, for Texas Tank 230-232

wheels 227, 228

peer-to-peer (P2P) architecture 310

pessimistic lock 313

physical rehabilitation clinic 217

piecemeal growth 10

Plano Wheelchair 219, 225

polymorphism 40

PoweredChair class 222

powered wheelchair

battery 229

motor 228

steering mechanism 229

primitive data types 324, 332

programming idiom 91

properties 328

Prototype pattern 290, 291

Proxy pattern 298, 299

publish-subscribe (pub-sub) pattern 310

Q

quality

defining 66

measuring 66

quality assurance (QA) 213, 296

quaternions 177

R

random-access memory (RAM) 294

ravioli code 40, 41

Red Green Blue (RGB) color model 158

Representational State Transfer (REST) 141, 300

Resharper 378

Rider

about 378

class, adding 380, 381

command-line project, creating 379, 380

console project, building 385

console project, running 385

interface, adding 380, 381

library project, creating 381-383

library project, linking to console project 383, 384

linked library, testing 384, 385

Roslyn 365

S

seat 225, 226

separation of concerns (SoC) 36, 310

SharpOS 323

signed numeric types 330-332

signed type 331

Simple Factory pattern 89, 90

Simple Object Access Protocol (SOAP) 141

Single Responsibility Principle (SRP) 49-51, 296

singleton 313

Singleton pattern

about 117-122, 236

adding 281, 282

SmallTalk 323

snapshot 302

software architecture patterns

about 310

client-server pattern 310

Command Query Responsibility Segregation (CQRS) pattern 311

Microservices pattern 310

Model-View-Controller (MVC) pattern 310

publish-subscribe (pub-sub) pattern 310

software development kit (SDK) 298

software evolution

explained with pasta 35

SOLID principles

Dependency Inversion Principle (DIP) 61-65

Interface Segregation Principle (ISP) 59-61

Liskov Substitution Principle (LSP) 55-58

Open-Closed Principle (OCP) 51-55

Single Responsibility Principle (SRP) 49-51

used, for creating maintainable systems 49

spaghetti code 35

Stack Overflow 47

State pattern 304-306

static type system 324

Stock Keeping Unit (SKU) 165

Stovepipe system

about 7-9

exploring 8, 9

Strategy pattern

about 199-202

applying 202

coding 203-206

strings 333, 324

strongly typed language 330

strong type system 324

Structured Query Language (SQL) 13, 311, 322

StyleCop 46

T

technical debt 8

Template Method pattern 306, 307

Texas Tank

about 217

designing, with patterns 217-220

track drive system 230-232

throwaway code

about 10

example 15-29

tiers 36

transaction 312

type system 323

U

UML class diagrams

best practices 401-405

Unified Modeling Language (UML) 7, 39, 126, 293

Unity 3D 326

unsigned types 331

US Department of Defense (DoD) 329

user experience (UX) 310

using statements 336

V

variables, C# 329, 330

view layer 310

Visitor pattern 308, 309

Visual Basic 355, 356

Visual Basic for Applications (VBA) 14

Visual Studio

about 356

class, adding 361, 362

command-line project, creating 357-361

console project, building 365-367

console project, running 365-367

editions 356

library project, adding to solution 362, 363

library project, linking to console project 363, 364

library, referencing 364, 365

Visual Studio Community 357

Visual Studio Enterprise 357

Visual Studio Professional 357

Volatile Organic Compounds (VOCs) 165

VS Code

.NET Core, installing 368

about 367

C# extension, adding 373, 374

class, adding 374, 375

code, adding to BicycleLibrary project 375-377

command-line project, creating 370, 371

console project, building 377, 378

console project, running 377, 378

download link 368

interface, adding 374, 375

launching 373

library, linking to console project 373

library project, creating 372

solution, creating 368-370

W

weakly typed language 330

weak type system 324

whatever-as-a-service (WaaS) 13

Wheelchair

base classes 268

class 221

components 259-267

composite pattern, finishing up 268, 269

painting, with Bridge pattern 283-287

project, setting up 252-259

wrapper 314

X

Xamarin Studio 356

Y

You Ain’t Gonna Need It (YANGI) 68, 224

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?

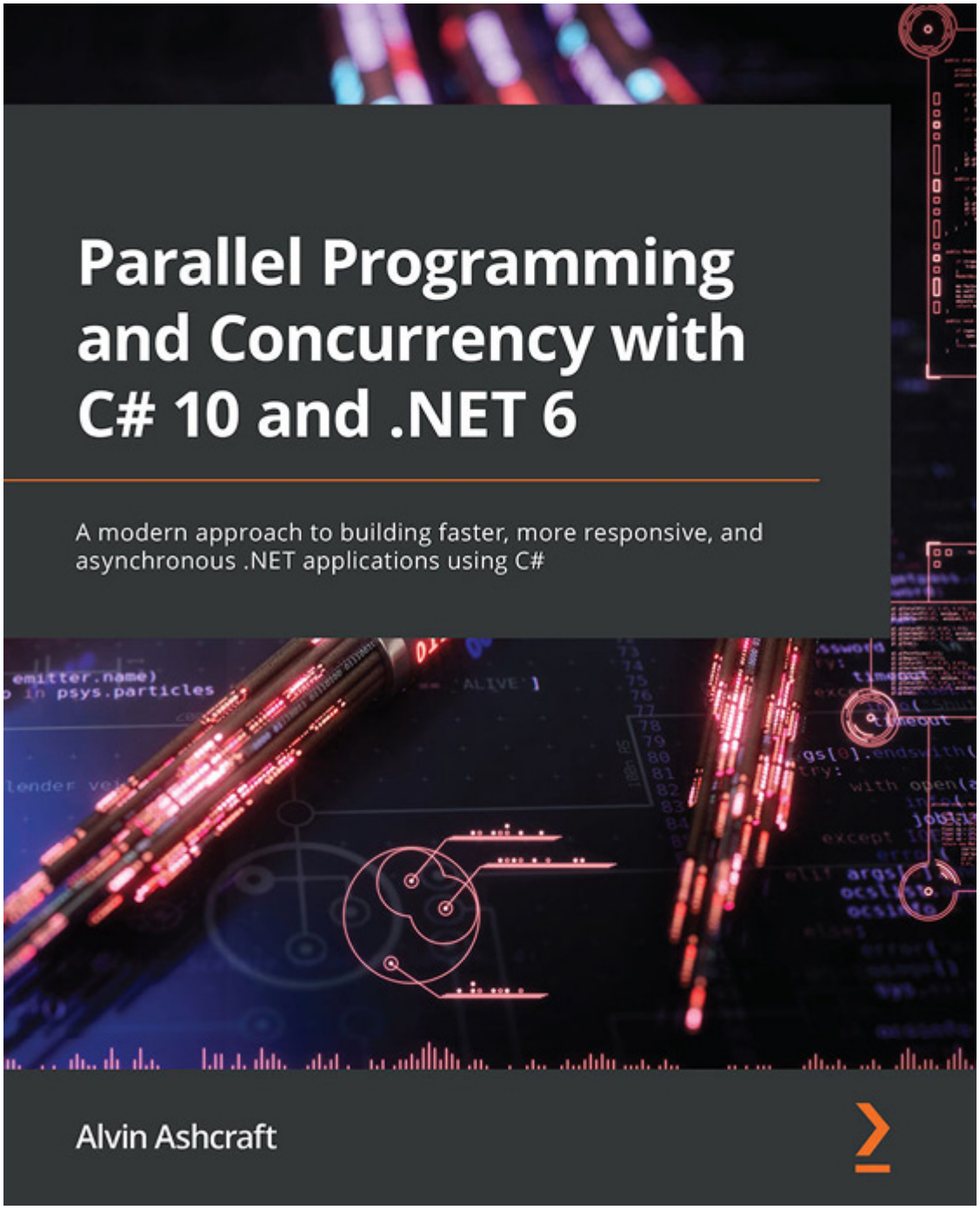
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print,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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

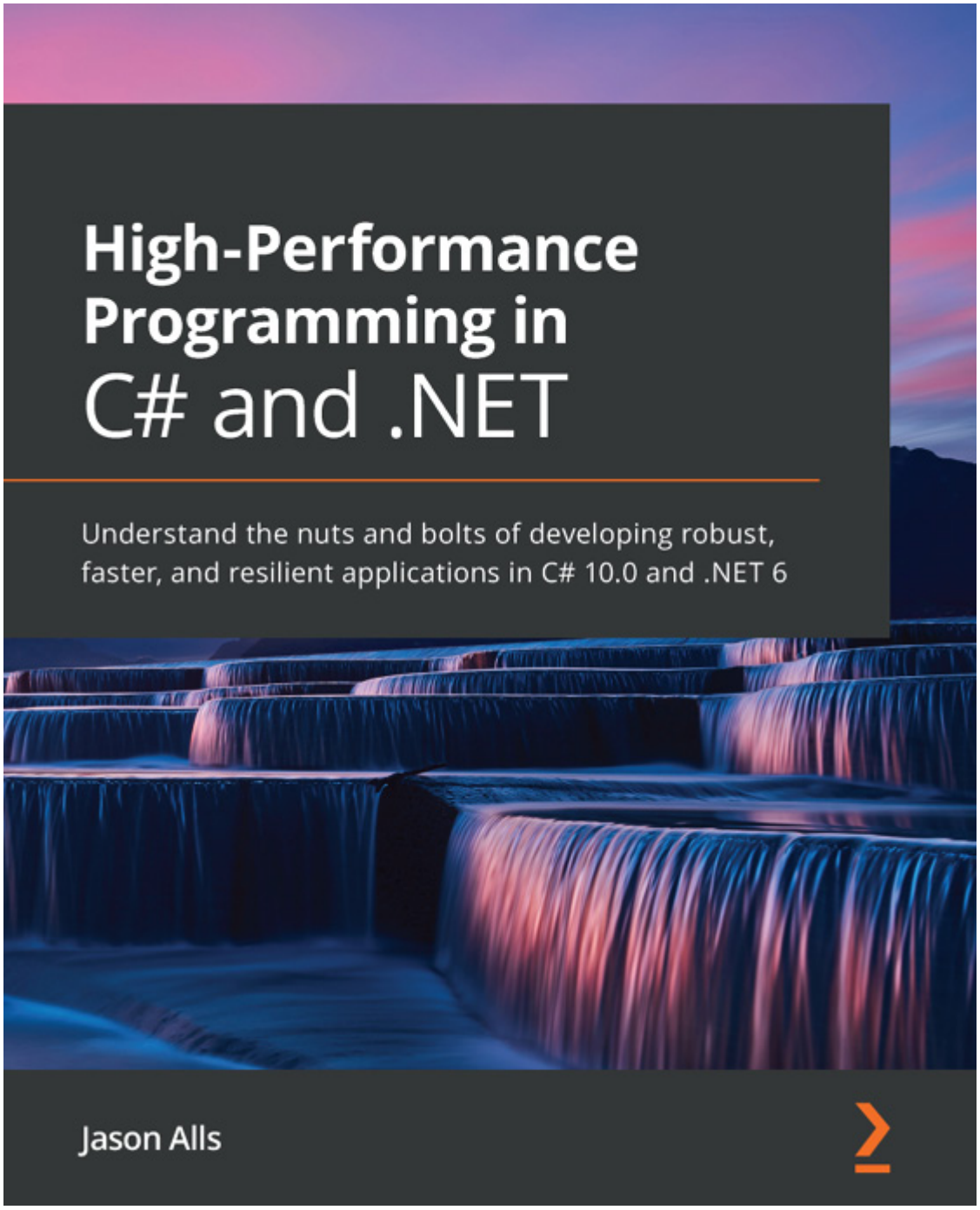


Parallel Programming and Concurrency with C# 10 and .NET 6

Alvin Ashcraft

ISBN: 978-1-803-24367-2

- Prevent deadlocks and race conditions with managed threading
- Update Windows app UIs without causing exceptions
- Explore best practices for introducing asynchronous constructs to existing code
- Avoid pitfalls when introducing parallelism to your code
- Implement the producer-consumer pattern with Dataflow blocks
- Enforce data sorting when processing data in parallel and safely merge data from multiple sources
- Use concurrent collections that help synchronize data across threads
- Debug an everyday parallel app with the Parallel Stacks and Parallel Tasks windows



High-Performance Programming in C# and .NET

Jason Alls

ISBN: 978-1-800-56471-8

- Use correct types and collections to enhance application performance
- Profile, benchmark, and identify performance issues with the codebase
- Explore how to best perform queries on LINQ to improve an application's performance
- Effectively utilize a number of CPUs and cores through asynchronous programming
- Build responsive user interfaces with WinForms, WPF, MAUI, and WinUI
- Benchmark ADO.NET, Entity Framework Core, and Dapper for data access
- Implement CQRS and event sourcing and build and deploy microservices

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.

Share Your Thoughts

Now you've finished *Real-World Implementation of C# Design Patterns*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.