

PRAGMATIC CLEAN CODE

A practical guide to
writing code that lasts

```
{  
  var cleanCode = code  
    .MakeReadable()  
    .MakeTestable()  
    .Refactor();  
}
```

KRYSTYNA
ŚLUSARCZYK



Pragmatic Clean Code

A practical guide to writing code that lasts

Krystyna Ślusarczyk

<packt>

Pragmatic Clean Code

Copyright © 2026 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.

Portfolio Director: Gebin George

Relationship Lead: Dhruv J. Kataria

Project Manager: Prajakta Naik

Content Engineer: Edwin Johnson

Technical Editor: Irfa Ansari

Indexer: Tejal Soni

Production Designer: Shankar Kalbhor

Growth Lead: Priyadarshini Sharma

First published: October 2026

Production reference: 1290926

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80849-379-9

www.packtpub.com

To my wonderful sister. Your turn.

– K.

Contributors

About the author

Krystyna Ślusarczyk is a software developer and programming instructor who has worked in the software industry since 2013. She has spent much of that time teaching, helping students and experienced developers alike make sense of concepts that usually get explained badly. Her approach is built on plain language and real examples, and on the conviction that a rule is easiest to apply once you understand the reasoning behind it.

About the reviewer

Raman Vasikarla is a Site Reliability Engineer specializing in multi-cloud infrastructure provisioning, configuration management, and deployment automation across AWS, Azure, and GCP. With extensive experience in enterprise reliability and observability, his core expertise includes Terraform, Kubernetes, Puppet, Docker, Python, and Go. Raman holds a Master's degree in Computer and Information Sciences and regularly serves as a peer reviewer for academic conferences and technical publications. He also contributes to open-source cloud modules and authors technical articles on cloud resilience and configuration validation.

Table of Contents

Preface	xvii
----------------	-------------

Free benefits with your book.....	xxiii
-----------------------------------	-------

Chapter 1: Understanding clean code	1
--	----------

Technical requirements.....	2
-----------------------------	---

Essential vocabulary	2
----------------------------	---

Why clean code matters	4
------------------------------	---

A project that starts well • 4

Complexity requires continuous maintenance • 5

The cost of poor code quality.....	5
------------------------------------	---

Working code is not finished code • 6

Understanding technical debt.....	6
-----------------------------------	---

Professional responsibility and code quality	8
--	---

What makes code clean?	9
------------------------------	---

A deliberately informal measurement • 9

Practical characteristics of clean code • 9

Comparing clean and poorly written code	10
---	----

Clean code in the age of AI-assisted development	13
--	----

Working with the refactoring case studies	15
---	----

Summary	16
---------------	----

Chapter 2: Meaningful names	17
------------------------------------	-----------

Technical requirements.....	18
-----------------------------	----

Why meaningful names matter.....	18
----------------------------------	----

Naming fundamentals	20
---------------------------	----

Naming variables, parameters, and types • 20

Naming methods • 21

Naming Boolean values and methods as questions • 21

Following language and project conventions	22
--	----

Why developers choose poor names.....	24
---------------------------------------	----

The curse of knowledge • 24	
Using the rubber duck method • 24	
Renaming and the Boy Scout rule.....	25
Renaming public APIs requires more care • 25	
When naming is hard, inspect the design.....	26
Writing expressive names	27
Choosing an appropriate name length	28
Long test names are often useful • 29	
Following the principle of least surprise	30
Maintaining semantic correctness	31
Avoiding ambiguity and inconsistency.....	32
Avoiding homonyms • 32	
Using synonyms consistently • 33	
Matching singular and plural forms • 34	
Using natural language in code • 34	
Choosing a language for identifiers • 35	
Avoiding meaningless words • 35	
Avoiding overly specific names • 35	
Avoiding Hungarian notation and type-based names • 36	
Avoiding confusingly similar names • 37	
Using abbreviations carefully.....	37
Using context effectively	38
Refactoring case study: Improving names.....	39
Understanding the code • 39	
Identifying the issues • 40	
Improving the code • 41	
<i>Make the domain visible • 41</i>	
<i>Make the Boolean method read like a question • 42</i>	
<i>Replace generic loop indices when domain names help • 42</i>	
<i>Replace explanatory comments with focused method names • 42</i>	
A practical naming review workflow.....	44
Summary	45
 Chapter 3: Writing better methods	 47
 Technical requirements.....	 48

Method fundamentals.....	48
Naming methods • 49	
Delegates and lambda expressions • 49	
Designing clear method signatures	49
Choosing the right number of parameters • 50	
Splitting methods to reduce parameters • 51	
Grouping related parameters • 52	
Using Boolean parameters carefully • 53	
Avoiding parameter-count anti-patterns • 55	
Ordering parameters logically • 56	
Keeping methods small and focused	57
Giving each method one task • 57	
Recognizing methods that do too much • 58	
Maintaining consistent levels of abstraction.....	60
Ordering methods for readability.....	62
Keeping methods independent	63
Using pure functions where they fit	63
Why pure functions are easier to work with • 65	
Refactoring case study: Improving method design.....	66
Understanding the code • 66	
Identifying the issues • 68	
Improving the code • 68	
<i>Clarifying the signature • 68</i>	
<i>Validating early and extracting constants • 70</i>	
<i>Extracting file-reading responsibilities • 71</i>	
<i>Simplifying the search and printing the result • 72</i>	
Summary	73
 Chapter 4: Formatting code	 75
Technical requirements.....	76
Why formatting matters	76
Choosing and following a formatting style	78
Using automated formatting tools	79
Applying indentation consistently	81
Structuring code blocks safely	82

Breaking long lines clearly	82
Using vertical space to organize code	84
Refactoring case study: Formatting the Hangman game	86
Understanding the code • 86	
Identifying the issues • 88	
Improving the code • 88	
<i>Apply the formatter before editing manually • 88</i>	
<i>Make scopes explicit • 89</i>	
<i>Extract code when formatting reveals an oversized block • 89</i>	
<i>Extract code when formatting reveals an oversized block • 91</i>	
Summary	93

Chapter 5: Applying the Single Responsibility Principle 95

Technical requirements.....	96
Introducing the SOLID principles	96
Understanding the Single Responsibility Principle	97
Why combining responsibilities creates problems • 97	
Separating the responsibilities • 99	
A single responsibility does not mean a single method • 100	
Orchestration can still be one responsibility • 101	
Using one reason to change as an SRP test • 102	
A practical SRP checklist • 103	
Summary	104

Chapter 6: Applying the Open-Closed Principle 105

Technical requirements.....	106
Reviewing abstract types and polymorphism	106
Understanding the Open-Closed Principle.....	107
Recognizing an OCP violation in account behavior • 108	
Refactoring account behavior behind an abstraction • 110	
Understanding the practical limits of OCP	112
Containing unavoidable changes in a factory • 112	
When the abstraction itself must change • 114	
Do not sacrifice performance for formal compliance • 115	
When changing existing code is the right choice • 115	

Bringing SRP and OCP together	116
Summary	117
Chapter 7: Applying the Liskov Substitution Principle	119
Technical requirements.....	120
Understanding the Liskov Substitution Principle	120
A memorable substitutability analogy • 120	
Recognizing LSP violations • 121	
When a subtype breaks the base-type contract • 121	
Extracting capabilities into interfaces • 123	
Moving shared behavior out of inappropriate subtypes • 124	
Avoiding subtype checks in client code • 124	
When a type should not implement the broader interface • 126	
Avoiding weakened postconditions and strengthened preconditions • 126	
Summary	129
Chapter 8: Applying the Interface Segregation Principle	131
Technical requirements.....	132
Understanding the Interface Segregation Principle	132
Thinking of an interface as a service boundary • 132	
When an interface forces meaningless behavior • 133	
Splitting the interface by capability • 134	
How ISP relates to LSP and SRP • 136	
Avoiding interface pollution • 137	
Summary	138
Chapter 9: Applying the Dependency Inversion Principle	139
Technical requirements.....	140
Understanding the Dependency Inversion Principle	140
Replacing a concrete delivery service with an abstraction • 140	
Keeping abstractions independent from implementation details • 143	
Dependency Inversion and Dependency Injection are not the same • 144	
Creating dependencies only when runtime data is available • 146	
Refactoring a concrete data-provider dependency • 148	
Refactoring a design that violates several SOLID principles	150

Understanding the code • 150	
Identifying the issues • 151	
Improving the code • 155	
<i>Isolating file access behind an abstraction • 155</i>	
<i>Moving report behavior to the objects that own the data • 157</i>	
Summary	160
Chapter 10: Static methods and dependencies	161
Technical requirements.....	162
Static methods at a glance.....	162
Deciding when a method should be static.....	163
Private methods • 163	
Why public static methods need more caution • 163	
When public static methods make sense • 167	
Wrapping static framework APIs • 167	
Keeping the dependency graph visible	169
Why hidden dependencies create fragile code • 171	
Making dependencies explicit.....	173
Summary	175
Chapter 11: Designing smaller classes	177
Technical requirements.....	178
Reviewing class-design fundamentals	178
Public and private members • 178	
Fields and properties • 179	
Inheritance • 179	
Understanding the dangers of large classes. God class antipattern	179
Recognizing when a large class is acceptable	180
Avoiding helper classes that only hide size • 181	
Avoiding superficial ways of making code shorter	181
Using class names to identify responsibilities	182
Compound names reveal multiple jobs • 182	
Incomplete names hide responsibilities • 183	
Vague names are a warning sign • 183	
Finding boundaries through data and dependencies	184

Separating personal data from account data • 185	
Grouping behavior that changes together	186
Extracting billing behavior from customer data • 186	
Separating levels of abstraction	187
Moving string-cleaning details out of blog-post storage • 187	
Avoiding unnecessary class proliferation	189
Refactoring case study: Decomposing PizzaGenerator	190
Understanding the code • 190	
Identifying the issues • 193	
Improving the code • 193	
Summary	197

Chapter 12: Organizing classes and projects 199

Technical requirements.....	200
Ordering class members for readability	200
Organizing classes into separate files	201
Structuring project files and folders	202
Maintaining consistency within classes	203
Refactoring project and class structure	205
Summary	206

Chapter 13: Balancing coupling, cohesion, and reuse 207

Technical requirements.....	208
Understanding and reducing coupling between classes	208
Applying the Law of Demeter	211
Designing for high cohesion.....	212
Applying the DRY principle to knowledge.....	215
Knowing when duplication is justified.....	217
Avoiding premature abstraction • 219	
Deferring design decisions behind abstractions • 220	
Composition and inheritance.....	220
Understanding both mechanisms • 220	
Issues of inheritance • 221	
Composition over inheritance • 224	
Refactoring case study: Reducing coupling in VacationPlanner	226

Understanding the code • 226	
Identifying the issues • 230	
Improving the code • 230	
Summary	234

Chapter 14: Using comments effectively **235**

Technical requirements.....	236
Why comments should not replace clear code	236
Outdated and misleading comments • 236	
Removing comments that restate the code • 237	
Removing navigation comments • 237	
Replacing block-description comments with methods • 238	
Keeping change history in version control • 239	
Removing commented-out code • 239	
When comments add value	239
Documenting complex algorithms • 240	
Explaining regular expressions • 240	
Documenting temporary workarounds • 241	
Using TO-DO comments carefully • 242	
Using structured documentation comments • 242	
Keeping required copyright comments concise • 244	
Summary	244

Chapter 15: Writing testable code and clean tests **245**

Technical requirements.....	246
Understanding unit testing	246
Using mocks for isolated tests • 248	
What unit tests do and do not prove • 250	
Using tests to protect and improve code quality	250
Using tests as a safety net for refactoring • 250	
Using testability as design feedback • 251	
Using test-driven development	252
Treating test code as production code	253
Writing readable and well-named tests • 253	
Separating arrange, act, and assert • 254	

Keeping tests focused • 255	
Removing duplication with parameterized tests • 255	
Writing meaningful assertions • 255	
Designing production code for testability	256
Avoiding static dependencies that must be replaced • 257	
Depending on abstractions rather than concrete types • 257	
Injecting dependencies instead of creating them internally • 259	
Keeping constructors simple • 259	
Refactoring case study: Cleaning Sorter and its tests	262
Understanding the code • 262	
Identifying the issues • 265	
Improving the code • 266	
Refactoring the overgrown test • 267	
Adding missing behavior checks • 268	
Summary	270
 Chapter 16: Unlock your exclusive benefits	 271
Unlock this book's free benefits in 3 easy steps	272
 Other Books You May Enjoy	 276
 Index	 279

Preface

Code is read far more often than it is written. A solution may produce the correct result today, yet still become expensive and risky to maintain if its intent is unclear, its responsibilities are tangled, or its design makes change difficult. **Clean code** addresses these problems by making software easier to understand, test, extend, and refactor.

Pragmatic Clean Code takes a practical approach to improving everyday code. Rather than presenting clean code as a rigid collection of rules, the book explains the reasoning behind each practice, shows the problems that arise when the practice is ignored, and demonstrates how to improve existing code through focused refactoring. The case studies allow you to compare an initial implementation with a cleaner version. You'll see each refactoring step as it happens, along with the reasoning behind it."

The examples are written in C#, but the underlying ideas apply broadly to other programming languages, including Java, C++, JavaScript, and Python. The book does not teach programming from the beginning. Instead, it assumes that you can read basic code and focuses on helping you make that code clearer, more maintainable, and easier to test.

The book begins with the foundations of clean code and progresses from names, methods, and formatting through the **SOLID** principles, static dependencies, class design and organization, coupling, cohesion, composition, comments, and testability. Throughout the book, concrete examples and refactoring case studies connect individual coding choices to the wider design of a software system.

From the author

I remember being a junior developer, staring at code I couldn't understand and quietly assuming something was wrong with me, terrified I'd chosen the wrong career. It took years to realize that wasn't quite it: the problem was the code itself, and even today, with over a decade of experience as a programmer and technical lead, I would have exactly the same problem with that same code. That realization changed how I think about clean code entirely. It is not an academic exercise; it is a form of respect for whoever reads the code next. These days, nothing makes me prouder of my code than watching someone junior, someone who has never seen it before, sit down, read it, and just get it. This book exists because I want every reader to feel that pride too.

Who this book is for

This book is for software developers who know the basics of at least one programming language and want to improve the clarity, maintainability, and testability of their code. It will be especially useful to junior and mid-level developers who are building strong coding habits, as well as experienced developers, technical leads, and code reviewers who want a practical framework for evaluating and improving existing code.

The code examples use C#, but advanced C# knowledge is not required. Familiarity with methods, classes, interfaces, inheritance, and basic object-oriented programming will help you follow the later chapters. Readers working primarily with Java, C++, JavaScript, Python, or another general-purpose language can still apply the principles, even when the language syntax differs.

What this book covers

Chapter 1, Understanding clean code, establishes the vocabulary and principles used throughout the book. It examines why clean code matters, how poor code-quality decisions create **technical debt**, what professionalism means in software development, and how AI-assisted coding changes the need for careful review and judgment.

Chapter 2, Meaningful names, shows how names communicate intent and influence the readability of a codebase. It covers naming conventions, semantic correctness, consistency, context, abbreviations, misleading names, the **Boy Scout rule**, and a naming-focused refactoring case study.

Chapter 3, Writing better methods, explores method signatures, parameter design, method size, single-purpose methods, levels of abstraction, ordering, independence, and **pure functions**, concluding with a multi-stage method-refactoring case study.

Chapter 4, Formatting code, examines how consistent formatting makes structure easier to recognize. It covers indentation, braces, line breaks, vertical organization, formatting tools, and a practical formatting refactoring.

Chapter 5, Applying the Single Responsibility Principle, introduces SOLID and focuses on class responsibilities, the one-reason-to-change formulation, focused types, and the difference between one responsibility and one method.

Chapter 6, Applying the Open-Closed Principle, explains how abstractions and **polymorphism** can contain expected variation. It also examines factories, changing contracts, performance trade-offs, bug fixes, and other practical limits of extension-oriented design.

Chapter 7, Applying the Liskov Substitution Principle, presents inheritance as a behavioral contract and examines invalid subtype behavior, runtime type checks, **capability interfaces**, and strengthened subtype preconditions.

Chapter 8, Applying the Interface Segregation Principle, shows how client-focused interfaces avoid meaningless operations and unnecessary dependencies by splitting broad contracts into smaller capabilities.

Chapter 9, Applying the Dependency Inversion Principle, shows how to replace concrete infrastructure dependencies with abstractions, distinguishes **dependency inversion** from **dependency injection**, uses factories for runtime creation, and concludes with a multi-principle SOLID refactoring case study.

Chapter 10, Static methods and dependencies, explains when static methods are appropriate, why public static dependencies and global state can create hidden coupling between components, and how to restore substitutability and testability.

Chapter 11, Designing smaller classes, focuses on class responsibilities and decomposition. It explains the **god class antipattern**, justified large classes, ineffective ways to make classes look smaller, and how names, data usage, change patterns, and abstraction levels reveal natural class boundaries.

Chapter 12, Organizing classes and projects, covers the physical and navigational organization of a codebase, including class-member order, one-class-per-file decisions, folder structure, consistency, and the trade-offs involved in reorganizing larger projects.

Chapter 13, Balancing coupling, cohesion, and reuse, examines loose coupling, the **Law of Demeter**, cohesion, **DRY**, appropriate duplication, **YAGNI**, **KISS**, inheritance risks, and composition as a flexible alternative.

Chapter 14, Using comments effectively, distinguishes comments that add necessary context from comments that merely restate or contradict the code. It covers the kinds of comments worth removing, such as outdated comments, changelists, and commented-out code. It also covers the cases where a comment earns its place: non-obvious algorithms, regular expressions, workarounds, XML documentation, and required notices.

Chapter 15, Writing testable code and clean tests, connects code quality with unit testing and testability. It introduces **unit tests** and **mocks**, explains how testing supports safer change and better design, and addresses readable tests, dependency injection, static-method pitfalls, simple constructors, and a final code-and-tests refactoring case study.

To get the most out of this book

You should be comfortable reading basic source code and working with common programming constructs such as variables, conditionals, loops, methods, and classes. The later chapters assume a basic understanding of object-oriented concepts, but important ideas are reviewed when they become relevant.

Throughout this book, you'll find *refactoring case studies* real, messy code that we'll work through together, spotting what has gone wrong and gradually making it better. Before reading the analysis of each one, try sitting with the example yourself first. See what problems you notice, attempt your own refactoring, and only then compare your solution with the one in the repository.

You do not need a local development environment to benefit from the explanations and refactoring discussions in this book. If you want to compile the examples, run the tests, and work through the refactoring case studies yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Git, if you want to clone the repository rather than download it as a ZIP archive.
- Enough disk space to download, build, and run the example solution and its tests.

No specific operating system is required; use any environment that supports .NET 10 and your chosen IDE.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>. If there is 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!

Download the color 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/gbp/978-1-80849-379-9>.

Conventions used

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

CodeInText: Indicates code words in text, class and method names, folder names, filenames, file extensions, pathnames, dummy URLs, and user input. For example: "The `GetAge` method belongs to the `Person` class."

A block of code is set as follows:

```
public int GetAge()
{
    return DateTime.Now.Year - YearOfBirth;
}
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example: "Select **Code**, and then select **Download ZIP**."

Note

Warnings or important notes appear like this.

Tip

Tips and useful recommendations 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 or have any general feedback, please email us at customer@packt.com and mention the book's 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 reported this to us. Please visit <http://www.packt.com/submit-errata>, click **Submit 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 <http://authors.packt.com/>.

Share your thoughts

Once you've read *Pragmatic Clean Code*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



<https://packt.link/r/1808493796>

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

Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the *"How to Unlock"* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



DRM-Free PDF Version

Download DRM-free PDF and ePub copies of this book.



7-Day Packt Library Access

Get 7-day unlimited access to 8,000+ books and videos. No credit card required.

Available for first-time Packt+ trial users only.



Next-Gen Reader Access

Read this book on Packt Reader with progress sync, dark mode and note-taking.

How to Unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require one

Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.



1

Understanding clean code

Working code and **clean code** are not the same thing. Code is clean when it can be *read*, *understood*, *tested*, and *changed* without undue effort or risk. Those qualities decide how fast a team can deliver features, how confidently it can fix defects, and how well a system copes as its requirements grow. This chapter establishes the foundation for the rest of the book.

We will cover the following topics:

- Essential vocabulary
- Why clean code matters to projects and businesses
- The cost of poor code quality and technical debt
- The definition of done
- Professional responsibility for code quality
- The characteristics of readable and maintainable code
- A comparison of clean and poorly written code
- Clean code in the age of AI-assisted development
- A practical approach to the refactoring case studies used in this book

By the end of this chapter, you will be able to explain how code quality affects software delivery, distinguish working code from clean code, describe the definition of done and technical debt, and recognize the role of clean code practices when working with AI-assisted development.

Note

Download the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Essential vocabulary

Before discussing clean code practices, it helps to review a few general programming concepts referenced throughout the book. These terms are not specific to clean code, and readers with programming experience may already be familiar with them. The table below is a quick reference, not an introduction to programming.

Term	Meaning
Clean code	Code of high quality that is easy to read, understand, and modify.
Refactoring	Improving the quality of code without changing its behavior. This may include renaming variables or functions, extracting smaller classes from larger ones, and similar changes.
Function / method	A reusable piece of code designed to perform a specific task. Functions can be called with arguments and may return a value; if not, they are called void functions. A function that belongs to a type is usually called a method.

Term	Meaning
Static method	A method that belongs to a class as a whole, not to any specific instance, and therefore does not have access to instance data such as fields or properties.
Inheritance	A relationship in which a derived class inherits properties and methods from a base class, allowing code reuse and hierarchical relationships between classes. A derived class can extend or modify the inherited behavior.
Virtual method	A method that can be overridden in derived types. The base type defines default behavior, and a derived type can provide its own implementation.
Abstract method	A method similar to a virtual method, but without a body in the base type. It must be overridden in derived, non-abstract types. A class containing an abstract method must itself be abstract and cannot be instantiated.
Lambda expression	A short, anonymous function used to define simple operations. In C#, lambda expressions are written with the arrow operator, with parameters on the left and the function body on the right.
Bug	An error in code that causes it to produce incorrect or unexpected results, or to behave in unintended ways.
Exception	A runtime error that disrupts the normal flow of execution, typically handled with mechanisms such as try-catch blocks. Exceptions are usually thrown when an operation cannot be completed, such as reading from a file that doesn't exist.
Unit test	An automated test that checks whether a specific piece of code works as expected.
Mock	An object that simulates the behavior of another object in a controlled way, often used in unit tests to replace dependencies such as a database.
Code review	The process of checking the correctness and quality of code by another programmer.
Version control system	A tool that tracks, manages, and stores changes to code over time, enabling collaboration and version tracking. Git is a common example.

Term	Meaning
Code repository	Centralized code storage managed by a version control system, where developers store, manage, and track changes to their codebase.
IDE	An Integrated Development Environment: a program that typically provides a code editor, debugger, and build tools to support writing, testing, and debugging code. Examples include Visual Studio, PyCharm, and IntelliJ.
Library	A collection of pre-written code that developers can reuse to perform common tasks without writing them from scratch.

Table 1.1: Essential vocabulary

Why clean code matters

The effect of **code quality** is easiest to understand through the life of a project. Consider a software company beginning a new application for a client. The team starts with a clean slate, and the first version they deliver is small and quick to ship. The code is covered by tests, defects are rare, and both the company and the client are pleased.

A project that starts well

As development continues, simple classes become larger and relationships between components *become more complicated*. The team notices areas that should be refactored, but a release deadline is approaching. Because the application still works, the cleanup is postponed. The next version is delivered, although a few minor defects appear during the demo.

By the following release, every change takes longer. Developers must trace unfamiliar dependencies, understand old decisions, and work around code that no longer fits the growing system. Test coverage becomes harder to maintain. The team again intends to refactor, but the pressure to deliver new features leaves little time for it.

Eventually, a release arrives late with missing features and serious defects. The client expects the team to recover while also delivering the next set of features. Management compares current productivity with the unusually fast first release and assumes that the same pace should continue. The team responds by working longer hours and reducing the time spent on tests, reviews, and refactoring. This creates more defects and makes the next release even harder.

Note

Key idea: How fast a team builds the first version says little about its long-term delivery speed. As the codebase grows, part of the team's time must go into testing and refactoring to keep existing behavior intact.

Complexity requires continuous maintenance

A software project is similar to a construction site. At the beginning, the area is clear and the work can progress quickly. As construction continues, unused tools, packaging, and debris accumulate. If nobody removes them, workers must step around the clutter and every task becomes slower and less safe.

A growing **codebase** accumulates its own form of clutter: duplicated logic, obsolete abstractions, inconsistent names, tightly coupled components, and tests that no longer reflect the current design. Complexity does not disappear by itself. Regular refactoring keeps it manageable and protects the team's ability to deliver new work.

The cost of poor code quality

The project in the previous section did not fail because of *one badly written method*. It failed because a sequence of short-term decisions gradually removed the team's ability to work safely, while developers and management quietly disagreed about what it actually meant for the work to be done. The most common consequences reinforce one another.

Decision	Short-term effect	Long-term consequence
Skip refactoring	More time appears available for the current feature.	Future changes require more analysis and rework.
Reduce automated testing	The release may be completed sooner.	Defects are discovered later and regressions become harder to diagnose.
Accept unclear code	The developer avoids spending time on names and structure.	Other developers need longer to understand the implementation.
Hide quality concerns	An uncomfortable conversation is postponed.	Management plans against an unrealistic delivery rate.

Decision	Short-term effect	Long-term consequence
Rely on overtime	A deadline may be met once.	Burnout, turnover, and knowledge loss reduce future capacity.

Table 1.2: Short-term code-quality decisions and their long-term consequences

Working code is not finished code

A common mistake is to treat a feature as complete as soon as it produces the expected result.

Working code and **good code** are not the same. Code is a form of *communication*: it records the current behavior of a system and explains that behavior to the people who will maintain it. That puts it in the same category as any other document written to be read. We would not call a user manual finished if it held all the right information in a confusing, repetitive, disorganized form. Code deserves the same standard.

Every project should have its own **definition of done**: A shared set of criteria that determines when a piece of work is actually finished, not just functional. It prevents quality activities from being treated as optional work that can be removed whenever a deadline becomes uncomfortable. The exact criteria depend on the organization and the risk of the system, but a practical definition often includes the following:

- The implementation satisfies the *agreed acceptance criteria*.
- Relevant unit, integration, and end-to-end tests pass.
- The code has been reviewed by the required number of developers.
- The implementation follows the project's coding and design conventions.
- Required documentation has been updated.
- The change has been merged into the shared repository and can be built successfully.

A task is finished only when the agreed conditions are met. This does not eliminate every defect, but it makes quality visible and protects the team from repeatedly trading long-term stability for short-term appearance.

Understanding technical debt

Every team eventually faces a choice between the *fast way* and the *right way*. That trade-off has a name borrowed from finance: **technical debt**.

Technical debt describes the future cost created when a team accepts a solution that is faster or easier now but more expensive to maintain later. The analogy with financial debt is useful because both provide an immediate benefit and both create an obligation.

Technical debt	Financial debt
The team accepts a lower-quality design or postpones necessary cleanup.	A person or organization borrows money.
Delivery becomes easier in the short term.	Spending becomes easier in the short term.
The code must later be refactored, retested, or redesigned.	The borrowed amount must later be repaid.
The longer the debt remains, the more areas depend on it and the harder repayment becomes.	Interest increases the total cost of repayment over time.

Table 1.3: Comparing technical debt with financial debt

Technical debt can create false confidence about *delivery speed*. If a team repeatedly skips cleanup to meet dates, management and the client may assume that the resulting pace is sustainable. Future plans are then based on a speed that exists only because necessary work has been deferred. When the debt finally has to be repaid, the team appears to slow down even though it is addressing work that was always part of the real cost.

This creates a **deadline trap**. The team is given more work because earlier features appeared to arrive quickly. The increased workload leaves less time for refactoring, which causes the debt to grow. Each release becomes harder, and the pressure to preserve the old delivery rate becomes stronger.

Note

LeBlanc's law: Later equals never. Work that isn't done now, when the context and motivation are fresh, tends to keep getting deprioritized until it's effectively abandoned. A large refactoring may not be practical immediately before a release, but once the need is identified, it should be recorded and scheduled. Unplanned debt tends to remain until it becomes a crisis.

The professional response is not to insist that every imperfection be fixed immediately. It is to make the compromise visible, understand its risk, and agree on when and how it will be addressed. Debt that is acknowledged and managed is very different from debt that quietly becomes the normal design of the system.

Professional responsibility and code quality

It is tempting to place all responsibility for poor quality on project management, *especially when developers have asked for time to refactor*. However, developers usually understand the technical consequences more clearly than anyone else involved in planning. That knowledge creates a responsibility to communicate **risk** in terms that non-technical stakeholders can use.

Estimates should include the full definition of done. If an estimate accounts only for writing the first version of the code, management may promise a date that leaves no time for tests, review, documentation, or integration. Once that promise is made, quality work is incorrectly presented as additional scope rather than part of completing the feature.

Effective communication focuses on outcomes instead of jargon. Saying that a module has technical debt may be meaningful to another developer, but a project manager may need to understand the practical effect: changes in this area take longer, defects are more likely, and each new feature increases the cost of future work.

Step	How to communicate it
Describe the condition	Identify the affected area and explain what makes changes difficult or risky.
Explain the impact	Connect the condition to delivery time, defect risk, support cost, or customer experience.
Present options	Offer realistic choices, such as a smaller feature scope, a scheduled refactoring task, or additional testing time.
Record the decision	Document the accepted trade-off so that the debt and its consequences remain visible.

Table 1.4: Communicating code-quality concerns to stakeholders

Note

Professionalism: A developer cannot control every business decision, but can ensure that decisions are made with accurate information about quality, risk, and the real cost of delivery.

What makes code clean?

Clean code is written for people. The computer can execute code with poor names, inconsistent formatting, and unnecessary complexity. Developers, however, must read that code to fix defects, add behavior, review changes, and understand why the system works as it does.

A deliberately informal measurement

One humorous way to describe readability is to count the number of "**WTF**" moments a reader experiences per minute. The expression is intentionally informal, but the underlying observation is useful: frequent surprise and confusion are signals that the code is hiding its intent.

$$\text{Cleanliness indicator} = \text{confusing moments} / \text{minute}$$

The important part is that the measurement is based on reading, not writing. Developers typically spend more time understanding existing code than typing new code. Even when you are the only developer on a project, you may return to the code several months later without the context you had when you originally wrote it. **Readability**, therefore, is not a stylistic preference; it directly affects the time and effort required to understand and maintain the code in the future.

Practical characteristics of clean code

Characteristic	What it looks like in practice
Readable	Names, layout, and structure help the reader follow the code without decoding it.
Clear in intent	The reason for each operation is clear from the code and is not complicated by vague names or surprising side effects.
Simple	The implementation avoids unnecessary abstraction, cleverness, and branching.
Focused	Methods and classes have a clear responsibility and operate at a consistent level of abstraction.
Low in duplication	A business rule has one authoritative implementation rather than several copies that can diverge.

Characteristic	What it looks like in practice
Predictable	Behavior matches the names, interfaces, and conventions used by the project.
Testable	The code's behavior can be verified in isolation, without relying on complex setup or hidden dependencies.
Easy to change	A developer can locate the relevant code, understand the effect of a change, and modify it with limited risk.

Table 1.5: Practical characteristics of clean code

There is no single visual form that all clean code must follow. Two implementations can use different structures and still be high quality. The goal is not to imitate one preferred style mechanically, but to make deliberate choices that reduce the reader's effort and preserve the system's ability to change.

Writing clean code is a skill developed through practice. Experienced developers still revise names, split responsibilities, and discover better structures. Refactoring is therefore not evidence that the original developer failed; it is part of adapting the code as understanding improves.

Comparing clean and poorly written code

The difference between working code and clean code becomes clearer with two implementations of the same behavior next to each other. Both versions of the method below authorize a user to perform an action, and both work. The first makes the reader decode abbreviated names, follow a manual loop, and work out what each block is for. The second one does not.

Hard to read version:

```
public AuthorizationResult Auth(
    User u, UserAction ua)
{
    if (_usersRepository
        .GetById(u.Id) == null)
    {
        throw new
            NonExistentUserException(u);
    }
}
```

```
var perm = _permissionsRepository
    .GetForUser(u.Id);
if (u.IsSuperUser)
{
    return AuthorizationResult
        .Authorized;
}
foreach (var p in perm)
{
    if (p.Action == ua)
    {
        return AuthorizationResult
            .Authorized;
    }
}
return AuthorizationResult
    .NotAuthorized;
}
```

Refactored version:

```
public AuthorizationResult Authorize(
    User user, UserAction action)
{
    if (!_usersRepository
        .Exists(user))
    {
        throw new
            NonExistentUserException(user);
    }

    if (user.IsSuperUser ||
        IsUserAuthorizedToPerform(
            action, user))
    {
        return AuthorizationResult
            .Authorized;
    }

    return AuthorizationResult
        .NotAuthorized;
}
```



```
}

private bool IsUserAuthorizedToPerform(
    UserAction action, User user)
{
    var userPermissions =
        _permissionsRepository
            .GetForUser(user.Id);
    return userPermissions.Any(
        permission =>
            permission.Action == action);
}
```

The first method eventually reveals its purpose, but several details slow the reader down. The name `Auth` does not say whether the method *authenticates* or *authorizes*. The parameters `u` and `ua` require mental translation. The permissions are retrieved before the super-user check, even though a super user does not need them. Furthermore, the loop asks a simple question—whether any permission matches the requested action—in a fairly verbose way.

The refactored version makes the same decision while exposing its intent more directly.

The name `Authorize` describes the operation, and the parameter names identify the information involved. The main method now reads as a sequence of business decisions: confirm that the user exists, then authorize a super user or a user with the required permission. The details of permission lookup are moved into a focused method whose name explains the question it answers.

Concern	Hard-to-read version	Refactored version
Method intent	<code>Auth</code> is ambiguous.	<code>Authorize</code> states the purpose.
Parameters	<code>u</code> and <code>ua</code> require interpretation.	<code>user</code> and <code>action</code> are explicit.
Control flow	Permission retrieval occurs before it is always needed.	The super-user path can complete without unnecessary work.
Levels of abstraction	Repository access and matching logic are mixed into the main method.	Permission matching is contained in a named helper method.

Concern	Hard-to-read version	Refactored version
Collection check	A loop manually searches for a matching permission.	Any method expresses the existence check directly.

Table 1.6: Improvements introduced by the authorization refactoring

Note

Both versions may pass the same tests. What distinguishes clean code is not its output, but *how easily a developer can verify, explain, and change it*.

Clean code in the age of AI-assisted development

AI coding assistants can generate, explain, and refactor code quickly. This raises a reasonable question: if an AI tool can improve a messy program, do developers still need to learn clean code? *Yes, definitely.* AI can accelerate the work, but a developer must still decide whether the result is correct, appropriate for the system, and maintainable.

When given a hard-to-read method, an AI assistant may improve names, format the code consistently, and simplify redundant logic. It may also add comments to describe each block. Some of these changes are useful; others require judgment. For example, AI tools often add comments that restate what the code already says. Comments like these add nothing, and they go stale as soon as the code around them changes. A well-named method *communicates the same intent and stays accurate*.

The assistant may also *miss* opportunities that depend on domain knowledge or project conventions. We must also be aware that another run with the same prompt may produce a different result. Generated code is therefore a proposal, not an approved implementation.

AI can often help with	A developer must still verify
Suggesting clearer names and formatting	Whether the names match the domain language used by the project
Extracting small methods or simplifying local control flow	Whether the new structure preserves behavior and fits the surrounding design
Generating an initial set of unit tests	Whether the tests cover important cases and assert the correct behavior

AI can often help with	A developer must still verify
Explaining unfamiliar code	Whether the explanation reflects the actual runtime behavior and dependencies
Proposing alternatives for a small, isolated component	Security, performance, architecture, business rules, and effects on the wider codebase

Table 1.7: Appropriate uses and limitations of AI-assisted coding

Large language models generate responses from patterns learned during training and from the context supplied in the prompt. Today, that context can be substantial—an entire codebase, relevant documentation, prior commits. But even with that much information available, AI tools still don't have the full picture: the history behind a design decision, the business rules that were never written down, or the reasoning a teammate had in mind but never documented. They can also make mistakes or unintentionally alter behavior while attempting a refactoring.

Note

Review rule: Treat AI-generated code as *unreviewed code*. Read it, test it, compare its behavior with the original implementation, and evaluate it against the same standards you would apply to a colleague's contribution.

AI tools are especially useful for small, distinct tasks, and they can handle larger ones too, provided they're given enough information about the system to work with. They can help draft tests, explain an unfamiliar API, suggest names, propose a local refactoring, or take a first pass at untangling a messier piece of code. None of that removes the need to understand clean code yourself. Writing sloppy code on the assumption that an AI tool will clean it up later just moves the risk downstream—the tool may miss a business rule that was never written down, or refactor the code in a way that quietly changes its behavior. Clean code remains your responsibility, not a problem to outsource.

The tools and their capabilities will continue to change. The durable skill is the ability to recognize clear design, identify defects in a proposed solution, and improve code deliberately. AI can reduce the effort required for some steps, but it cannot remove the need for technical judgment.

Working with the refactoring case studies

The remaining chapters include refactoring case studies that show how the clean code principles in this book translate into practical, hands-on work. Each case study starts with code that has problems—with *readability*, *design*, or *testability*. The goal is to understand those problems and rework the code until it's clean and well-structured. You can choose how you want to work through the case studies: reading through the analysis is valuable on its own, but you'll get the most out of it by trying the case study yourself first, then comparing your solution with the one presented. If you'd like to try the case studies yourself, use the following workflow:

1. Read the original code before looking at the proposed solution. Describe what it does in your own words.
2. Identify the specific source of difficulty, such as unclear naming, duplication, mixed responsibilities, or hidden dependencies.
3. Refactor in small steps. After each meaningful change, compile the code and make sure it still works.
4. Compare the before and after versions. Explain which reader questions became easier to answer and which risks were reduced.
5. Consider alternatives. Clean code principles guide decisions, but they do not require every developer to produce an identical design.
6. Compare your solution with the one proposed in the book.

Note

The case studies in this book do not come with a pre-written test suite, and that is partly the point. Code that has not been designed with care is often hard to test, or impossible, which is one of the clearest signals that something needs restructuring. If you want extra practice, try writing tests for the original code before refactoring it. Some behavior will be difficult to reach until the structure changes. Writing the tests after a first round of refactoring is a reasonable alternative, and it is what usually happens in real projects working with legacy code.

The accompanying repository includes the code used in the case studies. Keeping the original and refactored versions available allows you to inspect the changes directly, experiment with alternatives, and restore a known starting point when necessary.

Summary

This chapter established the practical foundation for the rest of the book. You saw how code quality affects delivery speed, defects, maintenance effort, and the ability to make future changes; why working code is not necessarily finished code; and how technical debt grows when quality work is repeatedly deferred. You also considered professional responsibility, compared hard-to-read and refactored code, and examined how AI-assisted development still requires human judgment.

In *Chapter 2*, you will focus on one of the most visible ways code communicates intent: meaningful names.

2

Meaningful names

Poor names make otherwise correct code expensive to read. A method may work perfectly and still force every future reader to decode abbreviations, inspect implementations, and reconstruct the original developer's intent. Because names appear everywhere in a codebase, the cost of unclear naming compounds quickly as a project grows.

The solution is to make names communicate purpose honestly and precisely, follow the conventions of the language and codebase, and rename things as understanding improves. This chapter develops those habits through practical examples and a complete naming-focused refactoring case study.

In this chapter, we will cover the following topics:

- Why meaningful names matter
- Naming fundamentals for variables, types, methods, and Boolean values
- Language and project naming conventions
- Why developers choose poor names and how to evaluate names objectively
- Renaming and the Boy Scout rule
- Writing expressive names and choosing an appropriate name length
- The principle of least surprise and semantic correctness
- Avoiding homonyms, inconsistent synonyms, confusing names, and unnecessary abbreviations
- Matching singular and plural forms and using context effectively
- Avoiding meaningless words, overly specific names, and Hungarian notation
- Refactoring case study: improving names in a tic-tac-toe implementation

By the end of this chapter, you will be able to choose meaningful names for variables, parameters, types, methods, and Boolean values; follow language and team conventions; identify misleading, ambiguous, overly long, or overly specific names; use context to remove unnecessary repetition; and refactor code when naming problems reveal deeper design problems.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Why meaningful names matter

Programming involves naming things all the time: variables, methods, parameters, classes, and files. Much of what we read in code consists of names that somebody chose. And remember, *code gets read more often than it gets written*. That's why readability is so crucial.

The listing below is highlighted to show every name its author chose. Those names alone make up most of the code.

```

public GameResult Play()
{
    var diceRollResult = _dice.Roll();
    Console.WriteLine(
        $"Dice rolled. Guess what number it shows in {InitialTries} tries.");

    var triesLeft = InitialTries;

    while (triesLeft > 0)
    {
        var guess = ConsoleReader.ReadInteger("Enter a number:");

        if (guess == diceRollResult)
        {
            return GameResult.Victory;
        }

        Console.WriteLine("Wrong number.");
        --triesLeft;
    }

    return GameResult.Loss;
}

```

Figure 2.1: Highlighted names in a C# method

If we also highlighted the names that come from the .NET standard library, almost nothing here would be left untouched.

All of those things could have been named differently, and different names would change how easy the code is to read. That is why *good names are the foundation of readable code*. They deserve real thought before anything gets settled. It is also why we devote this entire chapter to naming.

Now let's compare two implementations of the same filtering operation. Read the first version and try to understand what it does as quickly as possible.

A working implementation with weak names:

```

public List<string> Filter(int l, List<string> items)
{
    var res = new List<string>();
    foreach (var i in items)
    {
        if (i.Length < l)
        {
            res.Add(i);
        }
    }
}

```



```
    }  
    }  
    return res;  
}
```

This version works, but it is not obvious at first glance what this code does. Renaming alone exposes the intent of the method without changing its behavior:

The same implementation with meaningful names:

```
public List<string> GetWordsShorterThan(int length, List<string> words)  
{  
    var result = new List<string>();  
    foreach (var word in words)  
    {  
        if (word.Length < length)  
        {  
            result.Add(word);  
        }  
    }  
    return result;  
}
```

How much time did you need to read and understand the method now? Reading the signature alone is enough to know what it does: given a list of strings, it returns only the words shorter than a given length. Even with the signature hidden, the body would be easy to follow. It iterates over a collection of words and adds each word shorter than the given length to the result.

Finding good names takes effort at every level of experience. *The payoff is code that can be picked up and understood a year later.* Time spent thinking about names is repaid every time that code is read.

Naming fundamentals

We will now explore the fundamental rules of good naming. Most of them hold regardless of the programming language.

Naming variables, parameters, and types

The first rule is to match the grammatical form of a name to the kind of thing it represents.

Let's start with variables, parameters, and fields. With a few exceptions covered later, they should be named with nouns: person, file, or player. Modifiers can be added when a bare noun is not specific enough, as in `textFile`, `databaseConnection`, or `userToBeUpdated`, but the core of the name stays a noun.

Types follow the same rule as variables: they should be named with nouns, such as `Point`, `List`, `PeopleDataReader`, or `UserAuthorizer`. Here, "type" means any user-defined type, including classes, structs, enums, and interfaces. Most style conventions name types with an initial capital letter and name variables and parameters with a lowercase one.

Two of those examples end in *-er*. That suffix is conventional for types whose job is to do something rather than to carry data, and it states that job directly in the name.

Naming methods

The names of methods should be built around verbs. A method does something, and that action belongs in the name. Examples of typical method names could be `Read`, `Transform`, or `Load`. Longer names work the same way, with a verb at the core: `DecreaseBalance`, `CountNegativeNumbers`, `RemoveExcessSpaces`.

A special case is a word that works as both a noun and a verb, such as `request`. When the context makes it clear whether the name refers to a method or to a value, this is rarely a problem. When it does not, the fix is to make the name more specific: `SendRequest` for the method, `request` for the variable.

Naming Boolean values and methods as questions

Booleans are a special case, both as variables and as methods that return them. They should not be named with nouns or verbs. A Boolean name should read as a question that can be answered with yes or no: `isShorterThanFiveLetters`, `hasBeenSavedSuccessfully`, `canBeDeleted`.

Boolean names should also be positive. `isEnabled` is a good name; `isNotEnabled` is not. Negation belongs in the code that reads the value, not in the name, and the logical negation operator is there for that.

```
if (user.IsEnabled) { ... }
if (file.HasBeenSavedSuccessfully) { ... }
if (account.CanBeDeleted) { ... }

// Prefer this:
if (!user.IsEnabled) { ... }
```

```
// Over this:
if (user.IsEnabled) { ... }
```

Construct	Useful naming pattern	Examples
Variables, parameters, and fields	Nouns or noun phrases	person, file, score, textFile, userToBeUpdated
Classes, structs, enums, and other types	Nouns or noun phrases	Point, FileReader, UserAuthorizer
Methods and functions	Verbs or verb phrases	Read, Transform, Load, DecreaseBalance, RemoveExcessSpaces
Boolean values and Boolean-returning methods	Positive questions	isEnabled, hasBeenSavedSuccessfully, canBeDeleted

Table 2.1: Common naming patterns by code construct

Following language and project conventions

Meaningful names still feel wrong when they ignore the conventions of the language. Most languages have a mainstream convention. C# uses `PascalCase` for types and methods and `camelCase` for variables and parameters. In both, every word after the first starts with a capital letter, and the two differ only in the first word: `PascalCase` capitalizes it, `camelCase` does not. Python uses `snake_case` instead, which separates words with underscores. Conventions cover more than naming, and they also describe things like indentation and line breaking.

The table below shows several conventions related to naming side by side.

Style	Example	Typical use
PascalCase	DayOfBirth	Common for type names in many languages. In C#, also used for methods, properties, and other public members
camelCase	lastName	Local variables and parameters in C#, Java, and JavaScript

Style	Example	Typical use
snake_case	list_length	Variables and functions in Python, Ruby, and Rust
SCREAMING_SNAKE_CASE	MAX_RETRY_COUNT	Constants, in almost every language that has them
kebab-case	user-profile	Lisp and Clojure identifiers, CSS class names, URLs. Impossible in most languages, where the hyphen is a subtraction operator

Table 2.2: Common capitalization styles used in source code

No style is better than the others in itself. What matters is that code looks familiar to people working in that language. Moving from C# to Python means moving to Python's conventions, not carrying C# habits along.

Conventions matter for two reasons. The first is consistency. When all developers follow the same convention, code looks the same no matter who wrote it, and consistent code is easier to read and follow.

The second reason is that conventional names carry information beyond what they name. In C#, a field whose name starts with an underscore is private, and that can be read from the name alone, without looking up the declaration. This makes reading code faster. It also makes the code easier to maintain.

Teams may add local rules on top of language conventions. Such rules should be written down somewhere the team actually reads, and they should extend the language conventions rather than contradict them. One common example is a rule that unit tests name the object being tested cut, for "class under test". Across a few thousand tests, the subject is always called the same thing, so you know what you're looking at before you read the rest.

Writing a convention down is not the same as having it followed. Tooling closes that gap. Formatters apply layout rules on save, so nobody argues about them in review. Static analyzers check naming rules and flag violations while the code is being written. Both can run in the build pipeline, which turns the convention into something the build enforces rather than something a reviewer has to remember.

Why developers choose poor names

Nobody picks bad names just to annoy other programmers. Hopefully. The problem is that a name looks fine at the moment we write it, because we know things the future reader will not. After a few hours on the same method, we know the problem, the data, and what every temporary variable holds. None of that is in the code, and none of it survives a few weeks away. Plenty of code that looked clean when it was written turns out to be unreadable a month later, to the person who wrote it.

The curse of knowledge

This effect is known as the **curse of knowledge**. Once we know something, it becomes hard to imagine not knowing it. Explanations that feel complete to us turn out to skip the steps we stopped noticing a long time ago, and we misjudge how much of what we said actually landed.

Code is where this shows up constantly. A name that reads clearly to its author reads that way because the author knows what sits behind it. Someone opening the file for the first time does not. And neither, in a sense, do we a few months later, once the context has faded.

To counter this bias, *read your own code as if you did not write it*. It helps to have a specific person in mind: a junior developer, or someone who joined the team last week and does not know the codebase yet. What would they ask about? Would they know what a given variable holds, or what a method call does, without asking anyone?

Using the rubber duck method

Another option is the rubber duck method. Its name needs some explanation.

It happens often that we get stuck on a piece of code and call a colleague over to discuss it. Then, halfway through explaining the problem, the solution appears on its own. The colleague has not said a word yet. It turns out that explaining the problem out loud was the whole point, and the listener was optional. Which means we can save a colleague's time and explain the code to a rubber duck, or to anything else that will sit still and listen.

The same technique works for readability. Explain the code to the duck, line by line. If the walkthrough needs constant asides—this variable is actually the active customer, this method is called `C1ear` but it only trims trailing whitespace—the code is not speaking for itself, and those asides point to the exact names that need work. If the code can be read as it stands, it is in good shape.

Real programmers are better than rubber ducks. Ask the most junior person on the team to read the code and listen to what they ask about. Every question marks a place where the reader needed context that the code did not give them.

Try not to take those questions personally. It is uncomfortable to hear that somebody cannot follow our code, and the easy way out is to decide the reader is not good enough. It is worth resisting that, because this feedback is the fastest way to get better. *When somebody who joined last week can read our code without asking a single question, that is as good as it gets.*

Renaming and the Boy Scout rule

Good names often arrive gradually. The first one we pick reflects what we understood about the design at that moment, and as the design becomes clearer, a better name tends to suggest itself.

So when a name refuses to come, there is no point in staring at the screen. Give the class a temporary name, write the code, refactor it, and come back to the name once the class does exactly what we want. At that point the right name is usually waiting for us.

Renaming is a normal refactoring step, not an admission that the first attempt was a failure.

There is a rule worth knowing here, and it comes from scouting rather than from programming. Scouts are told to leave the campsite cleaner than they found it, whether or not they were the ones who made the mess. Applied to software, the Boy Scout rule says to leave the code a little better than we found it.

With naming, that can mean fixing one vague variable, replacing an abbreviation, or renaming a class whose responsibility has become clearer since it was written. Small improvements like these are easier to keep up than a large cleanup that may never get scheduled. Modern IDEs rename symbols safely across a whole project, so most of the mechanical cost is gone.

Note

Boy Scout rule: *Always leave the code you are working on a little bit better than you found it.*

A large refactoring is easy to postpone, and the longer it waits, the bigger it gets. The Boy Scout rule avoids that trap. Instead of demanding perfect code, it asks for a small cleanup in whatever class or method we happen to be working on. Those small changes add up.

Renaming public APIs requires more care

Renaming is straightforward inside a codebase you control, but a public API or shared library creates compatibility obligations.

Consider `Console.WriteLine`, one of the most familiar methods in the .NET API for C# developers. It writes a line of text to the console and is commonly encountered in introductory and everyday C# code.

Now imagine its authors decided the name could be better and renamed it. Every one of those programs would stop compiling. The fix would be mechanical, but somebody would have to make it, in every codebase, before anything could be built again.

That is why public names deserve more thought than internal ones. An internal name can be changed in a second. *A public one is a commitment*, and changing it later means deprecation, migration guidance, and usually a new major version.

When naming is hard, inspect the design

Sometimes the difficulty is not that you have failed to discover a clever name. The construct itself may be doing too much. A precise name becomes awkward because it has to describe several responsibilities at once. In that situation, *the naming problem is a design signal*.

Consider a method named `ActivateAccountAndSendNotification`. The word `And` is a warning that two operations have been combined. Splitting the work produces names that are shorter, clearer, and more reusable:

```
ActivateAccount();  
SendNotification();
```

Conditional behavior can reveal the same issue. Consider a method called `EmptyShoppingCartIfSessionInactive`. Instead of hiding the condition inside a long name, separate the question from the action:

```
if (!IsSessionActive())  
{  
    EmptyShoppingCart();  
}
```

Class names can reveal blurred responsibilities as well. Consider a class called `EmailManager`. What is its job exactly? Sending emails? Or maybe creating accounts in some email service? As it turns out, it does both things:

```
class EmailManager  
{  
    public void Send() { ... }  
  
    public void Create() { ... }  
}
```

Splitting the responsibilities allows more meaningful names:

```
class EmailSender
{
    public void Send() { ... }
}

class AccountCreator
{
    public void Create() { ... }
}
```

Notice that the names became easy once EmailManager was split.

When the best name we can find for a class is Manager, Handler, or any other vague word, the difficulty is usually not in our vocabulary. Such a name is often a sign that the class itself is vague, and that it does several unrelated things at once. When a name refuses to get more specific, we should take a close look at the class and consider splitting it into smaller, more focused types.

Writing expressive names

A good name should make the programmer's intention visible without depending on explanatory comments. Consider a method called Clear that removes whitespace only from the end of a string. The implementation may be correct, but the name is too broad:

```
// Removes whitespace from the end of input.
string Clear(string input) { ... }
```

A better name carries the explanation itself:

```
string RemoveTrailingWhitespaces(string input) { ... }

// Or, using familiar string terminology:
string TrimTrailingWhitespaces(string input) { ... }
```

The same principle applies to types. A class called FileWriter should write to files, and that is all it should offer. A public method that checks whether a file exists would be out of place. Privately, of course, the class can do whatever it needs, and checking that a file exists before writing to it is perfectly reasonable. The difference is that this check is part of how the class does its job, not part of what it offers to the outside.

A good name shows what the programmer intended, and it does that precisely. Sometimes the intention comes through, but only roughly. Take a `GameObject` class with a method called `Transform` that changes the object's position. Movement is a kind of transformation, so the name is not wrong. It is just wider than it needs to be, and a reader has to open the method to find out which kind of transformation this is. Renaming this method to `Move` would make what it does obvious from the start.

Choosing an appropriate name length

Short names reduce reading effort only when they preserve meaning. If the choice is between a long, clear name and a short, unclear one, *clarity wins*. Once the meaning is secure, remove words that add no information.

For example, a filtered collection might evolve through several possible names:

```
var onlyPeopleOlderThan18Years = ...;
var peopleOlderThan18Years = ...;
var adultPeople = ...;
var adults = ...;
```

If the domain clearly defines an adult as a person older than 18, `adults` is both shorter and more expressive. If that assumption does not hold in the application, the longer version may be safer. The surrounding domain decides how much information a name must carry.

Some names are necessarily longer because the distinction matters. `GetSingleLetterStrings` is more precise than `GetShortStrings` when the method must return only strings of length one. Shortening the name would make the contract less clear.

Still, if the only name that fits is a very long one, that usually says something about the code rather than the name. A method may be doing too much, or a variable may be holding something too complicated. Take a method whose only reasonable name is `FinalizeOrderAndStartPaymentProcess`. The name is not the problem. The method is, and the fix is to split it into `FinalizeOrder` and `StartPaymentProcess`. As a rule, a name that needs "and", "or", or "if" is worth treating as a refactoring signal.

Very long names can also reveal misplaced responsibilities. Suppose `Person` contains a method called `IsHomeAddressValid`:

```
class Person
{
    public Address HomeAddress;
```

```
public bool IsHomeAddressValid() {...}  
}
```

The name is long partly because `Person` is carrying a responsibility that belongs to `Address`. Moving the validation to `Address` produces a better code:

```
class Person  
{  
    public Address HomeAddress;  
}  
  
class Address  
{  
    public bool IsValid() { ... }  
}  
  
if (person.HomeAddress.IsValid())  
{  
    ...  
}
```

The overall expression is not dramatically shorter, but each part is easier to read and the responsibility now lives with the object that owns the relevant data.

Long test names are often useful

Tests are a special case. A useful test name often needs to describe both the scenario and the expected result. When a test fails, its name becomes part of the diagnostic output, so extra length can be justified if it communicates the failure precisely.

```
Play_ShallReturnVictory_IfTheUserGuessesTheNumberOnTheFirstTry  
Play_ShallReturnLoss_IfTheUserNeverGuessesTheNumber  
Play_ShallReturnLoss_IfTheUserGuessesTheNumberOnTheFourthTry
```

It helps to agree on one naming pattern for tests and stick to it. There is no need to stretch a name beyond what the scenario requires, but shortening it should never cost the information a developer needs when the test fails.

Following the principle of least surprise

The **principle of least surprise** says that *code should behave the way a reasonable reader expects it to behave*. In other words, the readers of the code should never be surprised by what they see. In naming, that means methods and classes should do what their names say, do that thing consistently, and avoid hidden behavior that contradicts the promise made by the identifier.

Following this principle has a practical effect: code can be read and understood without opening the implementation of every method it calls. The names are enough to know what happens. Read the following method and try to understand what it does.

```
public void Run()
{
    var stringBuilder = new StringBuilder();
    var ticketDocuments = _documentsReader.Read(_ticketsFolder);

    foreach (var document in ticketDocuments)
    {
        var lines = ProcessDocument(document);
        stringBuilder.AppendLine(
            string.Join(Environment.NewLine, lines));
    }

    _fileWriter.Write(
        stringBuilder.ToString(),
        _ticketsFolder,
        "aggregatedTickets.txt");
}
```

In the example, reading the method top to bottom tells the whole story: ticket documents are read, then processed, then the aggregated result is written. If some step needs a closer look, the reader can open it. The point is that they choose to, instead of having to.

Now imagine that `Read` secretly deletes files, or `Write` silently modifies unrelated state. In that case, high-level reading becomes unreliable. *The names are useful only when the implementation honors them.*

Following the principle of least surprise boils down to this: the methods or classes should do what their names say. They should do only this thing, they should do it well, and they should not do anything else.

A method called `SaveUser` provides a simple counterexample. If it validates the user and saves only when the validation succeeds, the name overstates what the method guarantees:

```
public void SaveUser(User user)
{
    if (user.Email is not null)
    {
        _usersRepository.Save(user);
    }
}
```

Renaming it to `SaveUserIfValid` would be more honest, but the method would still combine two responsibilities. A cleaner design separates the question from the action:

```
if (IsValid(user))
{
    Save(user);
}

public void Save(User user)
{
    _usersRepository.Save(user);
}

public bool IsValid(User user)
{
    return user.Email is not null;
}
```

Variables should follow the same principle. If a value stores the first even number, `firstNumber` is misleading; `firstEvenNumber` communicates the actual meaning. Another common example is a parameter called `list` that actually holds an array. The name suggests one concrete collection type while the code provides another. That mismatch sends the reader in the wrong direction and may become even more misleading as the implementation changes.

Maintaining semantic correctness

We have already talked about the principle of least surprise, which is about the code as a whole behaving the way a reader expects. Semantic correctness is the same idea narrowed down to a

single name. It means the meaning a name carries matches what the code behind it actually does.

The reason this deserves its own treatment is that certain words carry stronger meaning than others, and they carry it across the entire industry. A `Get` returns something. A `Set` changes something. A method starting with `Is`, `Has`, or `Can` answers a yes-or-no question and leaves everything as it found it. Nobody wrote these rules down, and no compiler enforces them, but every developer reads code with them in mind. Reach for one of these words and the promise comes along with it, whether we meant to make it or not.

That is what makes breaking them expensive. A `GetName` method that quietly loads data into a field will be called in a loop by someone who assumed it was cheap. An `IsEnabled` property that flips a flag while answering will produce a bug that takes hours to find, because the line that caused it looks like a question rather than an instruction. The reader had no reason to check. The name told them there was nothing to check.

The fix is not complicated. If a method does more than its name promises, either the name changes or the extra work moves elsewhere. A `Get` that has to load something can be renamed to `Load` or `Fetch`. A property that computes something expensive can become a method, since a method call signals work in a way a property does not.

These expectations are conventions rather than compiler-enforced rules, but following them consistently makes names more trustworthy to readers.

Avoiding ambiguity and inconsistency

A name can be clear on its own and still become confusing when the surrounding code uses ambiguous or inconsistent vocabulary. The following sections look at the most common forms of that problem.

Avoiding homonyms

Homonyms are words with many meanings. For example, "current" may refer to the present time or the flow of electricity or liquid. "Die" can mean "to stop living", or refer to an item we roll while playing games. Ambiguous words are not automatically bad; context can make them perfectly clear. In a board game, die naturally refers to the object being rolled. In other domains, however, words such as "current" can be ambiguous. It could mean the currently selected device or electrical current.

When more than one interpretation is plausible, add enough context to remove the ambiguity: `currentlyChosenDevice` and `electricCurrent` are longer than `current`, but they prevent different readers from assigning different meanings to the same name.

Using synonyms consistently

Synonyms are words with the same or nearly the same meaning, such as *quick* and *fast*, or *big* and *large*.

Consider a repository class that uses synonyms in method names:

```
public class PersonalDataRepository
{
    public DateTime GetDateOfBirth(int personId);
    public string FetchCityOfBirth(int personId);
    public string RetrieveSocialSecurityNumber(int personId);
}
```

This class looks chaotic, and the reason is that the same operation goes by three names: Get, Fetch, and Retrieve. All three mean retrieving data. Nothing distinguishes them, so a reader has to check each method to confirm they do the same kind of thing. This happens easily when several developers work on the same code over time.

The fix is to pick one word and use it everywhere. Get is the shortest and the most common, so it is the natural choice here.

```
public class PersonalDataRepository
{
    public DateTime GetDateOfBirth(int personId);
    public string GetCityOfBirth(int personId);
    public string GetSocialSecurityNumber(int personId);
}
```

Now, this code looks much better and it is easier to read. We see right away that those are related operations that read data for a person with the given ID.

Different words are appropriate when they represent truly different concepts. If Add combines two objects of the same type but Insert places a new element into an existing collection, the difference in vocabulary helps readers predict the different semantics.

Matching singular and plural forms

Names should reflect whether a value represents one item or a collection. A variable called `inactiveUserId` suggests one identifier; if it holds several values, the name should make that plurality visible unless the type itself represents one composite identifier.

```
// Misleading: the name implies one ID.  
var inactiveUserId = new[] { 4, 12, 16 };  
  
// Clear: the value is a collection.  
var inactiveUserIds = new[] { 4, 12, 16 };  
  
// Also clear if CompositeId is a single domain value.  
var inactiveUserId = new CompositeId(4, 12, 16);
```

Method names and parameter names should change when the cardinality changes. A refactoring from one vector to many vectors should update both identifiers:

```
Vector NormalizeVector(Vector vector) { ... }  
List<Vector> NormalizeVectors(List<Vector> vectors) { ... }
```

Plurality mistakes often appear after refactoring because a type changes while the original name is left behind. Updating the identifier at the same time keeps the code semantically correct.

Using natural language in code

Readable code often resembles ordinary language. Small words such as *than*, *by*, *to*, and *with* can make an expression much easier to parse because they turn an identifier into a phrase a reader would naturally say.

```
if (vector.IsShorter(otherVector)) { ... }  
if (vector.IsShorterThan(otherVector)) { ... }  
  
var newPoint = point.Transform(1, 2);  
var newPoint = point.MoveBy(1, 2);
```

The second version in each pair adds or changes only a small amount of text, but the intent is clearer because the call reads more like a sentence.

Choosing a language for identifiers

Many international software projects use English identifiers because programming ecosystems, APIs, documentation, and developer communities commonly use English. Some organizations or regulated projects may require another language. Whatever the project chooses, consistency matters more than mixing languages unpredictably within the same codebase.

If a project uses a language other than English for domain identifiers, document that choice and apply it consistently. Avoid switching between languages for equivalent concepts, because readers must then translate both the program logic and the vocabulary at the same time.

Avoiding meaningless words

Words such as `Manager`, `Handler`, `Data`, and `Info` are sometimes useful, but they often appear when the author has not identified a more precise responsibility. A class called `EmailManager` could send messages, manage accounts, validate addresses, or do all three. The word `Manager` tells the reader almost nothing.

```
class EmailManager { ... }  
class EmailHandler { ... }
```

The same goes for the three variables below:

```
var order = ...;  
var orderData = ...;  
var orderInfo = ...;
```

`Data` and `Info` add nothing to `order`, so a reader looking at all three has no way to tell what actually distinguishes them. If they hold different things, the names should say what those things are. If they hold the same thing, two of them should not exist.

The fix is to replace the generic word with whatever the domain actually calls the thing. If those two classes have distinct responsibilities, `EmailSender` and `AccountCreator` say what they are. If the three variables hold different parts of an order, `itemsInOrder`, `orderingCustomer`, and `orderStatus` tell the reader what each one contains before they read a single line further.

Avoiding overly specific names

Names can fail by being too vague, but they can also be too specific. A name should describe the abstraction that callers care about, not an implementation detail that may change later.

Consider an interface containing a method for reading people records. Naming its method `ReadFromSqlDatabase` is not the right choice. This name is too specific because it leaks an implementation detail into the abstraction. Callers do not care whether the data comes from a SQL database, a file, or a web service. They need to know that the method returns `Person` objects. If the storage moves elsewhere later, a name that mentions SQL becomes wrong, and every caller reads a lie until someone renames it.

The fix is to make the method name generic: `Read`. The classes implementing `IPeopleReader` can then be specific, since their names describe a concrete source rather than an abstraction.

```
public interface IPeopleReader
{
    List<Person> Read();
}

public class PeopleFromSqlDatabaseReader : IPeopleReader
{
    public List<Person> Read() { ... }
}

public class PeopleFromExcelFileReader : IPeopleReader
{
    public List<Person> Read() { ... }
}
```

Avoiding Hungarian notation and type-based names

Another way of packing implementation details into a name is Hungarian notation, which encodes the type in the identifier itself and produces names such as `intAge` or `strLastName`.

The convention made sense in environments with limited tooling, where knowing the type of a variable meant hunting for its declaration. Modern IDEs display that information on hover, so the prefix buys nothing and costs a few characters on every use.

It also carries the same risk as `ReadFromSqlDatabase`. A variable called `intCount` that has to start holding larger values becomes a `long`, and its name is now a lie. A variable called `numbersList` becomes misleading if its type changes to an array. Naming it `numbers` preserves the meaning across such implementation changes.

```
// Avoid type prefixes
int intAge;
string strLastName;
```

```
List<int> numbersList;  
  
// Prefer domain names  
int age;  
string lastName;  
List<int> numbers;
```

Avoiding confusingly similar names

Two names that differ by only one letter or a small grammatical change force readers to remember distinctions that the identifiers themselves do not explain. Classes called `UserDataStorage` and `UsersDataStorage`, for example, look as though they should mean the same thing. If they represent duplicate behavior, consolidate them; if they represent different behavior, rename them to expose the difference.

The same problem occurs with variables such as `ItemsCount` and `CountOfItems` in the same scope, or parameters distinguished only by numbers. Names such as `source` and `target` communicate a relationship that `parameter1` and `parameter2` cannot.

Using abbreviations carefully

Abbreviations save characters but often increase reading time. `addr` is less immediate than `address`, and `prod` could mean product or production. Source-code size is not a meaningful reason to shorten identifiers; compilers do not need developers to sacrifice readability to save a few characters in source text.

```
var addr = ...;      // Requires interpretation.  
var address = ...;   // Immediate meaning.  
  
var prod = ...;      // Product or production?  
var product = ...;   // Clear in this context.
```

The same caution applies inside lambda expressions. A single-letter parameter may be common in mathematical code, but a descriptive identifier can make the expression easier to read and easier to search across a codebase:

```
(a) => $"{a.Number} {a.Street} street."  
(address) => $"{address.Number} {address.Street} street."
```

There is one more argument against abbreviations, especially the single-letter ones: they make code hard to search. Searching for `"address"` returns the handful of places that word appears.

Searching for "a" returns every word that contains the letter, and finding the parameter among them takes forever.

However, some abbreviations are so widely used that expanding them would look odd rather than helpful. `ex` for an exception in a catch block, `i` for a loop index, and `id` for an identifier are read instantly by every developer, and nobody has to stop and work them out. The test is not the length of the name but whether a reader outside the project would recognize it without asking.

Identifier	Typical context	Why it can work
<code>ex</code>	Exception variable in a catch block	Widely recognized in a very narrow context
<code>i, j, k</code>	Short loop indices	Conventional and local; the scope is usually tiny
<code>a, b</code>	Simple mathematical operations	Matches familiar mathematical notation when the meaning is obvious

Table 2.3: When short conventional identifiers can be appropriate

Using context effectively

A name does not exist in isolation. The containing type, namespace, project, and call site already provide information, and repeating all of that context makes names longer without making them clearer.

Consider a variable called `state`. Its name gives no context at all. Is it the state of a process, or one of the United States? There is no way to tell without reading the surrounding code, and by then the name has already cost the reader time.

A name that shows the purpose right away avoids that. Renaming it to `addressState` makes clear that this is part of an address and will hold values like Texas or Ohio.

Renaming the variable is one way to make its meaning clear. Another is to give it a type that supplies the context. Instead of `addressNumber`, `addressCountry`, and `addressState` sitting next to each other, the three can be bundled into a struct called `Address`, and the prefixes can go. `Address.State` says everything `addressState` said, without saying "address" twice. The

type already carries that part, so a `State` field inside it can only mean California or Florida, not the state of a process.

```
struct Address
{
    public string Number;
    public string State;
    public string Country;
}
```

The same principle applies to methods. A `TextFileWriter` does not need a method called `WriteToFile` if callers already know the receiver is a text-file writer. `Write` is enough:

```
class TextFileWriter
{
    public void Write(string content) { ... }
}
```

Project context can remove repetition too. In a project dedicated entirely to an email service, prefixes such as `EmailSearchEngine`, `EmailNotifier`, `EmailSender`, and `EmailValidator` may make searching noisier without adding meaning. `SearchEngine`, `Notifier`, `Sender`, and `Validator` can be sufficient if no other domain objects create ambiguity.

Note

Context rule: Do not repeat information that the reader can reliably infer from the surrounding type or domain. Add context only when omitting it would create ambiguity.

Refactoring case study: Improving names

This case study brings the chapter's naming principles together in one example. The goal is to preserve the behavior while making the domain and intent easier to read.

Understanding the code

Let's start by working out what the code does, which is harder than it should be.

The class is called `Game`. It holds a 3 x 3 array of characters and has one public method called `Win`, which takes a character and returns a `bool`. Nothing in those names says what kind of

game this is, or what `Win` is supposed to mean. Does it check whether someone won? Does it make someone win?

Reading the body settles it. The method walks through the rows and looks for one where all three cells hold the same character as the one passed in. The comment tells us the column and diagonal checks work the same way. So this is tic-tac-toe, the array is the board, the character is a player's mark, and `Win` answers a question: has this player won?

All of that was in the code. None of it was in the names.

Starting point for the naming refactoring

```
public class Game
{
    private readonly char[,] gameArray = new char[3, 3];
    public Game(char[,] gameArray)
    {
        this.gameArray = gameArray;
    }

    public bool Win(char c)
    {
        // Check rows
        for (int i = 0; i < 3; i++)
        {
            if (gameArray[i, 0] == c && gameArray[i, 1] == c && gameArray[i,
2] == c)
            {
                return true;
            }
        }

        // Additional column and diagonal checks omitted here.
        return false;
    }
}
```

Identifying the issues

The class name is too general. `Game` would fit chess, poker, or hangman equally well. This class is specifically tic-tac-toe, and the name should say so.

The field name has three problems at once. `gameArray` does not follow the C# convention used in this codebase, which prefixes private fields with an underscore. It is also too general and too specific at the same time. "Game" would fit any game, while "Array" names the field's type rather than what the field holds, so the moment the type changes the name becomes wrong. On top of that, it is longer than it needs to be because the field represents the board and nothing else.

`Win` reads like a command. It sounds like a method that makes someone win. What it does is answer a question, and a method returning a `bool` should be named so the reader hears the question.

The parameter `c` says almost nothing. It most likely stands for `char`, which is its type, not its role. What it holds is a player's symbol.

The loop indices carry no meaning. A plain `i` is fine in a short loop, but here one loop walks the rows and another walks the columns, and the code does not say which is which.

The comments are doing the naming. The method separates the row, column, and diagonal checks with comments. Each of those blocks is a coherent operation, and a comment is a weaker way of labeling one than a method name would be.

Improving the code

With the naming problems identified, the refactoring can proceed in small steps. Each change below addresses one of the issues above while preserving the behavior of the code.

Make the domain visible

`Game` becomes `TicTacToe`, and the field becomes `_board`, with `board` as the constructor parameter:

```
public class TicTacToe
{
    private readonly char[,] _board;
    public TicTacToe(char[,] board)
    {
        _board = board;
    }

    ...
}
```

Make the Boolean method read like a question

Win becomes `IsWonBy`, and `c` becomes `playerSymbol`. The method now reads as the question it answers, and so does every call to it:

```
public bool IsWonBy(char playerSymbol)
{
    ...
}

Console.WriteLine("Is won by X? " + ticTacToe.IsWonBy('X'));
Console.WriteLine("Is won by O? " + ticTacToe.IsWonBy('O'));
```

Replace generic loop indices when domain names help

The conventional index `i` is acceptable in a short loop, but the domain provides more informative names here. The indices become `rowIndex` and `columnIndex`, so each loop says what it is walking through:

```
for (int rowIndex = 0; rowIndex < 3; rowIndex++)
{
    if (_board[rowIndex, 0] == playerSymbol && _board[rowIndex,
1] == playerSymbol && _board[rowIndex, 2] == playerSymbol)
    {
        return true;
    }
}
```

Replace explanatory comments with focused method names

The original method separates row, column, and diagonal checks with comments. Those comments mark coherent operations that can be extracted into their own methods—and extracting them creates an opportunity to name precisely what each one does. Once the methods are named `IsAnyRowFilledWith`, `IsAnyColumnFilledWith`, and `IsAnyDiagonalFilledWith`, the top-level method reads as a direct description of the winning condition.

```
public bool IsWonBy(char playerSymbol)
{
    return IsAnyRowFilledWith(playerSymbol)
        || IsAnyColumnFilledWith(playerSymbol)
```

```
        || IsAnyDiagonalFilledWith(playerSymbol);  
    }
```

The extracted helpers keep the same logic while making each piece of the algorithm explicit. The board size was also extracted into a constant, so its meaning doesn't have to be inferred from a bare number scattered throughout the code.

```
private const int BoardSize = 3;  
  
private bool IsAnyRowFilledWith(char playerSymbol)  
{  
    for (int rowIndex = 0; rowIndex < BoardSize; rowIndex++)  
    {  
        if (_board[rowIndex, 0] == playerSymbol && _board[rowIndex, 1] ==  
playerSymbol && _board[rowIndex, 2] == playerSymbol)  
        {  
            return true;  
        }  
    }  
  
    return false;  
}  
  
private bool IsAnyColumnFilledWith(char playerSymbol)  
{  
    for (int columnIndex = 0; columnIndex < BoardSize; columnIndex++)  
    {  
        if (_board[0, columnIndex] == playerSymbol && _board[1, columnIndex] ==  
playerSymbol && _board[2, columnIndex] == playerSymbol)  
        {  
            return true;  
        }  
    }  
  
    return false;  
}  
  
private bool IsAnyDiagonalFilledWith(char playerSymbol)  
{  
    return (_board[0, 0] == playerSymbol && _board[1, 1] == playerSymbol &&  
_board[2, 2] == playerSymbol)
```



```
    || (_board[0, 2] == playerSymbol && _board[1, 1] == playerSymbol &&  
    _board[2, 0] == playerSymbol);  
}
```

The row and column methods could be refactored further because their structure is similar, but that would move beyond the naming focus of this case study. The important result is that the code now exposes the domain directly: `TicTacToe` has a board, it can answer whether a player has won, and the winning condition is expressed in terms of rows, columns, and diagonals.

A practical naming review workflow

When reviewing names in your own code, use a repeatable sequence rather than searching randomly for "better words."

1. Read the signature before the implementation. Ask what you expect the construct to do based only on its name.
2. Check whether the name matches the construct: nouns for things, verbs for actions, and question-like names for Boolean answers.
3. Look for ambiguity, inconsistent synonyms, abbreviations, type information, and words that add no domain meaning.
4. Check singular and plural forms and verify that the name still matches the current type after refactoring.
5. Use the surrounding class, namespace, and project context to remove repeated information without creating ambiguity.
6. If the only accurate name is extremely long or contains several responsibilities, inspect the design before forcing a shorter name.
7. Rename safely with IDE tooling and run the relevant tests to confirm that the refactoring preserved behavior.

Summary

This chapter showed how names shape the readability of a codebase. You worked from basic naming patterns through language and project conventions, the Boy Scout rule, expressive and semantically correct names, consistency, context, abbreviations, and common naming problems. The tic-tac-toe case study demonstrated how naming changes can reveal and support deeper refactoring.

In *Chapter 3*, you will apply the same emphasis on clear intent to method signatures, parameter design, method size, abstraction, and side effects.

3

Writing better methods

Methods are where many clean-code problems first become visible. A method may begin as a few clear lines and gradually acquire more parameters, hidden dependencies, conditional branches, file operations, validation, and state changes. The method may still work, but every additional responsibility makes it harder to understand, test, and change safely.

The solution is not to chase an arbitrary method length or parameter count. Instead, design methods so that their signatures communicate what they need, their bodies stay focused on one task, and their dependencies remain visible. This chapter develops that approach from method fundamentals and parameter design through abstraction levels, ordering, independence, pure functions, and a larger refactoring case study.

In this chapter, we will cover the following topics:

- Method fundamentals, parameters, arguments, delegates, and lambda expressions
- Designing clear method signatures and parameter lists
- Reducing parameters without hiding dependencies
- Using Boolean parameters carefully and avoiding parameter anti-patterns
- Keeping methods small, focused, and at a consistent level of abstraction
- Ordering methods so that a class reads naturally
- Keeping methods independent and making prerequisites explicit
- Understanding pure functions and side effects
- Refactoring a large method into smaller, clearer units

By the end of this chapter, you will be able to design method signatures that communicate intent, recognize problematic parameter lists, split methods around clear responsibilities, keep operations at a consistent level of abstraction, reduce hidden dependencies and side effects, and refactor a large method into smaller units that are easier to understand and test.

Note**Download the PDF version of this book**

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Method fundamentals

Let's start with the basics. A function is a named block of code that performs a specific task. Functions make code reusable, and they are the basic building block of a program.

A function that belongs to a type is called a method. Some languages allow functions to be defined outside any type. C# does not, so every function in C# is a method. That's why this book uses both terms interchangeably.

A method can return a value or be void, meaning it doesn't return anything. It can also receive inputs. The names in the declaration are parameters; the concrete values supplied at the call site are arguments. In the following example, *a* and *b* are parameters, while 5 and 3 are arguments:

```
var sum = Add(5, 3);  
int Add(int a, int b) => a + b;
```

Naming methods

Method names should be built around a verb, because methods perform actions. Names like Add, Search, Load, and Modify clearly state what will happen. Methods returning a Boolean are the exception: their names read as yes-or-no questions, such as IsLoaded, CanBeDisabled, or HasBeenModified.

Delegates and lambda expressions

In many programming languages, methods can also be treated as values. In C#, delegates provide a type-safe way to store or pass callable behavior. The built-in Func delegates describe methods that return a value, while Action describes methods that return void. For example, the following variable can be assigned any method that accepts two integers and returns a Boolean:

```
Func<int, int, bool> someFunction = IsDivisibleBy;  
  
bool IsDivisibleBy(int number, int divisor) => number % divisor == 0;
```

Lambda expressions provide a concise way to define anonymous functions, which is particularly useful when the behavior is needed only at one call site. The following code uses a lambda predicate to keep only numbers greater than 10:

```
var numbers = new int[] { 1, 7, 19, 4, 20, 0 };  
var largeNumbers = numbers.Where(number => number > 10);
```

The parameter appears to the left of the => operator, and the expression that calculates the result appears to the right. Because this lambda answers whether a value satisfies a condition, it acts as a predicate.

Designing clear method signatures

A method signature is the contract visible to its callers. It includes the method name, return type, parameters, and modifiers. A good signature should make the method's requirements and result clear without requiring the caller to read the implementation.

We've already seen how important good method names are. But a method name is not read in isolation. At the call site, it appears together with the arguments, so the complete call should communicate its intent clearly.

Consider this call:

```
_fileWriter.Save("someString")
```

What does it do? It could save the string to a file, or it could write to a file named `someString`. The name and the argument together do not settle it.

Both cases have better names available. If the string is the content, `AppendText` says so when existing content is kept.

```
_fileWriter.AppendText("someString")
```

If the original content is not kept, `ReplaceContentWith` is a better name.

```
_fileWriter.ReplaceContentWith("someString")
```

If the string parameter represents a file name, `WriteTo` makes that clear.

```
_fileWriter.WriteTo("someString")
```

In each case the call reads as a sentence, and the ambiguity is gone.

A good signature carries information even without the name. Consider this one:

```
List<T> ???(List<T>, Func<T, bool>)
```

It takes a list and a predicate over the same type as items in this list, and returns a list of the same type. That is enough to make a reasonable guess: it likely filters the input using the predicate. The guess could be wrong, but the fact that a guess is possible at all is the point. Types carry meaning, and a good name adds to it rather than doing all the work alone.

Choosing the right number of parameters

How easy a method is to read and to call depends largely on how many parameters it takes. A method with none is the easiest case, since there is nothing for the caller to supply and nothing to get wrong. One parameter is still simple, though it already brings a value the caller can get wrong:

```
var firstNumber = numbers.First(); // impossible to call with an invalid argument
var itemAtIndexZero = numbers.ElementAt(0); // a valid argument is provided
here...
var invalidItem = numbers.ElementAt(-1); // ...but not here
```

As parameter lists grow, callers must remember more information and tests must account for more combinations. A connection method with many separate configuration values illustrates the problem:

```
public void ConnectToDatabase(  
    DbType type,  
    string databaseName,  
    string serverName,  
    Cluster serverCluster,  
    string userName,  
    string password,  
    int timeout)  
{  
    // ...  
}
```

A long list does not automatically mean the method is wrong, but it should prompt a design question: do these values expose several responsibilities or groups of data that should be modeled more clearly?

Splitting methods to reduce parameters

One reason a method accumulates parameters is that it performs more than one task. Consider a `SaveGame` method that receives game data, a file name, and a file extension. It both constructs the final file name and writes the game data:

```
var finalFileName = BuildFileName(  
    saveFileNameWithoutExtension,  
    saveFileExtension);  
SaveGame(gameData, finalFileName);
```

Separating the file-name construction from the saving gives each method a clearer purpose. `BuildFileName` can also be called from anywhere else that needs a file name built the same way, for example a method that reads the file back.

```
public string BuildFileName(  
    string fileNameWithoutExtension,  
    string extension)  
{  
    return fileNameWithoutExtension + "." + extension;  
}
```


The parameter reduction is a consequence of separating responsibilities rather than the goal by itself.

Grouping related parameters

Another option to reduce the number of parameters is to group values that naturally belong together. Let's consider a constructor for a `Circle` type:

```
public Circle(float x, float y, float radius) {...}
```

It takes three parameters. The first two are the coordinates of the circle's center, and the third is its radius.

The first two belong together, so they can be aggregated into a `Point` data structure. That leaves a constructor with two parameters instead of three, and the grouping matches how a circle is actually described: a center and a radius. The `Point` type is also reusable, and anywhere else in the application that currently passes a loose pair of coordinates can use it.

```
public struct Circle
{
    public Point Center { get; }
    public float Radius { get; }

    public Circle(Point center, float radius)
    {
        Center = center;
        Radius = radius;
    }
}

public record Point(float X, float Y);
```

Let's consider another example. Here is a method that establishes a database connection:

```
public IDatabaseConnection ConnectToDatabase(
    string userName,
    string password,
    string databaseName,
    string databaseServerName,
    string databaseClusterName)
```

It takes five parameters, which is enough to be a problem on its own. The first step is to look for parameters that belong together. Three of them do, and their names are already similar, which

is a hint that they describe one thing rather than three. They can be grouped into a `DatabaseIdentity` type.

```
public struct DatabaseIdentity
{
    public string DatabaseName;
    public string DatabaseServerName;
    public string DatabaseClusterName;
}
```

This is better, but there is one more grouping available. `userName` and `password` are both user credentials. Wrapping them in a dedicated type leaves the method with two parameters, each describing one thing.

```
public struct UserCredentials
{
    public string UserName;
    public string Password;
}
```

Now we can use those new types to make the signature of the `ConnectToDatabase` method much cleaner:

```
IDatabaseConnection ConnectToDatabase(
    UserCredentials userCredentials,
    DatabaseIdentity databaseIdentity);
```

This is useful only when the grouped values form a meaningful concept. A wrapper created only to reduce the parameter count hides the problem instead of fixing it.

Using Boolean parameters carefully

As noted earlier, methods often take too many parameters because they do too many things. Boolean parameters are a particular case: the method does one thing when the value is `true` and something else when it is `false`. Consider the following code:

```
var distance1 = CalculateDistance(point1, point2, true);
var distance2 = CalculateDistance(point1, point2, false);
```

What is the difference between those two results? The call sites do not say, and finding out means opening `CalculateDistance`.

```
double CalculateDistance(Point point1, Point point2, bool shouldBeAsMiles)
{
    double deltaX = point2.X - point1.X;
    double deltaY = point2.Y - point1.Y;
    var distanceInKm = Math.Sqrt(deltaX * deltaX + deltaY * deltaY);

    if (shouldBeAsMiles)
    {
        const double kilometersToMilesConversionFactor = 0.621371;
        return distanceInKm * kilometersToMilesConversionFactor;
    }

    return distanceInKm;
}
```

Inside, the Boolean turns out to decide whether the result is in kilometers or miles. That is two problems at once. The reader had to leave the call site to understand it, and the method does two things: it calculates a distance and it converts the unit.

The fix is to let the method do the first thing only. Renaming it to `CalculateDistanceInKilometers` puts the unit in the name, where the caller can see it.

```
var distanceInKm = CalculateDistanceInKilometers(point1, point2);
```

Callers who need miles get a separate method whose only job is the conversion:

```
var distanceInKm= UnitConverter.KilometersToMiles(distanceInKm);
```

Each method now does one thing, and the Boolean parameter is gone. The call site says what unit it produces, so nothing has to be looked up.

Not every Boolean parameter is a problem. A value such as `isAdmin` can be legitimate when it is simply data used by a single decision rather than a flag that switches the method between unrelated jobs:

```
bool IsAuthorized(string userName, bool isAdmin)
{
    if (isAdmin)
```

```
{  
    return true;  
}  
  
return _authorizedUsers.Contains(userName);  
}
```

Note

Review question: When you see a Boolean parameter, ask whether true and false cause the method to perform different responsibilities. If they do, separate those responsibilities or make the choice explicit in the API.

Avoiding parameter-count anti-patterns

Do not cut parameters at any cost. Some operations need several independent values, and no grouping will change that. The `DateTime` constructor is a good example. It represents a point in time, so it takes a year, a month, a day, an hour, a minute, and a second. Bundling those into another object to bring the count down would add a layer of indirection without making the concept any clearer.

```
var titanicSinkingDate = new DateTime(1912, 4, 15, 2, 20, 0);
```

Likewise, a method that compares two values within a tolerance naturally needs the two values and the tolerance:

```
bool AreEqualWithinTolerance(float a, float b, float tolerance)  
{  
    return Math.Abs(a - b) <= tolerance;  
}
```

A damaging anti-pattern is hiding a former parameter in instance state. The following pure helper exposes all three inputs in its signature, making it explicit and easy to understand:

```
public static decimal CalculateTotalPrice(  
    int quantity, decimal unitPrice, decimal tax)  
{  
    return quantity * unitPrice * (1 + tax);  
}
```

Someone might see three parameters and decide that moving tax to a field would be an improvement. It would not. The method still depends on three values, and one of them has merely been hidden from the signature. Reading the call no longer tells us what the calculation is based on. The method now depends on instance state, so it cannot be called with a different tax rate without constructing an object first, and testing it in isolation becomes harder than it was.

```
public class OrderPriceCalculator
{
    private readonly decimal _tax;

    public OrderPriceCalculator(decimal tax)
    {
        _tax = tax;
    }

    public decimal CalculateTotalPrice(int quantity, decimal unitPrice)
    {
        return quantity * unitPrice * (1 + _tax);
    }
}
```

Fields exist to hold state that belongs to the object: data it needs to remember between calls because that data is part of what the object is. A list keeps its elements. A database connector keeps the connection details it was given in its constructor and reuses them across Connect and Disconnect.

A parameter is different. It is an input to one particular operation, and the object has no reason to remember it afterward. Moving such an input into a field does not remove it from the design; it only removes it from the signature. The method now depends on a value set somewhere else.

Before turning a parameter into a field, *ask whether the object would still make sense holding that value once the method has returned*. If not, it belongs in the signature.

Ordering parameters logically

The order of parameters should follow a pattern that callers can predict. DateTime is naturally read from larger units to smaller ones: year, month, day, hour, minute, and second. A database

location can similarly be described from the broader server cluster to the server and then the database.

```
public void Connect(  
    string serverClusterName,  
    string serverName,  
    string databaseName)  
{  
    // ...  
}
```

Another useful rule is to follow the order suggested by the method name. If a method is called `IsPersonPresentInGroup`, a person parameter followed by a group parameter matches the sentence the caller reads.

Keeping methods small and focused

Long methods are difficult to understand because the reader must retain more state, branches, and intermediate values at once. Very large methods are also harder to test and more likely to combine unrelated responsibilities. Small methods reduce that cognitive load and make individual operations easier to name and reuse.

A practical heuristic is the size of the screen: if a method runs past roughly 20 lines, or past 120 characters in width, it is worth a second look. *Treat these numbers as warning thresholds rather than hard limits.* A cohesive 25-line method can be clearer than five artificial helpers, while a six-line method can still mix unrelated responsibilities.

Long lines can usually be broken without changing the design. Long methods require a different question: does every part of this method contribute to the same task at the same conceptual level?

Giving each method one task

A focused method does one coherent job. Some jobs are small computations, such as squaring a number or checking whether a number is even:

```
float Square(float number)  
{  
    return number * number;  
}  
  
bool IsEven(int number)  
{
```

```
    return number % 2 == 0;
}
```

A single task can also be orchestration. Reading user data from an API may require connecting, retrieving JSON, and deserializing it. A high-level method can coordinate those steps without implementing each detail itself:

```
List<User> ReadUsersData()
{
    var apiConnection = ConnectToApi("https://someApi/", "api/users");
    var usersAsJson = apiConnection.ReadAll();
    return JsonSerializer.Deserialize<List<User>>(usersAsJson);
}
```

The method still has one responsibility: coordinating the process of reading user data. Its sub-steps are delegated to operations at a lower level of abstraction.

Recognizing methods that do too much

A short method can still contain more than one responsibility. In the following example, `SaveToFile` writes content and silently adds a `.txt` extension:

```
void SaveToFile(string fileName, string content)
{
    File.WriteAllText(fileName + ".txt", content);
}
```

String concatenation is a low-level file-name construction detail placed directly beside the higher-level file-writing operation. The path-building logic is also stuck inside this method, so anything else that needs to build the same path has to repeat it.

Consider a method that filters empty strings, joins the remaining strings, and writes the result to a file:

```
void SaveToFile(List<string> words, string filePath)
{
    var nonEmptyWords = new List<string>();
    foreach (var word in words)
    {
        if (!string.IsNullOrEmpty(word))
        {
            nonEmptyWords.Add(word);
        }
    }
}
```

```

    }
}

string text = string.Join(Environment.NewLine, nonEmptyWords);
File.WriteAllText(filePath, text);
}

```

Renaming the method to `SaveNonEmptyToFile` makes the filtering visible but does not remove the mixed responsibilities. Extracting the filter creates a reusable operation and leaves the main method to coordinate the steps:

```

List<string> GetNonEmptyOnly(List<string> words)
{
    var nonEmptyWords = new List<string>();
    foreach (var word in words)
    {
        if (!string.IsNullOrEmpty(word))
        {
            nonEmptyWords.Add(word);
        }
    }
    return nonEmptyWords;
}

void SaveNonEmptyToFile(List<string> words, string filePath)
{
    var nonEmptyWords = GetNonEmptyOnly(words);
    var textToBeSaved = string.Join(Environment.NewLine, nonEmptyWords);
    File.WriteAllText(filePath, textToBeSaved);
}

```

The example highlights several signals that a method may be doing too much. These signals are summarized in *Table 3.1*.

Signal	What it may indicate
Mixed abstraction levels	The method coordinates a high-level task while also performing low-level implementation details.
Large size	Several responsibilities or branches may have accumulated.

Signal	What it may indicate
Difficult naming	The method may not have one coherent purpose.
AND, OR, or IF in the only accurate name	The method may perform multiple actions or embed a condition that belongs elsewhere.
Commented blocks	Each block may represent a separately nameable operation.

Table 3.1: Signals that a method may need to be split

Maintaining consistent levels of abstraction

Abstraction is a simplified view of something. It shows the idea we need and hides the machinery underneath.

We rely on it constantly without thinking about it. Take a method that splits a piece of text into words. To use it, we need to know one thing: text goes in, a collection of words comes out. We do not need to know how the text is stored in memory, how the method walks through it, or where the resulting words are allocated. All of that exists, and all of it is hidden behind the signature.

Abstractions come at different levels, and the difference between them is how much detail they hide.

Consider a `BuildHouse` method. It sits at a high level, because one call stands for an enormous amount of work. `BuildWalls` is one step below it, since building walls is part of building a house. Below that come `BuildFoundation`, `PlaceBricks`, `BindBricksWithMortar`, and `CoverWithPlaster`. Each step down brings more detail and a narrower view of the whole.

```
public void BuildHouse()
{
    BuildWalls();
    PlaceDoors();
    PlaceWindows();
    InstallElectricity();
}

public void BuildWalls()
{
    BuildFoundation();
    PlaceBricks();
}
```

```
        BindBricksWithMortar();  
        CoverWithPlaster();  
    }
```

A readable method calls operations that sit roughly one level below its own. `BuildHouse` calls `BuildWalls`, and `BuildWalls` calls the steps that make up a wall.

Flattening those lower steps into `BuildHouse` makes the method harder to read, because the reader keeps switching between scales:

```
public void BuildHouse()  
{  
    BuildFoundation();  
    PlaceBricks();  
    BindBricksWithMortar();  
    CoverWithPlaster();  
    PlaceDoors();  
    PlaceWindows();  
    InstallElectricity();  
}
```

The same principle applies to all operations our code executes. Let's consider the `SaveNonEmptyToFile` method again.

```
void SaveNonEmptyToFile(List<string> words, string filePath)  
{  
    var nonEmptyWords = GetNonEmptyOnly(words);  
    var textToBeSaved = string.Join(Environment.NewLine, nonEmptyWords);  
    File.WriteAllText(filePath, textToBeSaved);  
}
```

It reads as an instruction: filter the words, join them, write them to the file. It calls three methods, each one level less abstract than itself, and reading it does not require knowing how any of them work. It is enough that `GetNonEmptyOnly` returns the non-empty strings, which we can learn from its name alone. How it does that stays out of the way, which is exactly what makes the method readable. In the original version, the filtering loop sat inside the method and distracted from what the method was really doing.

Now imagine this method without `string.Join` and `File.WriteAllText`, with low-level code in their place: joining the strings character by character, opening a file handle, writing to a

stream, closing it. Following that would be much harder. And when low-level instructions are wrapped in higher-level methods, they can be reused elsewhere.

This is why the operations inside a method should sit at a similar level of abstraction. A method that mixes them forces the reader to switch between levels while reading, and that switching is what makes code tiring to follow. Moving the lower-level operations into methods of their own solves more than readability. Once the low-level logic lives in its own method, anything else that needs the same mechanism can call it instead of repeating it.

Note

Practical rule: *Within a method, prefer operations that sit at a consistent conceptual level.*

Delegate lower-level details when exposing them would distract from the method's primary task.

Ordering methods for readability

Method order is mainly a reading concern. Language requirements come first, but where the language gives you freedom, arrange methods so that readers encounter the public behavior and high-level flow before implementation details.

- Place public constructors before ordinary methods; keep constructors grouped together.
- Place public methods before private implementation helpers.
- When one method calls another, place the higher-level caller above the lower-level helper where practical.
- Keep overloads together; ordering them from fewer parameters to more parameters can make the group easier to scan.

Treat these as readability guidelines rather than rigid rules.

For example, MethodA appears before the helpers it delegates to:

```
void MethodA()  
{  
    MethodB();  
    MethodC();  
}  
  
void MethodB()  
{  
}
```

```
void MethodC()
{
}
```

This top-down arrangement allows a reader to understand the main flow and inspect lower-level details only when necessary.

Keeping methods independent

A method should not depend on an undocumented ritual before it can be called. Suppose a PDF library appears to support the following straightforward usage:

```
var pdfReader = new PdfReader();
string content = pdfReader.ReadFrom("someFile.pdf");
```

If the call fails because the user was expected to invoke `Init` first, the method contract hides an important prerequisite:

```
var pdfReader = new PdfReader();
pdfReader.Init();
string content = pdfReader.ReadFrom("someFile.pdf");
```

Required setup should be made explicit. Stable configuration can be required in the constructor, while operation-specific data can be required as method parameters. *The caller should not have to remember a magic sequence of unrelated calls or global assignments.*

Using pure functions where they fit

One of the best ways of improving your functions is to make them pure. A pure function satisfies two conditions: its result depends only on its input values, and it has no side effects. Given the same arguments, it therefore returns the same result and does not alter state elsewhere in the program.

```
int Max(int a, int b)
{
    return a >= b ? a : b;
}
```

`Max` is pure because its result is completely determined by `a` and `b`.

Now, let's consider another method:

```
class ItemsStorage
{
    private List<Item> _items = new();

    public Item FindItemWithName(string name)
    {
        return _items.FirstOrDefault(item => item.Name == name);
    }
}
```

The `FindItemWithName` method reads a field, which is what makes it impure. The field can change independently of the arguments, so the same argument may produce a different result on the next call.

Likewise, a method that reads `DateTime.Now` depends on an external value. It will produce a different result every time it is called, which means it is not pure:

```
string GetCurrentTime(string format)
{
    return DateTime.Now.ToString(format);
}
```

Side effects are the other source of impurity. A side effect is any change a function makes outside its return value, such as writing to a field, modifying a collection it was given, or touching a file or a database. An example could be mutating a list supplied by the caller.

```
public void RemoveFirstItem(List<Item> items)
{
    items.RemoveAt(0);
}
```

Characteristic	Pure function	Impure function
Inputs	All required values are explicit arguments.	May also depend on fields, global state, time, repositories, or other external data.

Characteristic	Pure function	Impure function
Repeatability	Same arguments produce the same result.	Same arguments can produce different results when external state changes.
Side effects	Does not modify external state.	May update fields, parameters, files, databases, UI state, or other objects.
Testing	Requires only input values and expected outputs.	May require setup, fixtures, mocks, cleanup, or environmental control.

Table 3.2: Pure and impure functions compared

Why pure functions are easier to work with

Pure functions are easy to read, because understanding one requires nothing but its own body. There is no additional context to keep in mind, no state of the surrounding class, no global variables. Everything the function needs is listed in its parameters.

The benefit is clearer when set against impure functions. An impure function may reach for anything: fields of its own class, static members of other classes, global state. A method may appear to take two parameters while in fact it also reads two fields from the current class, three from another, and a handful of globals. Its dependencies are not visible in its signature, so understanding it means finding all of them first.

Impure functions may also modify state, whether their own parameters, the fields of their class, or objects belonging elsewhere. This makes changes hard to trace. A list passed to an innocent-looking method can come back empty, and nothing at the call site suggested that it might. When any call can change state behind the scenes, no assumption about the state of the program is safe.

Pure functions remove both problems. They depend only on their parameters and they change nothing, so a caller knows that nothing else in the program moved while the function ran.

They are also unusually easy to test. A test needs no setup, since there is no context to establish, and the same arguments always produce the same result. Calling the function with a few sets of arguments and checking the results is enough.

Not every function can be pure. Programs have to talk to users, databases, and other systems, and all of that involves side effects. *The practical goal is a codebase split in two: impure functions at the boundaries, where the program meets the outside world, and a core built from pure ones.*

Refactoring case study: Improving method design

This case study brings the chapter's method-design principles together in one example, moving from diagnosis to a sequence of focused refactorings.

Understanding the code

Start with the following method:

```
using System.Text.Json;

public class FileManager
{
    public void IdFinder(string name, string dir, string ex, int id)
    {
        var file = dir + "/" + name + "." + ex;
        if (!File.Exists(file))
        {
            Console.WriteLine("File does not exist under path: " + file);
            return;
        }

        List<int> numbers = new List<int>();
        if (ex == "txt")
        {
            var txt = File.ReadAllText(file);
            var ids = txt.Split(',');
            foreach (var fileId in ids)
            {
                numbers.Add(int.Parse(fileId));
            }
        }
        else if (ex == "json")
        {
            var txt = File.ReadAllText(file);
            numbers = JsonSerializer.Deserialize<List<int>>(txt);
        }
        else
        {
            Console.WriteLine("Unsupported file extension: " + ex);
            return;
        }
    }
}
```

```
    }

    foreach (var number in numbers)
    {
        if (number == id)
        {
            Console.WriteLine($"Id {id} has been found in the file {file}.");
            return;
        }
    }

    Console.WriteLine($"Id {id} has not been found in the file {file}.");
}
}
```

Neither the names nor the structure tell us much, so the only way to find out what this method does is to read the body, and ideally run it. The code is in the repository accompanying this book, so feel free to work through it yourself.

The method checks whether a given ID is present in a file. The path arrives in three parts: the file name, the directory, and the extension, which is either txt or json. Those three are joined into a path, and if no file exists there, the method prints a message and returns.

The two supported formats hold the same data in different shapes. A txt file holds numeric IDs separated by commas:

```
1,2,3,4
```

A json file stores a JSON array:

```
[1,2,3,4]
```

Reading them differs, but the goal is the same: build a list of IDs from the file. For txt, the method reads the text, splits it on commas, and parses each part into a number. For json, it hands the text to the deserializer.

With the list in hand, the method iterates over it looking for the ID it was given, and prints a message saying whether it was found.

Identifying the issues

Understanding this method required reading every line of it. Good code does not demand that. Here is what stands in the reader's way.

The method is too long and does too many things. It builds a path, checks that a file exists, decides how to parse it, parses it, searches the result, and prints a message. Each of those is a separate job.

It mixes levels of abstraction. Concatenating strings to build a path sits right next to printing a result to the console, and the reader has to move between those levels while following what the method does.

The names do not describe what things are. `IdFinder` is a noun, so it names a thing rather than an action, and `FileManager` could be anything at all. The local variable `file` holds a path, not a file.

The parameters are unclear and badly ordered. The parameters `name` and `dir` seem to build a file path between them, while `ex` usually means an exception rather than an extension. On top of that, a path reads as directory, then name, then extension, while the parameters arrive in a different order.

Three of the four parameters describe one thing. The directory, the file name, and the extension are not independent values. Together they identify a file.

The supported extensions are hard-coded as string literals. `"txt"` and `"json"` are embedded directly in the parsing branches, so the set of supported formats is not defined in one place.

Improving the code

With the problems identified, the refactoring can proceed in a sequence of focused changes.

Clarifying the signature

The class name `FileManager` is too general, and `IdFinder` is a noun-like name that does not describe the operation precisely. Renaming them to `IdExistenceChecker` and `CheckIfIdExistsInFile` reveals the intent.

```
public class IdExistenceChecker
{
    public void CheckIfIdExistsInFile(string name, string dir, string ex, int id)
```

Now the parameters. Their order is the first thing to fix. A file path reads as directory, then name, then extension, and the parameters arrive in a different order. Matching the order of the path makes the signature easier to read and harder to get wrong at the call site.

```
public void CheckIfIdExistsInFile(string dir, string name, string ex, int id)
```

Directory, file name, and extension jointly describe a file identity. Bundling them reduces the number of independent string parameters and gives the data a reusable meaning:

```
public struct FileIdentity
{
    public string Directory { get; }
    public string Name { get; }
    public string Extension { get; }
    public FileIdentity(string directory, string name, string extension)
    {
        Directory = directory;
        Name = name;
        Extension = extension;
    }
}
```

We can now use this new type in the signature of the `CheckIfIdExistsInFile` method:

```
public void CheckIfIdExistsInFile(FileIdentity fileIdentity, int id)
```

This is better already, though there is one more thing to fix in the signature. The name mentions the ID first and the file second, while the parameters arrive in the opposite order. Swapping them so the parameters follow the name makes the whole signature read naturally.

```
public void CheckIfIdExistsInFile(int id, FileIdentity fileIdentity)
```

There is one more improvement we can add to the `FileIdentity` type. Consider the first line of the method. It builds a file path from parts:

```
var file= dir + "/" + name + "." + ex;
```

But this is a job of its own and belongs in a method. Notice, though, how much of `FileIdentity` that line uses. Every value it needs comes from that type, and it needs nothing else. That is a strong signal the method belongs on `FileIdentity` rather than here.

```
public struct FileIdentity
{
    public string Directory { get; }
    public string Name { get; }
    public string Extension { get; }

    public FileIdentity(string directory, string name, string extension)
    {
        Directory = directory;
        Name = name;
        Extension = extension;
    }

    public string AsPath() => Directory + "/" + Name + "." + Extension;
}
```

The signature of the method and the supporting `FileIdentity` type are now complete. Let's start dealing with the method body.

Validating early and extracting constants

Let's now improve the method implementation. This method is too long and does too many things. It also mixes abstraction levels: a low-level operation such as concatenating strings sits next to the higher-level task of reporting the result to the console, forcing the reader to move between levels while following the method.

The original method discovers unsupported extensions only after it has already performed other work. Validating prerequisites early simplifies the remaining flow.

Hard-coded extension literals can also be moved to constants so that the supported formats are defined in one place.

```
private const string Txt = "txt";
private const string Json = "json";

public void CheckIfIdExistsInFile(int id, FileIdentity fileIdentity)
{
    var filePath = fileIdentity.AsPath();
```

```
    if (!File.Exists(filePath))
    {
        Console.WriteLine("File does not exist under path: " + filePath);
        return;
    }

    if (fileIdentity.Extension != Txt && fileIdentity.Extension != Json)
    {
        Console.WriteLine(
            "Unsupported file extension: " + fileIdentity.Extension);
        return;
    }
    //...
}
```

A local variable that stores the path should be named `filePath` rather than `file`, because that is the value it actually contains. Precise names remove another source of mental translation.

Extracting file-reading responsibilities

The text and JSON branches are independently nameable operations at a lower level of abstraction than the main method. Extracting them allows the main workflow to stop dealing with parsing details:

```
private static List<int> ReadIdsFromFile(FileIdentity fileIdentity)
{
    var fileContent = File.ReadAllText(fileIdentity.AsPath());

    return fileIdentity.Extension == Txt
        ? ReadIdsFromText(fileContent)
        : ReadIdsFromJson(fileContent);
}

private static List<int> ReadIdsFromText(string fileContent)
{
    return fileContent
        .Split(',')
        .Select(fileId => int.Parse(fileId))
        .ToList();
}

private static List<int> ReadIdsFromJson(string fileContent)
```

```
{  
    return JsonSerializer.Deserialize<List<int>>>(fileContent);  
}
```

Simplifying the search and printing the result

Once we have IDs read from the files, we can check whether the given ID is present in the collection. The original foreach loop searches the list manually. The Contains method gives us a simpler and more elegant way to do it:

```
var idsFromFile = ReadIdsFromFile(fileIdentity);  
bool isIdPresentInFile = idsFromFile.Contains(id);
```

Printing the result can also be extracted so that the high-level method reads as a sequence of decisions and actions rather than string-formatting details:

```
private void PrintResult(bool isIdPresentInFile, int id, string filePath)  
{  
    var status = isIdPresentInFile ? "" : "not ";  
    Console.WriteLine($"Id {id} has {status}been found in the file {filePath}.");  
}
```

Let's see the final version of the method:

```
public void CheckIfIdExistsInFile(int id, FileIdentity fileIdentity)  
{  
    var filePath = fileIdentity.AsPath();  
    if (!File.Exists(filePath))  
    {  
        Console.WriteLine("File does not exist under path: " + filePath);  
        return;  
    }  
  
    if (fileIdentity.Extension != Txt && fileIdentity.Extension != Json)  
    {  
        Console.WriteLine(  
            "Unsupported file extension: " + fileIdentity.Extension);  
        return;  
    }  
  
    var idsFromFile = ReadIdsFromFile(fileIdentity);
```

```
bool isIdPresentInFile = idsFromFile.Contains(id);  
PrintResult(isIdPresentInFile, id, filePath);  
}
```

It can now be easily read and understood: validate the file, read the IDs, determine whether the requested ID is present, and report the result. The extracted methods are short and easy to name because each one has a clear responsibility. *The refactoring does not change the intended behavior; it changes how clearly the code communicates that behavior.* Moreover, the `FileIdentity` type, along with its path-building method, can now be easily reused in other parts of the application.

Note

Case-study lesson: *Good method design is cumulative.* Clear names, grouped parameters, early validation, smaller methods, direct collection operations, and consistent abstraction levels reinforce one another rather than working as isolated rules.

Summary

This chapter developed a practical approach to method design. You examined signatures and parameters, grouped related data, treated Boolean flags cautiously, kept methods focused and at consistent abstraction levels, ordered methods for readability, made prerequisites explicit, and used pure functions where they simplify reasoning. The final refactoring case study combined these ideas in a larger workflow.

In *Chapter 4*, you will turn from method structure to the visual organization of source code and see how consistent formatting reduces the effort required to read it.

4

Formatting code

Code can be logically correct and still be difficult to work with. When indentation is inconsistent, long expressions run off the screen, braces appear unpredictably, or unrelated statements are packed together, readers must reconstruct structure before they can think about behavior.

The solution is to make formatting predictable. A shared style, supported by automated tools, lets the visual structure of the code reinforce its logical structure. This chapter focuses on practical formatting decisions, and on a **refactoring** exercise that takes a poorly formatted program and makes it easier to scan and maintain.

In this chapter, we will cover the following topics:

- Why formatting matters to readability and maintainability
- Choosing and documenting a consistent formatting style
- Using **IDEs** and automated tools to enforce conventions
- Applying indentation consistently
- Structuring code blocks and using braces safely
- Keeping line length manageable and breaking long expressions clearly
- Using empty lines for vertical organization
- Refactoring poorly formatted code

By the end of this chapter, you will be able to define and apply a consistent formatting approach, use automated tooling to reduce formatting drift, organize code so that its logical structure is visible at a glance, and refactor poorly formatted code.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Why formatting matters

Code formatting is the consistent arrangement of source code: indentation, spacing, line breaks, braces, and the visual separation of related statements. *Formatting changes how code looks rather than how it behaves, yet it has a direct effect on how quickly a developer can understand and safely modify that code.*

The same principle is obvious in ordinary text. A page with inconsistent font sizes, random line breaks, and uneven spacing is harder to read even when every word is correct. Source code behaves similarly: the reader relies on visual patterns to recognize nesting, related statements, and boundaries between logical units. *Figure 4.1* compares two presentations of the same passage to show how consistency affects readability.

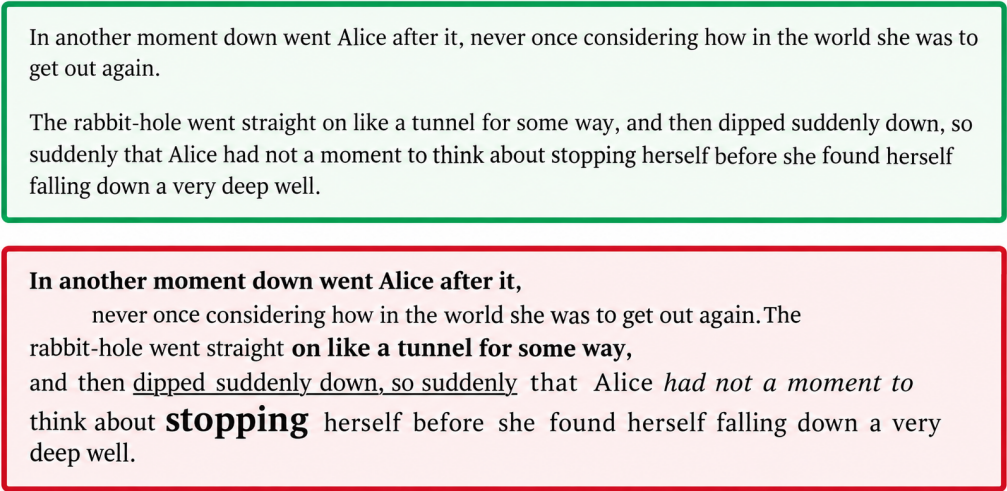


Figure 4.1: The same passage shown with consistent formatting at the top and mixed, inconsistent formatting at the bottom.

The following method illustrates the same effect in code. Both versions return false when any number is odd and true when every number is even, but the first compresses the loop and condition into a single line:

Before formatting	After formatting
<pre>bool AreAllEven(List<int> numbers) { for (int i = 0; i < numbers.Count; i++) if (numbers[i] % 2 != 0) return false; return true; }</pre>	<pre>bool AreAllEven(List<int> numbers) { for (int i = 0; i < numbers.Count; i++) { if (numbers[i] % 2 != 0) { return false; } } return true; }</pre>

The formatted version requires more vertical space, but the loop, condition, and return paths can be recognized immediately. *This is the central purpose of formatting: reduce the effort required to recover the structure that is already present in the program.*

Choosing and following a formatting style

There is no single formatting style that is correct for every language, project, or team. Some decisions are dictated by the language itself, while others come from community conventions or local team standards. Understanding the source of a rule helps you decide how strictly it must be followed.

Source	What it means	Example
Language syntax	The language parser requires the structure; changing it can alter or invalidate the program.	Python uses indentation depth as part of its syntax.
Language or community convention	The compiler may allow alternatives, but the ecosystem expects a familiar form.	C# projects normally follow established casing and brace conventions.
Team or project standard	The team selects one consistent approach where several acceptable styles exist.	A team agrees on brace placement, tab width, or line-breaking rules.

Table 4.1: Sources of formatting rules

For example, C# allows substantial freedom in indentation because braces define blocks. Python is different: indentation is part of the language syntax and therefore cannot be treated as a purely visual preference. The distinction becomes clear when the same logic is written in both languages.

In C#, braces determine the boundaries of the `for` loop and `if` statement:

```
bool AreAllEven(List<int> numbers)
{
    for (int i = 0; i < numbers.Count; i++)
    {
        if (numbers[i] % 2 != 0)
        {
            return false;
        }
    }
}
```

```
    }  
  }  
  return true;  
}
```

Python expresses the same structure through indentation rather than braces:

```
def are_all_even(numbers):  
    for number in numbers:  
        if number % 2 != 0:  
            return False  
    return True
```

In the C# example, changing the indentation does not necessarily change which statements belong to a block because the braces define that structure. In Python, the indentation itself determines which statements belong to the loop and condition, so incorrect indentation can change the program structure or make the code invalid.

Beyond syntax, *consistency matters more than individual preference*. A codebase is easier to read when the same choice is applied everywhere. Once a reader learns the project pattern, the visual structure becomes predictable and attention can remain on behavior rather than presentation.

If your team defines conventions beyond the mainstream language style, document them. A written standard prevents repeated debates and gives new team members a clear reference. The goal is not to create an exhaustive rulebook for every possible line of code, but to record the decisions that materially affect consistency.

Using automated formatting tools

Formatting is an ideal candidate for automation because the rules are repetitive and objective once a style has been chosen. Most modern IDEs can reformat an entire document or a selected block of code, and many can also flag violations as you type.

In Visual Studio, for example, the **Format Document** command can normalize spacing, indentation, and brace alignment across a file. The following example shows how an automated formatter can transform inconsistent code into a version that follows the project style:

Before formatting	After formatting
<pre>bool IsWithinRange(int min, int max, int value, bool isInclusive) { if (isInclusive) { return value >= min && value <= max; } else { return value > min && value < max; } }</pre>	<pre>bool IsWithinRange(int min, int max, int value, bool isInclusive) { if (isInclusive) { return value >= min && value <= max; } else { return value > min && value < max; } }</pre>

In the formatted version, the logic remains the same, but the structure is clearer and more consistent with the configured project style.

The default formatter is only a starting point. IDE settings can be changed to match project conventions, and configuration files can be committed to the repository so that all developers use the same rules.

Tool or mechanism	Typical role
IDE formatter	Reformats a document or selection according to configured rules.
IDE inspections and quick fixes	Highlight style inconsistencies and offer automated corrections.
Linters and analyzers, such as ESLint or SonarQube for IDE	Flag style and quality problems as the code is written.

Tool or mechanism	Typical role
CI/CD quality checks, such as SonarQube Server	Apply the agreed rules automatically every time code is pushed to the repository, so formatting does not depend on how each developer has configured their own editor.

Table 4.2: Tools that can help enforce formatting and style

Automated tools should remove routine formatting work from code review. Reviewers can then focus on design, correctness, naming, and maintainability instead of repeatedly pointing out spacing or brace placement.

Applying indentation consistently

Indentation shows what belongs to what. A statement indented under a loop, a condition, a method, or a class is read as part of it. When the indentation is consistent, the structure of the code can be taken in before a single expression is read. Compare an unindented block with the same logic formatted conventionally. The compiler may accept both, but the second version communicates the structure much more clearly:

Before formatting	After formatting
<pre>bool AreAllEven(List<int> numbers) { for (int i = 0; i < numbers.Count; i++) { if (numbers[i] % 2 != 0) { return false; } } return true; }</pre>	<pre>bool AreAllEven(List<int> numbers) { for (int i = 0; i < numbers.Count; i++) { if (numbers[i] % 2 != 0) { return false; } } return true; }</pre>

Consistent indentation makes the nesting visible at a glance, so the reader can understand the structure before reading each expression.

The same principle applies to nested loops, try-catch blocks, methods, classes, and any other construct that introduces a **scope**. Each nesting level should be visually offset from the level above it.

Teams also need a consistent choice of **whitespace characters**. A project may use tabs, spaces, or tooling that converts one to the other. What matters most is that the repository does not contain a mixture that causes alignment to change between editors. Configure the development environment so the agreed choice is applied automatically.

Structuring code blocks safely

In languages that use **braces** to delimit code blocks, a useful safety practice is to keep the braces even when the block contains only one statement. Omitting them may save two lines, but it creates an easy path to a bug when another statement is added later.

Without braces, only the first statement belongs to the loop:

```
for (int i = 0; i < numbers.Count; i++)
    Console.WriteLine("Do something");
    Console.WriteLine("Do something else");
```

If both statements are intended to repeat, the block should be explicit:

```
for (int i = 0; i < numbers.Count; i++)
{
    Console.WriteLine("Do something");
    Console.WriteLine("Do something else");
}
```

The braces make the scope unmistakable and remove the need to remember to add them when the block grows.

A second useful signal is **block size**. If a loop or conditional is so large that its opening and closing braces cannot be seen together on a reasonable screen, the reader must scroll to recover the structure. That usually indicates an opportunity to extract part of the block into a well-named method. The common guideline is 120 characters of width and 20 lines of height, which is roughly what fits on one screen without scrolling in either direction.

Breaking long lines clearly

The 120-character limit deserves a closer look, since **line length** affects readability in its own right. Long lines force horizontal scrolling and make related expressions harder to compare.

Every project can set its own limit, but what matters more than the exact number is breaking lines at points that are meaningful and consistent across the codebase. Consider a condition from a tic-tac-toe implementation:

```
if (gameArray[i, 0] == c && gameArray[i, 1] == c && gameArray[i, 2] == c)
{
    return true;
}
```

Breaking each related condition in the same way makes the comparison easier:

```
if (gameArray[i, 0] == c &&
    gameArray[i, 1] == c &&
    gameArray[i, 2] == c)
{
    return true;
}
```

Parameter lists benefit from the same treatment. When a signature gets crowded, giving each parameter its own line creates a predictable pattern. And it is worth going all the way: either every parameter sits on its own line, or none of them do. Three on one line and two on the next is harder to scan than any consistent choice.

```
bool IsWithinRange(
    int min,
    int max,
    int value,
    bool isInclusive)
```

Method chains are another natural break point. A short chain of two calls reads fine on one line. Once it grows, break it, and again break all of it rather than some of it. Each transformation goes on its own line, with the dot starting the line:

```
return numbers
    .Where(number => number % 2 != 0)
    .Select(number => number * number)
    .Take(3);
```


Situation	Preferred break pattern	Why it helps
Boolean expression	Put comparable conditions on separate, aligned lines.	Differences between conditions are easier to see.
Long parameter list	Place parameters on separate lines using one consistent indent.	The signature becomes easier to scan and edit.
Method chain	Place each chained operation on its own line.	The transformation pipeline becomes visible step by step.
Long interpolated or concatenated expression	Break at logical components rather than arbitrary character positions.	The visual split follows the meaning of the expression.

Table 4.3: Useful places to break long lines

Do not wait until a line physically exceeds the configured limit if an earlier break makes the code clearer. Readability is the objective; the character count is a guardrail.

Using vertical space to organize code

Empty lines serve a role similar to paragraph breaks in prose. They separate logical units and help the reader see which statements belong together. Two methods, for example, should normally be visually separated because they are distinct units of behavior.

```

bool AreAllEven(List<int> numbers)
{
    // ...
}

void CountEvenAndOdd(List<int> numbers)
{
    // ...
}

```

Within a single construct, however, unnecessary blank lines can fragment related code. An if and its else branch belong together and should normally remain visually connected.

Blank lines are also useful inside a larger method when they mark meaningful stages. The following method becomes easier to scan when declarations, iteration, and final output are separated:

```
void CountEvenAndOdd(List<int> numbers)
{
    int totalEven = 0;
    int totalOdd = 0;

    for (int i = 0; i < numbers.Count; i++)
    {
        Console.WriteLine($"Processing number at index {i}");

        if (numbers[i] % 2 == 0)
        {
            totalEven++;
        }
        else
        {
            totalOdd++;
        }
    }

    Console.WriteLine($"Total even numbers: {totalEven}");
    Console.WriteLine($"Total odd numbers: {totalOdd}");
}
```

The exact placement of blank lines is partly a style decision. Some teams place an empty line before a final return statement in larger methods; others omit it in very small methods. Once again, consistency is more valuable than insisting that one preference is universally correct.

One rule worth keeping simple is the number of consecutive blank lines. A single empty line is enough to separate concepts. Multiple empty lines lengthen files without adding more meaning and increase the amount of scrolling required.

Refactoring case study: Formatting the Hangman game

Let's now apply the formatting principles from this chapter to a small Hangman program. We'll begin by understanding how the original code works, then identify its formatting problems and improve it step by step.

Understanding the code

Here is a little program that lets us play Hangman:

```
Console.WriteLine("Let's play Hangman! Enter a word to guess:");
string word = Console.ReadLine();
Console.Clear();
HangmanGame hangmanGame = new HangmanGame(word);
hangmanGame.Play();
Console.ReadKey();
class HangmanGame
{
    private string _wordToGuess;
    private HashSet<char> _guessedLetters = new HashSet<char>();
    private const int MaxAttempts = 6;

    public HangmanGame(string word) { _wordToGuess = word.ToLower();}

    public void Play()
    {
        var usedAttempts = 0;
        while (usedAttempts < MaxAttempts && !IsWordGuessed())
        {
            PrintWord();
            Console.WriteLine($"Attempts remaining: {MaxAttempts -
usedAttempts}");
            Console.Write("Guess a letter: ");
            char currentGuess = ReadSingleChar();
            Console.WriteLine();
            if (_guessedLetters.Contains(currentGuess))
                Console.WriteLine("You've already guessed that letter. Try
again.");
            else
            {

```

```
        _guessedLetters.Add(currentGuess);
        if (!_wordToGuess.Contains(currentGuess))
        {
            usedAttempts++;
            Console.WriteLine("Wrong guess!");
        }
    }
    Console.WriteLine();
}
PrintWord();
if (IsWordGuessed())

Console.WriteLine("Congratulations! You've won the game! Press any key to
close.");
else
    Console.WriteLine($"Game over :( The word was: {_wordToGuess}. Press
any key to close.");
}

private void PrintWord()
{
    foreach (char letter in _wordToGuess)
    {
        if (_guessedLetters.Contains(letter)) Console.Write(letter + " ");
        else Console.Write("_ ");
    }
    Console.WriteLine();
}

private bool IsWordGuessed()
{
    foreach (char letter in _wordToGuess)
    {
        if (!_guessedLetters.Contains(letter))
            return false;
    } return true;
}

private static char ReadSingleChar() { return
Console.ReadKey().KeyChar.ToString().ToLower()[0];}
}
```

The formatting here makes this harder to read than it needs to be, and we will come back to that. First let's work out what the program actually does.

Hangman is a guessing game: a player picks a word, the other guesses it letter by letter, and a wrong guess costs one of a limited number of attempts.

A correct guess reveals every occurrence of that letter and costs nothing. A wrong guess prints a message and takes one attempt away. Guessing a letter that has already been revealed prints a warning but does not count as an invalid guess. The game ends either when the whole word is revealed, with a congratulations message, or when the attempts run out, with a message saying the word was not guessed.

None of that is hard to understand. The logic is reasonable and even the naming is decent. It is the formatting that stands between the reader and the program.

Identifying the issues

The formatting here is poor in almost every way it can be. Lines run too long, indentation is inconsistent, some single-line conditions omit braces, and spacing does little to separate logical stages. The result is a program that is hard to follow even though it is short.

Improving the code

Let's now bring the formatting principles together in a practical refactoring of the Hangman program.

Apply the formatter before editing manually

The first step is to let the IDE normalize what it can: indentation, spacing, and brace alignment. This removes mechanical inconsistencies quickly and gives us a stable baseline for the changes that require judgment.

Automatic reformatting will not be enough, though. Let's consider this compacted constructor:

```
public HangmanGame(string word) { _wordToGuess = word.ToLower(); }
```

This constructor would read better with the braces and the body on separate lines, and with a blank line separating it from the field declarations above it.

```
private string _wordToGuess;  
private HashSet<char> _guessedLetters = new HashSet<char>();  
private const int MaxAttempts = 6;  
  
public HangmanGame(string word)
```

```
{  
    _wordToGuess = word.ToLower();  
}
```

Make scopes explicit

The `Play` method contains conditions without braces. Add braces consistently so that later edits cannot accidentally move statements outside the intended branch. At the same time, add blank lines where they separate major stages of the loop: displaying status, reading a guess, and processing that guess.

```
PrintWord();  
Console.WriteLine($"Attempts remaining: {MaxAttempts - usedAttempts}");  
Console.Write("Guess a letter: ");  
char currentGuess = ReadSingleChar();  
Console.WriteLine();  
  
if (_guessedLetters.Contains(currentGuess))  
{  
    Console.WriteLine("You've already guessed that letter. Try again.");  
}  
else  
{  
    _guessedLetters.Add(currentGuess);  
    if (!_wordToGuess.Contains(currentGuess))  
    {  
        usedAttempts++;  
        Console.WriteLine("Wrong guess!");  
    }  
}
```

Extract code when formatting reveals an oversized block

Once the basic formatting is corrected, the `while` loop is still long enough to obscure the high-level flow of `Play`. This is where formatting becomes a diagnostic tool: if a logical block remains too large to scan comfortably, extract it into a method whose name explains the step.

Most IDEs can help us extract a method from a selected piece of code. In this case, we want to achieve this shape of the `Play` method:

```
public void Play()  
{
```

```
var usedAttempts = 0;

while (usedAttempts < MaxAttempts && !IsWordGuessed())
{
    usedAttempts = AttemptToGuessLetter(usedAttempts);
}

PrintGameResult();
}
```

AttemptToGuessLetter could be implemented like this:

```
private int AttemptToGuessLetter(int usedAttempts)
{
    PrintGameStatus(usedAttempts);
    char currentGuess = GuessLetter();

    if (_guessedLetters.Contains(currentGuess))
    {
        Console.WriteLine("You've already guessed that letter. Try again.");
    }
    else
    {
        _guessedLetters.Add(currentGuess);
        if (!_wordToGuess.Contains(currentGuess))
        {
            usedAttempts++;
            Console.WriteLine("Wrong guess!");
        }
    }

    Console.WriteLine();
    return usedAttempts;
}
```

Notice the consistent formatting and the short length of this method. It uses several small, focused, and easy-to-read helper methods:

```
private void PrintGameStatus(int usedAttempts)
{
    PrintWord();
}
```

```
        Console.WriteLine($"Attempts remaining: {MaxAttempts - usedAttempts}");
    }

    private static char GuessLetter()
    {
        Console.Write("Guess a letter: ");
        char currentGuess = ReadSingleChar();
        Console.WriteLine();
        return currentGuess;
    }

    private void PrintWord()
    {
        foreach (char letter in _wordToGuess)
        {
            if (_guessedLetters.Contains(letter))
            {
                Console.Write(letter + " ");
            }
            else
            {
                Console.Write("_ ");
            }
        }
        Console.WriteLine();
    }

    private static char ReadSingleChar()
    {
        return Console.ReadKey().KeyChar.ToString().ToLower()[0];
    }
}
```

Extract code when formatting reveals an oversized block

We also need a dedicated method for printing the game result:

```
private void PrintGameResult()
{
    PrintWord();

    if (IsWordGuessed())
    {

```



```
        Console.WriteLine(
            "Congratulations! You've won the game!");
    }
    else
    {
        Console.WriteLine(
            $"Game over :( The word was: {_wordToGuess}.");
    }

    Console.WriteLine("Press any key to close.");
}

private bool IsWordGuessed()
{
    foreach (char letter in _wordToGuess)
    {
        if (!_guessedLetters.Contains(letter))
        {
            return false;
        }
    }
    return true;
}
```

That covers the whole file. Looking at the result, the program is easier to read than it was. It is longer now, but *the extra length is what makes it readable*: the earlier version was compressed to the point where the structure disappeared. Now lines fit on the screen, and the logical parts of the program are separated visually as well as logically.

Original problem	Refactoring response	Result
Compacted declarations and inconsistent spacing	Run the IDE formatter and normalize braces and indentation.	Mechanical inconsistencies are removed quickly
Single-line conditions without braces	Add braces consistently.	Scopes remain explicit and safer to extend
Large while loop body	Extract AttemptToGuessLetter.	Play shows the high-level game flow

Original problem	Refactoring response	Result
Several unrelated visual sections in one method	Extract <code>PrintGameStatus</code> , <code>GuessLetter</code> , and <code>PrintGameResult</code> .	Each section has a clear name and purpose
Long lines and dense blocks	Break lines and use one blank line between logical units.	The code becomes easier to scan and modify

Table 4.4: Formatting problems addressed in the Hangman refactoring

Summary

This chapter showed how formatting makes the structure of a program visible. We looked at the difference between syntax and style conventions, at the tools that can enforce consistency automatically, and at what predictable indentation and brace placement do for a reader. We also covered where to break long expressions and how vertical spacing separates one idea from the next. The Hangman refactoring showed how consistent formatting can reveal opportunities for small structural improvements while preserving behavior. In *Chapter 5*, you will move from presentation to design and explore the SOLID principles, starting with the Single Responsibility Principle.

5

Applying the Single Responsibility Principle

Classes often grow by accumulation. A class may be written with one clear job, but then a new requirement comes along, and the easiest place to put it is a class that already exists and is roughly related. Nobody makes a decision to give it a second **responsibility**. It just happens, one small addition at a time, until the class does several unrelated things and a change to one of them can break the others. The **Single Responsibility Principle (SRP)** provides a practical way to reason about that. Rather than demanding one method per class, SRP asks whether the class serves one coherent purpose and has *one meaningful reason to change*.

This chapter introduces the SOLID principles family and then develops SRP through practical examples. *The following chapters cover the remaining four principles one at a time. Chapter 9 then brings all five principles together in a single refactoring case study, showing how they can work together in practice.*

In this chapter, we will cover the following topics:

- Introducing the SOLID principles
- Understanding the Single Responsibility Principle
- Recognizing classes with multiple responsibilities
- Separating unrelated responsibilities
- Using one reason to change as an SRP test
- Applying a practical SRP checklist

By the end of this chapter, you will be able to explain the Single Responsibility Principle, identify classes that combine unrelated responsibilities, separate those responsibilities into focused types, and use the one-reason-to-change formulation to evaluate class boundaries.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Introducing the SOLID principles

SOLID is a **mnemonic** for five object-oriented design principles that help keep software understandable, maintainable, extensible, and testable. *The principles are not mechanical rules that guarantee good design.* Instead, they give you a vocabulary for recognizing common design problems and reasoning about better alternatives. *Table 5.1* summarizes the focus of each principle.

Letter	Principle	Core concern
S	Single Responsibility Principle	Keep a class focused on one coherent responsibility.
O	Open-Closed Principle	Make code extensible without repeatedly modifying it.
L	Liskov Substitution Principle	Ensure subtypes can safely stand in for their base types.
I	Interface Segregation Principle	Avoid forcing clients to depend on operations they do not need.
D	Dependency Inversion Principle	Depend on abstractions rather than concrete implementation details.

Table 5.1: The five SOLID principles

This chapter is about the Single Responsibility Principle. The chapters that follow take the remaining SOLID principles one at a time.

Understanding the Single Responsibility Principle

The **Single Responsibility Principle** states that a class should have one coherent responsibility. You will also encounter a common formulation: *a class should have no more than one reason to change*. Both formulations are useful, but the word responsibility needs careful interpretation. A responsibility is the coherent purpose that the class serves.

Why combining responsibilities creates problems

A simple analogy is to imagine a jacket with a backpack sewn onto it. A jacket keeps you warm; a backpack carries your belongings. You may sometimes use both at the same time, but that does not mean they should be one object. If the weather is warm, you may want the backpack without the jacket. If you need more carrying capacity, you may want a different backpack without replacing the jacket. Combining them removes that flexibility.

The same problem appears in software when a class combines unrelated work. Consider a class that reads personal data from a **SQL database** and also formats a person as text. A client that needs only database access is forced to depend on formatting code as well, and a client that needs only formatting must know about a data-access class.

The following class illustrates the problem:

```
public class PersonalDataAccess
{
    private readonly string _connectionString;

    public PersonalDataAccess(string connectionString)
    {
        _connectionString = connectionString;
    }

    public List<Person> GetAll()
    {
        var people = new List<Person>();
        using SqlConnection connection = new SqlConnection(_connectionString);
        connection.Open();
        string query = "SELECT Id, FirstName, LastName, Age FROM People";
        using SqlCommand command = new SqlCommand(query, connection);
        using SqlDataReader reader = command.ExecuteReader();
        while (reader.Read())
```

```

    {
        var person = new Person
        {
            Id = reader.GetInt32(0),
            FirstName = reader.GetString(1),
            LastName = reader.GetString(2),
            Age = reader.GetInt32(3)
        };
        people.Add(person);
    }
    return people;
}

public string FormatPersonalData(Person person)
{
    return $"ID: {person.Id}, " +
        $"Name: {person.FirstName} {person.LastName}, " +
        $"Age: {person.Age}";
}
}

```

The class has two independent responsibilities: obtaining the data and formatting it. That comes at a cost, and one of those costs is worth going back to the jacket and backpack analogy for.

Say we want a light jacket, a winter jacket, and a rain jacket, and a small, a medium, and a large backpack. Kept separate, that is six things. Sewn together, every jacket needs a version with each backpack, so we end up with nine, and each new jacket adds three more. *Classes behave the same way*. Once responsibilities are mixed, the number of classes we need can grow very fast. *Table 5.2* summarizes the main design costs of combining these responsibilities.

Problem	Why it matters
Forced dependency	A client that needs one responsibility must still depend on the class that contains both.
Reduced flexibility	Changing SQL access to another source requires modifying the same class that also contains the formatting behavior.

Problem	Why it matters
More combinations	Combining two responsibilities in one class means a separate class for every combination of them, so the number of classes multiplies instead of adding up.
Harder maintenance	A change to one responsibility marks the whole class as changed and may require broader review, retesting, and redeployment.
Hidden interaction risk	Methods for unrelated responsibilities can accidentally share state or depend on one another.
Less focused tests	A test fixture for the class may need to accommodate unrelated behaviors.
Poorer naming	Names such as <code>PeopleDataReaderAndFormatter</code> are unwieldy, while names such as <code>PeopleManager</code> are too vague.
More merge conflicts	Developers changing separate responsibilities are more likely to touch the same file and class.

Table 5.2: Consequences of combining data access and formatting

These problems are not caused by the number of lines in the class. *They come from the fact that two independent reasons for change are packaged together.*

Separating the responsibilities

The refactoring is straightforward once the responsibilities are clear. Move the formatting behavior into a dedicated class. To keep the focus on the split, the repeated method bodies are abbreviated below:

```
public class PersonalDataFormatter
{
    public string FormatPerson(Person person)
    {
        // Formatting Logic omitted for brevity.
    }
}
```


The original class can then be renamed so that its remaining responsibility is explicit:

```
public class PeopleDataReader
{
    private readonly string _connectionString;

    public PeopleDataReader(string connectionString)
    {
        _connectionString = connectionString;
    }

    public List<Person> GetAll()
    {
        // Data-access logic omitted for brevity.
    }
}
```

The two classes can still be used together when the application needs both operations:

```
var peopleDataReader = new PeopleDataReader("some connection string");
var people = peopleDataReader.GetAll();
var firstPerson = people.First();
var personalDataFormatter = new PersonalDataFormatter();
Console.WriteLine(
    "First person's data: " +
    personalDataFormatter.Format(firstPerson));
```

The important change is not merely that the code has been split. *Each new class has a precise name, can be reused independently, can be tested around one purpose, and can change without forcing unrelated behavior to change with it.*

A single responsibility does not mean a single method

Inexperienced developers often assume the Single Responsibility Principle is about counting public methods. If a class exposes two of them, it does two things, so it must be violating the principle. That is not what the principle says. A class can expose many public methods and still have one responsibility, as long as they all serve the same purpose. Data structures can serve as a useful example. Consider the `List<T>` class in C#, which represents a **dynamic collection** of items. It has many methods, such as `Add`, `Remove`, `Clear`, and `Sort`. That does not mean it has more than one responsibility. Its responsibility is to provide a dynamic structure for storing

multiple items of the same type, and covering that takes more than one method. A simplified sketch of the relevant **API** looks like this:

```
public class List<T>
{
    public void Add(T item);
    public void Remove(T item);
    public void Clear();
    public void Sort();
    // and more
}
```

Splitting every operation into its own class would be a terrible idea. It would give us classes such as `ListItemsAdder`, `ListItemsRemover`, `ListClearer`, and `ListSorter`—four types working on the same data and useless apart from each other. These operations belong together because they all serve the same purpose: managing a list. There is also a practical cost: those classes would need some way to reach and modify the list's state, introducing extra **coupling** and weakening the collection's control over its own invariants.

A **repository** provides another example. A `PeopleRepository` may legitimately add, update, delete, and retrieve people because all of those methods serve one responsibility: managing the persistence of `Person` objects. A simplified repository API could look like this:

```
public class PeopleRepository
{
    public void AddPerson(Person person);
    public void UpdatePerson(Person person);
    public void DeletePerson(int personId);
    public Person GetPersonById(int personId);
    public List<Person> GetAllPeople();
}
```

Orchestration can still be one responsibility

Consider a `UserAuthorizer` class and its `IsAuthorized` method. Given a user ID and an action type, it reads the user from a repository, reads the list of actions that user can perform, and checks whether any of them matches the requested type. In later chapters, we will decouple

this further using interfaces, but concrete delegation still demonstrates single responsibility at this stage.

```
public class UserAuthorizer
{
    private readonly UsersRepository _usersRepository;
    private readonly ActionsRepository _actionsRepository;

    public UserAuthorizer(
        UsersRepository usersRepository,
        ActionsRepository actionsRepository)
    {
        _usersRepository = usersRepository;
        _actionsRepository = actionsRepository;
    }

    public bool IsAuthorized(int userId, ActionType actionType)
    {
        var user = _usersRepository.GetById(userId);
        var allowedActions = _actionsRepository.GetAllowedFor(user);
        return allowedActions.Any(action => action.Type == actionType);
    }
}
```

One could argue that this class does three things instead of one. It reads a user, it reads actions, and it compares them. Three steps, so some developers may see this as an SRP violation.

Not necessarily. The class still has one responsibility, which is deciding whether a user is authorized to perform an action. Getting there takes several steps, but the class does not carry them out itself. Data access is delegated to repositories, and `UserAuthorizer` only orchestrates the steps. *Its responsibility has not changed.*

This kind of class is an **orchestrator**. Its methods may invoke several collaborators, but the class still has a clear purpose. *SRP asks whether those operations belong to one responsibility, not whether there is only one operation.*

Using one reason to change as an SRP test

The alternative formulation of SRP, that a class should have only one reason to change, can make class boundaries easier to evaluate. The earlier `PersonalDataAccess` class has at least two independent reasons to change. The way data is read may change, for example from SQL to

Excel, and the way a person is formatted may change, for example by omitting the ID or changing the textual representation.

```
public class PersonalDataAccess
{
    public List<Person> GetAll();
    public string FormatPersonalData(Person person);
}
```

After the refactoring and splitting this type into two independent classes, the problem is gone: `PeopleDataReader` changes when data-access requirements change, while `PersonalDataFormatter` changes when formatting requirements change. Each class now responds to a different axis of change.

The same reasoning explains why `List<T>` can still follow SRP. Improving `Add` performance or strengthening `Remove` error handling are both changes to the same responsibility: making the list a better data structure.

`UserAuthorizer` works the same way. Its responsibility is to check whether a user may perform an action, and to do that it uses a few other classes, such as the users repository and the actions repository. How many reasons to change does it have? One. The flow of the checks may change, for example when we add a verification that the user with the given ID exists, or that the user is an administrator.

One could think it has more. What if we switch to a different database? But connecting to a database is the job of the repositories. Changing the database is their reason to change, not the authorizer's, and when that change happens, the code of `UserAuthorizer` stays exactly as it is.

Tip

***Notice who asks for changes** to a class. If the requests keep coming from two different people, each responsible for a different part of the business, the class likely has two reasons to change.*

A practical SRP checklist

SRP violations become easier to spot with experience, and *Table 5.3* provides a practical set of questions for evaluating class boundaries against SRP.

Question	What to look for
Can you describe the responsibility in one brief sentence?	If the description naturally becomes "does X and Y," the class may contain unrelated responsibilities.
Does splitting the class improve the design?	A good split usually produces independent, focused, well-named types.
Do the proposed new classes become tightly coupled?	If they constantly need each other's internals, the original behavior may belong together.
Are there multiple independent reasons to change?	Changes driven by unrelated requirements are a strong sign of multiple responsibilities.
Is the class merely orchestrating collaborators?	Coordination can still be one responsibility when the detailed work is delegated.
Does naming the class require vague words such as Manager?	Vague or compound names can signal that the class has accumulated too much.

Table 5.3: Questions for evaluating a class against SRP

A refactoring experiment can also be informative. *If dividing a class produces two clean, independent abstractions, the original boundary was probably too broad.* If the split feels artificial and the new classes cannot operate without extensive knowledge of one another, revert it and reconsider the responsibility.

Summary

This chapter introduced SOLID and focused on the Single Responsibility Principle. You saw why unrelated responsibilities create forced dependencies, poorer naming, broader tests, and more reasons for a class to change. Refactoring data access and formatting into separate types made those responsibilities explicit, while examples such as repositories and orchestrators showed why one responsibility does not mean one method.

In *Chapter 6*, you will examine the Open-Closed Principle and learn how abstractions can accommodate expected variation while reducing the need to modify existing code.

6

Applying the Open-Closed Principle

Software development is a continuous process. It almost never happens that we learn all the requirements before starting a project and that nothing changes along the way. Usually, we get a rough idea of what we are building and detailed requirements for a few basic features. The rest arrives as the work progresses, and some of it replaces what we were told earlier, sometimes reversing a decision completely.

Releasing the application does not end the process either. Think about services like YouTube or Instagram. They change constantly. Features are added, bugs are fixed, and interfaces are redesigned.

So, *change is not an exception we prepare for just in case. It is the normal state of a codebase*, and our code has to be ready for it. That is what the **Open-Closed Principle (OCP)** is about.

This principle states that modules, classes, and methods should be ***open for extension but closed for modification***.

This chapter starts with a review of **abstraction** and **polymorphism**, the two concepts needed to understand the Open-Closed Principle. It then looks at a type that breaks the principle and works through a possible refactoring. It closes with the practical limits of the principle, so that it stays a design aid rather than a rigid rule. In this chapter, we will cover the following topics:

- Reviewing abstract types and polymorphism
- Understanding the Open-Closed Principle
- Recognizing an OCP violation in existing code
- Refactoring behavior behind an abstraction
- Containing unavoidable changes in factories and selection logic

- Recognizing when an abstraction itself must change
- Balancing OCP with performance, bug fixes, and major requirement changes
- Relating the Open-Closed and **Single Responsibility Principles**

By the end of this chapter, you will understand how abstraction and polymorphism make code extensible, be able to recognize when the Open-Closed Principle is being broken, and know how to refactor such code so that new behavior can be added without modifying what already exists. You will also know where the principle has its limits, and when changing existing code is the right decision anyway.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Reviewing abstract types and polymorphism

The **OCP** is easier to apply once we are comfortable with abstract types and polymorphism. Many modern languages let us define abstract types that describe a **contract** independently of a particular implementation. In C#, an **interface** is one way to express such a contract. In the simple form used here, the interface declares member signatures, while implementing types provide the corresponding behavior. Code that depends on the interface can work with any implementation that satisfies that contract. For example:

```
public interface SomeInterface
{
    void SomeMethod();
    int SomeOtherMethod(string param);
}
```

A concrete class that implements this interface provides implementations for its declared methods:

```
public class SomeClass : SomeInterface
{
    public void SomeMethod()
    {
        // do something
    }

    public int SomeOtherMethod(string param)
    {
        // do something
        return 0;
    }
}
```

Interfaces are one way C# supports polymorphism: code can work with a common type while each concrete implementation behaves in its own way. Code holding a `SomeInterface` reference can call `SomeMethod` and `SomeOtherMethod` on it without knowing whether the object behind it is a `SomeClass` or some other implementation. The same applies to interfaces from the standard library. A method that accepts an `ICollection<int>` works with any collection satisfying that contract. In both cases, code holding a value through the interface can rely only on the operations the interface declares, and not on anything specific to the implementation behind it.

Note

C# convention: Interface names conventionally begin with the letter I, as in `ICollection`, `IAccount`, or `IVideosRepository`.

This ability to write code against a stable abstraction is central to OCP. New concrete implementations can often be added later without rewriting the code that depends on the abstraction.

Understanding the Open-Closed Principle

Requirements change throughout the life of an application. New features arrive, existing behavior is refined, and assumptions that once seemed stable turn out not to be. *The goal is to*

design the system so that when change comes, its impact stays contained. The Open-Closed Principle states that modules, classes, and methods should be open for extension but closed for modification. In practical terms, you should prefer designs in which new behavior can be introduced by adding new code instead of repeatedly editing code that already works.

Modification is not wrong in itself, but it carries risk. Existing code may be used elsewhere in the application, or by consumers outside it, so changing it can introduce defects, surprise developers relying on the current behavior, or break **backward compatibility**. Automated tests reduce that risk. They do not remove it.

Note

Open-Closed Principle: Prefer adding a new implementation behind a stable abstraction when a predictable family of behaviors is expected to grow. *Do not interpret the principle as a ban on ever editing existing code.*

Recognizing an OCP violation in account behavior

Consider a video-streaming application with free and premium accounts. Premium users can access all videos, while free users can access only videos marked as free. Here is the code implementing this logic:

```
public class User
{
    public AccountType AccountType { get; }
    private readonly IVideosRepository _videosRepository;

    public User(AccountType accountType, IVideosRepository videosRepository)
    {
        AccountType = accountType;
        _videosRepository = videosRepository;
    }

    public List<Video> GetAvailableVideos()
    {
        if (AccountType == AccountType.Premium)
        {
            return _videosRepository.GetAll();
        }

        return _videosRepository
```

```
        .GetAll()  
        .Where(video => video.IsFree)  
        .ToList();  
    }  
}  
  
public enum AccountType  
{  
    Free,  
    Premium  
}
```

The design appears manageable until the business introduces a Kids account. Supporting the new account requires modifying the enum, the video data structure, and, crucially, the existing `GetAvailableVideos` method:

```
public List<Video> GetAvailableVideos()  
{  
    if (AccountType == AccountType.Premium)  
    {  
        return _videosRepository.GetAll();  
    }  
  
    if (AccountType == AccountType.Kids)  
    {  
        return _videosRepository  
            .GetAll()  
            .Where(video => video.IsForKids)  
            .ToList();  
    }  
  
    return _videosRepository  
        .GetAll()  
        .Where(video => video.IsFree)  
        .ToList();  
}
```

Each new account category extends the same conditional. In a larger system, equivalent branching may be repeated in many places. The code is therefore not closed for modification: every new account type pushes another change into existing behavior.

Refactoring account behavior behind an abstraction

A better design makes account behavior polymorphic. Start with an interface that represents the behavior shared by account types:

```
public interface IAccount
{
    AccountType AccountType { get; }
    List<Video> GetAvailableVideos();
}
```

FreeAccount and PremiumAccount can then implement the behavior appropriate to each account type:

```
public class FreeAccount : IAccount
{
    public AccountType AccountType { get; } = AccountType.Free;
    private readonly IVideosRepository _videosRepository;

    public FreeAccount(IVideosRepository videosRepository)
    {
        _videosRepository = videosRepository;
    }

    public List<Video> GetAvailableVideos()
    {
        return _videosRepository
            .GetAll()
            .Where(video => video.IsFree)
            .ToList();
    }
}

public class PremiumAccount : IAccount
{
    public AccountType AccountType { get; } = AccountType.Premium;
    private readonly IVideosRepository _videosRepository;

    public PremiumAccount(IVideosRepository videosRepository)
    {
        _videosRepository = videosRepository;
    }
}
```

```
    }

    public List<Video> GetAvailableVideos()
    {
        return _videosRepository.GetAll().ToList();
    }
}
```

User no longer decides which filtering rule applies. It depends only on IAccount and delegates the operation:

```
public class User
{
    private readonly IAccount _account;

    public User(IAccount account)
    {
        _account = account;
    }

    public List<Video> GetAvailableVideos()
    {
        return _account.GetAvailableVideos();
    }
}
```

A new KidsAccount can now be added as another implementation of the same abstraction:

```
public class KidsAccount : IAccount
{
    public AccountType AccountType { get; } = AccountType.Kids;
    private readonly IVideosRepository _videosRepository;

    public KidsAccount(IVideosRepository videosRepository)
    {
        _videosRepository = videosRepository;
    }

    public List<Video> GetAvailableVideos()
    {
        return _videosRepository

```

```
        .GetAll()  
        .Where(video => video.IsForKids)  
        .ToList();  
    }  
}
```

The existing `User`, `FreeAccount`, and `PremiumAccount` classes do not need to change when `KidsAccount` is introduced. The system is extended by adding a new implementation.

The `AccountType` enum still needs a localized modification to add `Kids`, so this design is not completely modification-free. The important improvement is that the account-specific behavior is added through `KidsAccount` without reopening the existing account implementations.

This design also makes account-specific behavior easier to test because each filtering rule has its own focused type. **SRP** and **OCP** reinforce one another here: each account class has a clear responsibility, and new account behavior can be introduced through extension.

Understanding the practical limits of OCP

The Open-Closed Principle is a design goal, not a promise that existing code will never change. Some modifications are unavoidable, and forcing every requirement into an extension-only design can produce more complexity than value. The useful question is where change should be contained and whether the abstraction reflects real variation in the domain.

Containing unavoidable changes in a factory

Even after introducing `IAccount`, the application still needs some code that chooses which concrete account type to create. If a user selects an account type during registration, some selection logic must map that choice to a concrete object. A direct conditional might look like this:

```
IAccount CreateAccount(AccountType accountType)  
{  
    var videosRepository = new VideosRepository();  
  
    if (accountType == AccountType.Free)  
    {  
        return new FreeAccount(videosRepository);  
    }  
  
    if (accountType == AccountType.Kids)
```

```
{  
    return new KidsAccount(videosRepository);  
}  
  
return new PremiumAccount(videosRepository);  
}
```

Adding a new account type still requires this selection logic to change. That does not invalidate the broader refactoring. The important improvement is that the unavoidable modification is isolated to one small place instead of being repeated throughout the application.

A dedicated **factory** makes that responsibility explicit:

```
public class AccountsFactory  
{  
    public IAccount CreateAccount(AccountType accountType)  
    {  
        var videosRepository = new VideosRepository();  
  
        switch (accountType)  
        {  
            case AccountType.Free:  
                return new FreeAccount(videosRepository);  
            case AccountType.Kids:  
                return new KidsAccount(videosRepository);  
            case AccountType.Premium:  
                return new PremiumAccount(videosRepository);  
            default:  
                throw new ArgumentOutOfRangeException(nameof(accountType));  
        }  
    }  
}
```

Note

Design principle: *If some modification cannot be eliminated, contain it.* A small, explicit factory is easier to reason about than account-type conditionals scattered through the codebase.

When the abstraction itself must change

Another limitation appears when a new requirement changes the contract shared by every implementation. Suppose video availability must now depend on the user's region. The original `IAccount` contract exposes a parameterless `GetAvailableVideos` method, so it cannot express the new input.

```
public interface IAccount
{
    AccountType AccountType { get; }
    List<Video> GetAvailableVideos();
}
```

One reasonable response is to change the contract so the region becomes explicit:

```
public interface IAccount
{
    AccountType AccountType { get; }
    List<Video> GetAvailableVideos(string region);
}
```

Every implementation must then be updated, for example:

```
public List<Video> GetAvailableVideos(string region)
{
    return _videosRepository
        .GetAll(region)
        .Where(video => video.IsForKids)
        .ToList();
}
```

This is a modification of existing code, so the design is not closed to this particular kind of change. The important lesson is that *no abstraction can anticipate every future requirement*. Trying to predict all possible variations in advance often creates abstractions that are more complicated than the real problem and may still fail to match the eventual requirement.

Instead, design around variations that are currently meaningful and reasonably likely. When a new axis of change appears, modify the design carefully and use the experience to decide whether the abstraction should evolve.

Do not sacrifice performance for formal compliance

It may be possible to avoid changing `IAccount` by loading all videos first and introducing a new `VideosFilterer` that filters by region afterward. That approach adds new code without modifying existing code, so it appears to satisfy OCP. However, it can be a poor engineering decision if filtering after retrieval causes unnecessary data transfer and slower execution.

```
var youngUser = new User(new KidsAccount(videosRepository));
var availableVideos = youngUser.GetAvailableVideos();

availableVideos = new VideosFilterer()
    .FilterByRegion("Australia", availableVideos);
```

When the repository can translate filtering into a database-side query, applying the region restriction before materializing the data is generally more efficient. Once all data has already been loaded into the application, later filtering occurs in memory and may transfer far more data than necessary.

The broader principle is that *clean design must serve the software, not the other way around*. A design that technically avoids modifying existing code but introduces a significant performance problem is not an improvement.

When changing existing code is the right choice

Some requirements are so different from the original model that preserving the old abstraction would leave obsolete concepts embedded in the design. If a streaming platform abandons account-based access entirely and moves to free or pay-per-view videos, retaining the previous account hierarchy merely to avoid modification may produce convoluted code. In such cases, redesigning or replacing parts of the system can be the cleaner option, provided the change is planned, tested, and protected by version control.

Bug fixes are even clearer. *If existing code is wrong, correct it*. Do not create a parallel subtype that reproduces the old behavior except for the defect simply to claim that the original implementation remained closed for modification.

Table 6.2 summarizes how to apply OCP pragmatically when extension is not the only concern.

Situation	Recommended response
A predictable family of behaviors is growing	Introduce or extend an abstraction and add new implementations
Concrete-type selection must change	Isolate the selection logic in a small factory or composition boundary
A new requirement changes the shared contract	Update the abstraction deliberately; do not pretend the new requirement was already modeled
Extension-only design creates a serious performance cost	Prefer the design that meets performance and correctness needs, even if existing code must change
The domain model changes radically	Refactor or redesign the affected modules rather than preserving obsolete abstractions
Existing behavior contains a bug	Modify the code and fix the defect

Table 6.1: Applying the Open-Closed Principle pragmatically

Bringing SRP and OCP together

SRP and OCP address different questions, but they often lead to compatible designs. SRP asks whether a class has one coherent responsibility. OCP asks whether expected variations can be introduced without repeatedly modifying stable behavior. In the streaming example, moving video-selection behavior into separate account classes gives each class a focused purpose and allows new account variants to be added behind `IAccount`.

The principles should not be applied mechanically. Splitting every method into a class would misuse SRP, while creating abstractions for every imaginable future requirement would misuse OCP. *Good design comes from identifying real responsibilities, real variation, and the likely cost of change.*

Summary

This chapter applied the Open-Closed Principle to designs that must accommodate new behavior. You reviewed interfaces and polymorphism, replaced awkward conditionals with an abstraction, and isolated unavoidable concrete-type selection in a small factory. You also examined the limits of OCP: contracts evolve, performance matters, requirements can change radically, and bugs should be fixed rather than protected by formalism.

In *Chapter 7*, you will continue the **SOLID** sequence with the **Liskov Substitution Principle** and ask whether subtypes truly honor the contracts of their base types.

7

Applying the Liskov Substitution Principle

When a class derives from another one, it takes on a promise. *Whatever the base type says it can do, the derived type says it can do as well.* Code written against the base type is entitled to rely on that.

The compiler only checks half of it. A subtype can override every method, match every signature, compile without a complaint, and still refuse to do what the base type promised. It throws where the base type returned a value, or rejects inputs the base type accepted. The hierarchy still looks correct from the outside, so the callers are the ones who pay. They start checking what type they actually received, or wrapping calls in try-catch, or handling one subtype differently from the rest.

The Liskov Substitution Principle asks a simple question about every subtype: *can it stand in for its base type without anything breaking?* This chapter treats **inheritance** as a **contract** about behavior rather than about method signatures. We will go through the ways that contract gets broken, and see how to redesign a hierarchy that no longer tells the truth.

In this chapter, we will cover the following topics:

- Understanding the Liskov Substitution Principle as a contract
- Recognizing common LSP violations
- Refactoring base types that promise too much
- Extracting capabilities into focused interfaces

By the end of this chapter, you will understand what **substitutability** means, and be able to spot subtypes that do not deliver what their base type promises. You will also know how to fix

such a design: how to get rid of the type checks it forces on callers, and how to restructure a hierarchy that puts classes together which do not in fact share the same behavior.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Understanding the Liskov Substitution Principle

The **Liskov Substitution Principle (LSP)** states that an object of a derived type should be usable wherever an object of the base type is expected without causing errors, invalid results, or subtype-specific handling. Code written against the base type should not need to know which concrete subtype it received.

This idea is easiest to understand as a contract. A base type defines what callers are allowed to assume. *A subtype may specialize how the contract is fulfilled, but it must not invalidate the promises made by the base type.*

Note

LSP in practice: Code that consumes a base type should assume nothing beyond what that base type promises, and *every subtype must honor those promises and conventions.*

A memorable substitutability analogy

Imagine we are standing on a tower, in front of a wide open window. I hand you objects one by one, all of them derived from a base `Bird` type, and your job is to throw them out of the window. `Bird` defines a `Fly` method, so every object you get has one. Things go well and the birds fly away, until I hand you an ostrich.

The example is silly, but it shows the Liskov Substitution Principle being broken. You are the method here. You accept `Bird` objects and you handle them according to what that type

promises, which is that they can fly. You do not know their concrete types, and you should not have to. Then one of them fails while being used exactly as the base type said it could be.

The fix starts with recognizing that `Bird` is the wrong parameter type for this job. What you need is not birds but things that fly, so the right abstraction is an `IFlyable` interface. An ostrich would not implement it. Flying birds would, and so would a kite or a drone, and all of them would have been fine going out of that window.

None of this means `Bird` should not exist. It is a useful type whenever we want to treat different birds uniformly. It just should not define a `Fly` method, because not every bird can fly.

You may encounter another classic illustration of this principle: a `Square` class inheriting from a `Rectangle` class. Since a square is mathematically a rectangle with equal sides, it seems natural to model it that way. But if `Rectangle` exposes independent `Width` and `Height` setters, a `Square` subclass must override them so that setting one also changes the other, to preserve the "equal sides" invariant. Code that sets width and expects height to remain unchanged, valid for any `Rectangle`, breaks when given a `Square`. This is the same substitutability problem you just saw with birds and flight.

Recognizing LSP violations

An LSP violation can originate in the subtype or in the abstraction it implements. Either way, the symptom is the same: *callers can no longer rely on the base type's contract without knowing which concrete subtype they received.*

The following sections examine these failure modes individually, including subtypes that cannot honor inherited behavior, client code that checks concrete types, abstractions that expose capabilities some types cannot support, and subtypes that accept a narrower range of valid inputs than the base type.

When a subtype breaks the base-type contract

Consider an abstract `Plane` type that assumes every plane has fuel. It exposes `MaxFuel` and `RemainingFuel` properties and calculates the percentage of fuel remaining. A `CargoPlane` fits that design, but a `ToyPlane` does not. Because `ToyPlane` returns 0 for both fuel values, the inherited calculation divides a floating-point zero by zero and produces an invalid result.

The original base type looks like this:

```
public abstract class Plane
{
    public virtual int MaxFuel => 1000;
    public virtual int RemainingFuel => 1000;
```

```
public float PercentOfRemainingFuel()
{
    return ((float)RemainingFuel / MaxFuel) * 100;
}

public abstract string Land();
}
```

A conventional fueled subtype works as expected:

```
public class CargoPlane : Plane
{
    public override int MaxFuel => 2000;
    public override int RemainingFuel => 2000;

    public override string Land()
    {
        return "Land on the airstrip near the cargo terminal";
    }
}
```

The difficulty appears when ToyPlane is forced into the same contract:

```
public class ToyPlane : Plane
{
    public override int MaxFuel => 0;
    public override int RemainingFuel => 0;

    public override string Land()
    {
        return "Just don't hit the floor too hard.";
    }
}
```

Now a caller that only sees the Plane type can receive a ToyPlane and obtain an invalid result from behavior that the base class appears to guarantee. For a ToyPlane, an attempt to calculate the percent of remaining fuel will produce an invalid result. A caller receiving a Plane has the

right to assume this method will work fine, yet they will be surprised when the Plane object turns out to actually be a ToyPlane.

```
Plane plane = new ToyPlane();  
var remainingFuel = plane.PercentOfRemainingFuel();  
Console.WriteLine(remainingFuel);
```

This example of breaking LSP is a typical case of a subtype breaking the base class contract. The ToyPlane violates the contract that the Plane class defines.

There are several possible remedies. You could change the base type so that fuel is no longer part of the general Plane contract, decide that ToyPlane should not derive from Plane, or change the subtype implementation if the domain actually requires every plane in the system to have meaningful fuel values. *The right choice depends on the application domain.*

Extracting capabilities into interfaces

If both fueled and non-fueled planes are valid in the application, Plane should contain only behavior that is meaningful for every plane. Move the fuel-specific members into an IFuelable interface so that only types that support that capability implement it:

```
public interface IFuelable  
{  
    int MaxFuel { get; }  
    int RemainingFuel { get; }  
}  
  
public abstract class Plane  
{  
    public abstract string Land();  
}  
  
public class CargoPlane : Plane, IFuelable  
{  
    public int MaxFuel => 2000;  
    public int RemainingFuel => 2000;  
  
    public override string Land()  
    {  
        return "Land on the airstrip near the cargo terminal";  
    }  
}
```



```
public class ToyPlane : Plane
{
    public override string Land()
    {
        return "Just don't hit the floor too hard.";
    }
}
```

ToyPlane now remains a Plane but does not implement IFuelable, so it cannot be passed to code that requires fuel-related behavior. *The type system prevents that misuse at compile time instead of allowing an invalid fuel calculation to surface later.*

Moving shared behavior out of inappropriate subtypes

If the same fuel calculation would otherwise be copied into several fuelable classes, place it in a separate service that accepts the capability interface. This avoids duplicating business logic while keeping the abstraction honest:

```
public class FuelCalculator
{
    public float GetPercentOfRemainingFuel(IFuelable fuelable)
    {
        return ((float)fuelable.RemainingFuel / fuelable.MaxFuel) * 100;
    }
}
```

The calculator can now work with a fueled plane, car, boat, or any other object that implements IFuelable. It does not need to know which concrete type it receives.

Avoiding subtype checks in client code

A common warning sign of an LSP problem is client code that checks for specific subtypes before using a base abstraction. For this example, assume IFuelable also declares an Attach(FuelHose hose) operation. A KeroseneLamp contains fuel, but it cannot accept the same hose as a vehicle. A tempting workaround is to detect KeroseneLamp inside FuelAll and run a special path:

```
void FuelAll(List<IFuelable> fuelables, FuelHose hose)
{
    foreach (var fuelable in fuelables)
    {
```

```

        if (fuelable is KeroseneLamp)
        {
            var lamp = fuelable as KeroseneLamp;
            var adapter = new HoseAdapter(lamp);
            adapter.Attach(hose);
        }
        else
        {
            fuelable.Attach(hose);
        }
        hose.Fuel(fuelable);
    }
}

```

The client can no longer rely on the IFuelable abstraction alone; it must know that one concrete subtype behaves differently. *That is a sign that the abstraction does not represent all of its implementations uniformly.* It also creates an **Open-Closed Principle** problem because each new exceptional subtype may require another branch in the client.

If using an **adapter** is valid business behavior, move that responsibility into KeroseneLamp so the client can continue to treat every IFuelable uniformly:

```

public class KeroseneLamp : IFuelable
{
    public int MaxFuel => 3;
    public int RemainingFuel => 3;
    public void Attach(FuelHose hose)
    {
        var adapter = new HoseAdapter(this);
        adapter.Attach(hose);
    }
}

```

FuelAll can then return to the straightforward form promised by its signature:

```

void FuelAll(List<IFuelable> fuelables, FuelHose hose)
{
    foreach (var fuelable in fuelables)
    {
        fuelable.Attach(hose);
        hose.Fuel(fuelable);
    }
}

```

```
    }  
}
```

When a type should not implement the broader interface

If attaching a hose to a kerosene lamp is not valid in the domain, *the more accurate design is to stop claiming that every fuel-containing object is hose-fuelable*. Split the capabilities:

```
public interface IFuelContainer  
{  
    int MaxFuel { get; }  
    int RemainingFuel { get; }  
}  
  
public interface IFuelable : IFuelContainer  
{  
    void Attach(FuelHose hose);  
}  
  
public class KeroseneLamp : IFuelContainer  
{  
    public int MaxFuel => 3;  
    public int RemainingFuel => 3;  
}
```

This design preserves the common concept of storing fuel while preventing invalid operations on types that cannot support them. It also foreshadows the **Interface Segregation Principle (ISP)**, where narrower interfaces allow each type to implement only the capabilities that make sense.

Avoiding weakened postconditions and strengthened preconditions

A subtype inherits the promises its base type made, and there are two main ways to break them.

Weakening a postcondition means promising callers less than the base contract promises. If a base type guarantees that `Find` always returns an object, a subtype that returns `null` when it finds nothing has broken that guarantee. Callers written against the base type do not check for `null`, because they were told they never have to.

Strengthening a precondition is the mirror image: requiring more from callers than the base contract requires. Under LSP, a subtype should not introduce such a stricter requirement. Here is what that looks like in practice.

Consider a `BankAccount` whose `Withdraw` method allows amounts below 10,000 without extra authorization. `ChildrenAccount` lowers that threshold to 1,000. A caller using a `BankAccount` reference may therefore expect a withdrawal of 5,000 to proceed without extra authorization, but that expectation no longer holds when the actual object is a `ChildrenAccount`.

```
public class BankAccount
{
    public virtual void Withdraw(int amount)
    {
        if (amount < 10000)
        {
            Console.WriteLine($"Withdrawing amount: {amount}");
            return;
        }
        Console.WriteLine($"Withdrawing {amount} requires extra authorization");
    }
}

public class ChildrenAccount : BankAccount
{
    public override void Withdraw(int amount)
    {
        if (amount < 1000)
        {
            Console.WriteLine($"Withdrawing amount: {amount}");
            return;
        }
        Console.WriteLine($"Withdrawing {amount} requires extra authorization");
    }
}
```

Here, inheritance is being used only to vary a configuration value. A simpler design is to keep one `BankAccount` type and supply the threshold as configuration. If the application should

expose only valid account configurations, a **static factory method** can centralize the permitted values:

```
public class BankAccount
{
    private readonly int _maxWithdrawalWithoutExtraAuthorization;

    private BankAccount(int maxWithdrawalWithoutExtraAuthorization)
    {
        _maxWithdrawalWithoutExtraAuthorization =
            maxWithdrawalWithoutExtraAuthorization;
    }

    public static BankAccount Create(BankAccountType type)
    {
        return type switch
        {
            BankAccountType.Adults => new BankAccount(10000),
            BankAccountType.Children => new BankAccount(1000),
            _ => throw new NotSupportedException("Unsupported account type.")
        };
    }

    public void Withdraw(int amount)
    {
        if (amount < _maxWithdrawalWithoutExtraAuthorization)
        {
            Console.WriteLine($"Withdrawing amount: {amount}");
            return;
        }
        Console.WriteLine(
            $"Withdrawing {amount} requires extra authorization");
    }
}

public enum BankAccountType
{
    Adults,
    Children
}
```

This removes an unnecessary inheritance relationship, and the substitution problem goes with it. Adding a new account type still means changing the factory, but *that is where the creation decision belongs*. Keeping that decision in one place means the rest of the code does not need subtype-specific knowledge. As discussed in *Chapter 6*, isolating the selection point is one practical way to contain unavoidable modification while applying the **Open-Closed Principle**.

Summary

This chapter discussed inheritance as a *promise about behavior* rather than a match of method signatures, which is what the Liskov Substitution Principle asks us to check. We went through the signs that the promise has been broken: a subtype that returns an invalid result where the base type returns a valid one, callers that have to check what type they actually received, inherited members that mean nothing for some subtypes, and overrides that reject inputs the base type accepted. We also saw two ways out. One is an interface describing what an object can actually do. The other is a single type that takes its differences as configuration. In *Chapter 8*, you will apply the Interface Segregation Principle to keep contracts focused on the capabilities that their clients actually need.

8

Applying the Interface Segregation Principle

Interfaces are meant to create useful boundaries, but an interface can become a liability when it collects unrelated operations. Implementers are then forced to provide meaningless members and clients depend on capabilities they never use.

The Interface Segregation Principle (ISP) keeps those contracts focused. This chapter examines interfaces as service boundaries, shows how polluted interfaces create artificial dependencies, and refactors broad contracts into smaller capabilities that better match the needs of their clients.

In this chapter, we will cover the following topics:

- Understanding the Interface Segregation Principle
- Thinking of interfaces as client-facing service boundaries
- Recognizing interfaces that force meaningless behavior
- Splitting broad interfaces by capability
- Relating ISP to LSP and SRP
- Avoiding interface pollution and unnecessary dependencies

By the end of this chapter, you will be able to recognize oversized or polluted interfaces, identify which clients actually need each capability, split contracts into focused abstractions, and explain how Interface Segregation complements both substitutability and single-responsibility design.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Understanding the Interface Segregation Principle

The **ISP** states that clients of an interface should not be forced to depend on methods they do not use. In practice, this means shaping interfaces around cohesive client needs rather than collecting unrelated operations in one broad contract. Smaller interfaces are often the result, but *size is not the goal by itself*: an interface should be as broad as necessary to represent one coherent capability or client role.

ISP is stated from the **client's** perspective, but both sides of the relationship matter. A client is code that receives or stores an object through the interface and calls its members; an **implementer** is the concrete type that fulfills the contract. Client dependencies reveal whether the interface exposes more than a caller needs, while awkward or meaningless implementations can reveal that the contract is too broad.

Thinking of an interface as a service boundary

A bank provides a useful analogy. A regular customer may interact with a clerk who can open accounts and exchange currencies. If the bank later adds a specialized valuables-storage service, every regular customer should not suddenly need to understand new security procedures. The bank can expose the specialist service through a separate role while keeping the ordinary customer-facing interface unchanged.

The software equivalent is to avoid adding unrelated methods to a broadly used interface. Instead, *create a second interface for the new capability and let only the appropriate concrete types implement it*.

When an interface forces meaningless behavior

Say we have an `IPersonalDataAccess` interface with a single `Read` method, and two classes implementing it. One reads the data from a text file. The other pulls it from a public census API. Both can do what the interface asks of them:

```
public interface IPersonalDataAccess
{
    string Read(int id);
}

public class TextFilePersonalDataAccess : IPersonalDataAccess
{
    public string Read(int id)
    {
        var peopleLines = File.ReadAllLines("someFile.txt");
        return peopleLines.First(
            line => line.StartsWith($"{id} "));
    }
}

public class CensusPersonalDataAccess : IPersonalDataAccess
{
    // Simplified for brevity.
    // Production code should reuse
    // an injected HttpClient and use async/await.
    public string Read(int id)
    {
        var url = $"https://someCensusApi.gov/residents/{id}";
        var response = new HttpClient().GetStringAsync(url).Result;
        return JsonSerializer.Deserialize<CensusRecord>(response).PersonalData;
    }
}
```

Now the text-file implementation needs to support writing new data. If `Write` is added to `IPersonalDataAccess`, every implementation is forced to provide it.

```
public interface IPersonalDataAccess
{
    string Read(int id);
```

```
void Write(int id, string firstName, string lastName);  
}
```

The census implementation has nowhere to write to because a public census source is read-only. A common but problematic workaround is to add a stub that throws `NotImplementedException`. The code then satisfies the interface syntactically, but the concrete type still cannot honor the behavior that the interface promises.

```
public class CensusPersonalDataAccess : IPersonalDataAccess  
{  
    public string Read(int id)  
    {  
        var url = $"https://someCensusApi.gov/residents/{id}";  
        var response = new HttpClient().GetStringAsync(url).Result;  
        return JsonSerializer.Deserialize<CensusRecord>(response).PersonalData;  
    }  
  
    public void Write(int id, string firstName, string lastName)  
    {  
        throw new NotImplementedException();  
    }  
}
```

Forcing `CensusPersonalDataAccess` to implement `Write` causes two problems.

The first is a violation of the **Liskov Substitution Principle**. Anyone holding an `IPersonalDataAccess` reference may call `Write` on it, because the interface promises the method is there. If the object behind it is `CensusPersonalDataAccess`, the call throws an exception.

The broader design problem is a violation of the **Interface Segregation Principle**, and in this example it creates the conditions for the **Liskov Substitution Principle** failure. The interface bundles reading and writing as if every data source supported both capabilities. Because that assumption is false, one implementation cannot honor the full contract.

Splitting the interface by capability

The issue is easily solved if we separate reading and writing into focused abstractions:

```
public interface IPersonalDataReader  
{  
    string Read(int id);  
}
```

```
}

public interface IPersonalDataWriter
{
    void Write(int id, string firstName, string lastName);
}
```

The text-file implementation can provide both capabilities, while the census implementation provides only reading:

```
public class TextFilePersonalDataAccess :
    IPersonalDataReader, IPersonalDataWriter
{
    private const string FilePath = "someFile.txt";
    public string Read(int id)
    {
        var peopleLines = File.ReadAllLines(FilePath);
        return peopleLines.First(
            line => line.StartsWith($"{id} "));
    }

    public void Write(int id, string firstName, string lastName)
    {
        var newLine = $"{id} {firstName} {lastName}";
        File.AppendAllLines(FilePath, new[] { newLine });
    }
}

public class CensusPersonalDataAccess : IPersonalDataReader
{
    public string Read(int id)
    {
        var url = $"https://someCensusApi.gov/residents/{id}";
        var response = new HttpClient().GetStringAsync(url).Result;
        return JsonSerializer.Deserialize<CensusRecord>(response).PersonalData;
    }
}
```

Interface segregation does not require a separate concrete class for every capability. `TextFilePersonalDataAccess` can implement both `IPersonalDataReader` and

IPersonalDataWriter because it supports both. The benefit is that each client can depend only on the focused interface it actually needs.

A client that only prints data can now depend on IPersonalDataReader instead of on a larger abstraction:

```
public class PersonalDataPrinter
{
    private readonly IPersonalDataReader _personalDataReader;

    public PersonalDataPrinter(IPersonalDataReader personalDataReader)
    {
        _personalDataReader = personalDataReader;
    }

    public void Print(int id)
    {
        var personalData = _personalDataReader.Read(id);
        Console.WriteLine($"Data of the person with id {id}:");
        Console.WriteLine(personalData);
    }
}
```

The result is more precise. Implementers are no longer forced to provide operations they cannot support, and clients no longer depend on methods they never use. There is also no way for a client to call a method that will fail on them, because a class that cannot write does not have a write method to call in the first place.

How ISP relates to LSP and SRP

ISP, **LSP**, and **SRP** often reinforce one another, but they ask different questions. When an oversized interface forces a type to provide a stub that throws, the broad contract is an ISP problem and the throwing implementation creates an LSP problem because callers cannot safely rely on the interface. An ISP problem can also contribute to an SRP problem if a concrete class is made responsible for unrelated capabilities simply to satisfy a broad contract, but that is not automatic. SRP asks whether responsibilities that change together belong together; ISP asks whether a client depends on operations it does not need. *Table 8.1* separates these questions.

Principle	Primary question	Typical warning sign
SRP	Does this class contain one coherent responsibility?	Unrelated reasons to change are combined in one class.
LSP	Can every subtype safely stand in for the base type?	A subtype throws, returns invalid results, or requires special-case client logic.
ISP	Does this client depend only on the operations it needs?	An interface contains methods that some clients or implementers do not need or cannot support.

Table 8.1: Distinguishing SRP, LSP, and ISP

A simple stack data structure provides a useful example of how SRP and ISP *do not always point in the same direction*. A stack class has Push and Pop, and both belong to one responsibility, so SRP has no objection to keeping them together. A client that only ever pushes, however, may benefit from a narrower interface that omits Pop, particularly when different client roles need different capabilities.

That does not mean every two-method interface should be split. Segregation has a cost: more abstractions to name, document, and maintain. A split is worthwhile when it clarifies client dependencies or prevents unsupported behavior, not merely to minimize the number of methods in an interface. *The point is that SRP and ISP ask different questions, and a design satisfying one can still fall short of the other.*

Avoiding interface pollution

Interface pollution, also called **interface bloating**, occurs when an interface accumulates so many operations that implementers are forced to provide methods they do not meaningfully support and clients must navigate a large API to reach the few operations they actually use. Table 8.2 provides practical questions for spotting this problem before the contract grows further.

Check	What to look for
Adding a new method to an interface	Will every current and plausible implementation be able to support it meaningfully?

Check	What to look for
Implementing an interface	Do any required methods feel unrelated to the type or require a stub implementation?
Using an interface	Does the client use only a small fraction of the exposed methods?
Changing an interface	Will the change force unrelated clients or implementations to rebuild, redeploy, or change?

Table 8.2: Practical checks for preventing interface pollution

Note

Design rule: Do not add stub implementations just to satisfy an interface. A method that does not fit a type is a sign that something is wrong, and there are two things worth questioning. Perhaps the interface covers more than it should and needs splitting. Or perhaps the interface is fine, and this type should not be implementing it at all.

Summary

This chapter showed how the Interface Segregation Principle keeps interfaces useful rather than burdensome. You examined client-facing service boundaries, recognized contracts that force meaningless behavior, and split broad interfaces by capability. You also saw that one concrete type can implement several focused interfaces while each client depends only on the capability it needs, and you distinguished **ISP** from **LSP** and **SRP**. Ultimately, *the goal is not the smallest possible interface, but a cohesive contract that matches the needs of its clients.*

Chapter 9 completes the SOLID sequence with the **Dependency Inversion Principle**, which uses abstractions to keep high-level behavior separate from the concrete details underneath it.

9

Applying the Dependency Inversion Principle

Almost no class works in isolation. Classes have dependencies, meaning other types they rely on. A class that presents customer data may depend on a service that fetches customers, another one that manages orders, and yet another that aggregates and formats the data. This is perfectly normal, and it is often the result of good practices: refactoring classes toward a single responsibility and toward being more focused.

The problem appears when the relationships between classes become rigid. If our class needs customer data and depends on one very specific class that fetches it, it may cause problems in the future. *A relationship this rigid means we cannot swap the implementation easily and safely, either when the requirements change or for the purposes of testing.* This is what the Dependency Inversion Principle is about, and it is the subject of this chapter.

This principle, abbreviated as DIP, states the following:

1. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
2. *Abstractions should not depend on details. Details should depend on abstractions.*

That is the official definition, but it may be easier to think of it in a simplified form: do not depend on concrete classes, depend on abstractions instead.

In this chapter, we will cover the following topics:

- Understanding the Dependency Inversion Principle
- Replacing concrete services with abstractions
- Keeping abstractions independent from implementation details
- Creating dependencies when runtime data is available

- Distinguishing Dependency Inversion from Dependency Injection
- Refactoring concrete data-provider dependencies
- Refactoring a design that violates several SOLID principles
- Reviewing how SOLID principles work together

By the end of this chapter, you will be able to replace concrete dependencies with abstractions, create runtime dependencies without coupling high-level logic to concrete classes, distinguish dependency inversion from dependency injection, and refactor a design in which several SOLID violations reinforce one another.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Understanding the Dependency Inversion Principle

The **Dependency Inversion Principle** is about the direction of dependencies. *High-level code should not be tightly coupled to low-level implementation details. Both should depend on abstractions.* Likewise, abstractions should not be shaped by the incidental details of one concrete implementation; the details should conform to the abstraction. Here, high-level code expresses the application's main behavior, while low-level code handles implementation details such as accessing databases, external APIs, or file systems.

Note

DIP in one sentence: Do not depend directly on concrete implementation classes when the behavior may vary; depend on an appropriate abstraction instead.

Replacing a concrete delivery service with an abstraction

To understand it better, let's consider a real-life example.

Imagine that you own a small online bookstore. You sell books to customers in your country, but you don't have your own delivery fleet, so you rely on another company to deliver the orders.

You find a company called `FastWheels` that fits your current needs. They offer decent prices and specialize in delivering small items within your country. In code, your `OnlineBookstore` class could depend directly on a `FastWheels` class and call its `DeliverSmallItem` method to ship books to customers.

The problem is that `FastWheels` is a concrete, low-level dependency. What happens if you expand your business and start shipping internationally? Or if you decide to sell furniture as well as books? `FastWheels` may no longer meet your needs.

The `OnlineBookstore` class is now tightly coupled to `FastWheels`. A change in the delivery service becomes a change that you need to make in the bookstore's code as well.

```
class OnlineBookstore
{
    private readonly FastWheels _fastWheelsService;

    public OnlineBookstore(FastWheels fastWheelsService)
    {
        _fastWheelsService = fastWheelsService;
    }

    public void Send(int bookId, int customerId)
    {
        var package = /* create package with book */;
        var address = /* get customer's address */;
        _fastWheelsService.DeliverSmallItem(package, address);
    }
}
```

For example, if we decide to change the `FastWheels` type to something else, the code calling the `DeliverSmallItem` method may stop compiling—because the new type's method may be called differently.

Adjusting the code in this small class may not be difficult, but imagine it is larger and uses `FastWheels` in dozens of places. Or even worse, imagine dozens of classes depending on `FastWheels`, spread across our entire codebase. Replacing references to `FastWheels` would

require changes across all of those classes, creating a lot of work and increasing the risk of bugs. We would also need to update dozens or hundreds of tests.

So how can we resolve this? To understand it, let's go back to the real-life analogy, which presents a similar problem: a bookstore depending on a very specific delivery company.

Instead of working directly with FastWheels, we could use a company called, let's say, AnyDelivery that arranges deliveries for us. We simply tell them what needs to be delivered and where, and they find a suitable delivery company. In other words, they worry about details, and we only need to tell them what we need.

AnyDelivery is an abstraction over the delivery service. We can use the same approach in code. Instead of depending on the concrete FastWheels type, we can depend on an IDelivery interface. The OnlineBookstore class does not need to know which specific type implements it. Initially, that type can be FastWheels.

When our delivery needs change, we can pass a different implementation of IDelivery to the OnlineBookstore constructor. We don't need to change the OnlineBookstore code or its tests. Only the code that creates the object needs to be adjusted, and that change has a much smaller impact. Let's see it in code:

```
public interface IDelivery
{
    void Deliver(Package package, Address address);
}

class OnlineBookstore
{
    private readonly IDelivery _deliveryService;

    public OnlineBookstore(IDelivery deliveryService)
    {
        _deliveryService = deliveryService;
    }

    public void Send(int bookId, int customerId)
    {
        var package = /* create package with book */;
        var address = /* get customer's address */;
        _deliveryService.Deliver(package, address);
    }
}
```

```
class FastWheels : IDelivery
{
    public void Deliver(Package package, Address address)
    {
        // FastWheels-specific implementation.
    }
}
```

OnlineBookstore now depends on what it needs rather than on who currently provides it. Another IDelivery implementation can be supplied without modifying OnlineBookstore.

Keeping abstractions independent from implementation details

The second part of the DIP definition says that abstractions should not depend on details, and details should depend on abstractions.

In this context, an abstraction is typically an interface or abstract class that defines what should be done without specifying how. A detail is a concrete implementation that defines how it is done.

For example, suppose IPeopleDataReader has two implementations: one reads data from an Excel file, and the other from cloud storage.

```
public interface IPeopleDataReader
{
    List<Person> GetAll();
}

public class CloudSourcedPeopleDataReader : IPeopleDataReader
{
    public List<Person> GetAll()
    {
        // Read from cloud storage.
    }
}

public class ExcelSourcedPeopleDataReader : IPeopleDataReader
{
    public List<Person> GetAll()
    {
```

```
        // Read from Excel.  
    }  
}
```

Suppose the Excel implementation needs a `fileName` parameter. In that case, we should not add it to the interface. Doing so would also force the cloud implementation to accept a parameter that has no meaning for it. This would be a case where a change in an implementation detail affects the abstraction, which would break DIP.

The `fileName` is a detail specific to the Excel implementation, so it should be handled there. For example, we could pass it to the `ExcelSourcedPeopleDataReader` constructor instead. *The interface should only change when the abstraction itself needs to change, not because one of its implementations has new requirements.*

Dependency Inversion and Dependency Injection are not the same

Dependency Inversion and **Dependency Injection** sound similar, but they describe different ideas. *DIP is a design principle about depending on abstractions. Dependency Injection is a construction pattern in which a class receives its dependencies from outside instead of creating them internally.* Let's compare two pieces of code:

```
class OnlineBookstore  
{  
    private readonly IDelivery _deliveryService;  
  
    public OnlineBookstore()  
    {  
        _deliveryService = new FastWheels();  
    }  
}
```

In this code, the `OnlineBookstore` class creates the dependency it needs itself. This means it must be aware of the implementation details of the `FastWheels` class, namely how its objects are created. This may seem like a minor problem in this example, but creating real-world objects is often much more complicated. It is not the responsibility of `OnlineBookstore` to know how to create a `FastWheels` object.

It's better to use Dependency Injection and inject this object into the class through its constructor.

```
class OnlineBookstore
{
    private readonly IDelivery _deliveryService;

    public OnlineBookstore(IDelivery deliveryService)
    {
        _deliveryService = deliveryService;
    }
}
```

This way, the class simply declares what it needs without concerning itself with the specific type it will receive or how it should be created.

If we use Dependency Injection, this class does not need to know about `FastWheels` or any other type implementing `IDelivery`. We can modify those implementations independently, as long as they continue to follow the interface.

Dependency Injection also allows us to switch between different implementations without modifying the `OnlineBookstore` class. An application may have several types implementing `IDelivery`, and we may want to choose between them based on configuration or user decisions. With Dependency Injection, we can provide the appropriate implementation at runtime.

It also makes the class easier to test. When testing `OnlineBookstore`, we can inject a mock of `IDelivery` instead of using a real delivery service. If the class creates `FastWheels` itself, we cannot easily replace it with a mock.

As a general guideline, inject dependencies instead of creating them inside the class that uses them.

To summarize: Dependency Inversion is a principle that says a type should depend on abstractions rather than concrete implementations. Dependency Injection is a technique where a class receives the dependencies it needs instead of creating them itself. *They work together well, but they are not the same thing.* Table 9.1 summarizes the distinction.

Concept	What it says	Example
Dependency Inversion Principle	High-level and low-level code should depend on abstractions; details should conform to those abstractions.	OnlineBookstore depends on IDelivery rather than FastWheels.
Dependency Injection	A class receives the objects it needs instead of constructing them itself.	An IDelivery object is passed to OnlineBookstore through its constructor.

Table 9.1: Dependency Inversion compared with Dependency Injection

Creating dependencies only when runtime data is available

Constructor injection is usually the clearest way to provide a dependency, but what if the dependency cannot be created until runtime?

Consider a `WeatherDataPrinter` class that asks the user to select a country and then retrieves the weather for that country. The `WeatherDataAccess` class requires the country in its constructor, so we cannot create it upfront and inject it into `WeatherDataPrinter`'s constructor. The required value simply is not available yet.

```
public class WeatherDataPrinter
{
    public void Print()
    {
        Console.WriteLine("Please select a country:");
        var country = Console.ReadLine();

        var weatherDataAccess = new WeatherDataAccess(country); //we need country
        to create this
        var weatherData = weatherDataAccess.GetCurrentWeather();

        Console.WriteLine($"The weather in {country} is: {weatherData}");
    }
}
```

In the code above, we create `WeatherDataAccess` directly inside the `Print` method, but this makes `WeatherDataPrinter` depend on the concrete `WeatherDataAccess` type, which violates the Dependency Inversion Principle.

We still need to create the dependency inside the method, because this is when we know the value of the `country` variable—so how can we avoid the concrete dependency? We can use a factory.

```
public interface IWeatherDataAccess
{
    string GetCurrentWeather();
}

public interface IWeatherDataAccessFactory
{
    IWeatherDataAccess Create(string country);
}

public class WeatherDataAccessFactory : IWeatherDataAccessFactory
{
    public IWeatherDataAccess Create(string country)
    {
        return new WeatherDataAccess(country);
    }
}

public class WeatherDataPrinter
{
    private readonly IWeatherDataAccessFactory _factory;

    public WeatherDataPrinter(IWeatherDataAccessFactory factory)
    {
        _factory = factory;
    }

    public void Print()
    {
        Console.WriteLine("Please select a country:");
        var country = Console.ReadLine();

        var weatherDataAccess = _factory.Create(country);
    }
}
```



```
        var weatherData = weatherDataAccess.GetCurrentWeather();
        Console.WriteLine($"The weather in {country} is: {weatherData}");
    }
}
```

Now `WeatherDataPrinter` depends on `IWeatherDataAccessFactory` rather than `WeatherDataAccess`. It can call the factory when the country is known and create the dependency only when it is needed.

The factory still depends on the concrete `WeatherDataAccess`, but this dependency is isolated in one place. The rest of the application can work with the abstractions.

This approach also makes the code easier to test. We can provide a mock `IWeatherDataAccessFactory` and control which `IWeatherDataAccess` the printer receives.

So, when a dependency cannot be created upfront because it requires runtime data or other conditions, a factory can help us keep the consuming class dependent on an abstraction while creating the concrete object only when needed.

Refactoring a concrete data-provider dependency

Let's see another example. `PopulationStatsPrinter` creates a `TextFilePopulationDataProvider` itself. That ties the two classes together tightly. There is no way to hand the printer a different data source, whether because the requirements change or because a test needs one. And the printer has to know how the provider is built, down to what its constructor expects:

```
public class PopulationStatsPrinter
{
    public void PrintAverageAge()
    {
        List<Person> people =
            new TextFilePopulationDataProvider("people.txt").GetAll();

        var averageAge = people.Average(person => person.Age);
        Console.WriteLine($"The average age is {averageAge}.");
    }
}
```

A better approach would be to introduce an abstraction and inject it:

```
public interface IPopulationDataProvider
{
    List<Person> GetAll();
}

public class TextFilePopulationDataProvider : IPopulationDataProvider
{
    private readonly string _filePath;

    public TextFilePopulationDataProvider(string filePath)
    {
        _filePath = filePath;
    }

    public List<Person> GetAll()
    {
        // Read and process the file.
    }
}

public class PopulationStatsPrinter
{
    private readonly IPopulationDataProvider _populationDataProvider;

    public PopulationStatsPrinter(
        IPopulationDataProvider populationDataProvider)
    {
        _populationDataProvider = populationDataProvider;
    }

    public void PrintAverageAge()
    {
        var people = _populationDataProvider.GetAll();
        var averageAge = people.Average(person => person.Age);
        Console.WriteLine($"The average age is {averageAge}.");
    }
}
```

Let's summarize what changed and why it made the code better.

Previously, `PopulationStatsPrinter` depended directly on `TextFilePopulationDataProvider` and created it inside the `PrintAverageAge` method. This made it difficult to switch to another data source or use a mock without modifying `PopulationStatsPrinter`.

Now, the class depends on the `IPopulationDataProvider` abstraction, and the concrete implementation is injected through the constructor. We can easily provide another implementation, such as one that reads data from a SQL database or a cloud service, without changing `PopulationStatsPrinter`. Unit testing is also easier because we can inject a mock of `IPopulationDataProvider`.

The classes are now loosely coupled. The interface connects them, making the code easier to maintain and modify.

There is one more dependency here that we haven't addressed: `PopulationStatsPrinter` uses the static `Console` class to communicate with the user. This also goes against the Dependency Inversion Principle because the class is tightly coupled to a specific way of communicating with the user. If we wanted to use a different way of communicating with the user, or mock it for testing, we would not be able to do so without modifying `PopulationStatsPrinter`.

We will not address this problem now. In *Chapter 10*, we will take a closer look at static methods and the problems they can cause, including how they can break the Dependency Inversion Principle.

Refactoring a design that violates several SOLID principles

Let's now bring together the SOLID principles in a single refactoring case study.

Understanding the code

As usual, we will start by understanding what the initial code does.

```
Employee employee = new Developer
{
    FirstName = "Anna",
    LastName = "Lima",
    Position = "Developer",
    HoursThisMonth = 160,
    BugsFixed = 6
};
```

```
var employeeReportCreator = new EmployeeReportCreator();
employeeReportCreator.CreateReportFor(employee);

Console.WriteLine("Done!");
Console.ReadKey();
```

Here we see a `Developer` object being created. It must derive from `Employee`, since it is assigned to a variable of that type. Then an `EmployeeReportCreator` object is created and produces a report.

If we ran this program, we would not see much on the screen, but a file would be generated: a very basic report containing the data of one employee. The report is a single line of text summarizing the employee: their name, their position, and the hours they worked. Developers get one extra piece of information, the number of bugs they fixed. Which version gets produced depends on the employee's position, checked by `CreateReportFor` before it picks a generator.

The whole thing is small and it works. Now let's go through the classes behind it method by method, because that is where the problems are.

Identifying the issues

Functionally, this is a simple application. The classes behind it are another matter. They are written in a convoluted way, and between them they manage to break every SOLID principle.

We will start with the `Employee` data structure. It carries some fundamental data about an employee. The `Developer` type is derived from it and extends it with a `BugsFixed` property.

```
public class Employee
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
    public string Position { get; init; }
    public int HoursThisMonth { get; init; }
}

public class Developer : Employee
{
    public int BugsFixed { get; init; }
}
```

Now, let's see the `EmployeeReportCreator` class. It implements the `IEmployeeReportCreator` interface.

```
public interface IEmployeeReportCreator
{
    void CreateReportFor(Employee employee);
    void SaveToFile(string path, string contents);
}

public class EmployeeReportCreator : IEmployeeReportCreator
{
    public void CreateReportFor(Employee employee)
    {
        ReportGenerator reportGenerator;

        if (employee.Position == "Developer")
        {
            reportGenerator = new DeveloperReportGenerator();
        }
        else
        {
            reportGenerator = new ReportGenerator();
        }

        var report = reportGenerator.Generate(employee);
        SaveToFile(BuildFileName(employee), report);
    }

    public void SaveToFile(string path, string contents)
    {
        File.WriteAllText(path, contents);
    }

    private string BuildFileName(Employee employee)
    {
        return $"{employee.FirstName}_{employee.LastName}.txt";
    }
}
```

Here, we see the first of the SOLID principles broken: this interface contains two completely unrelated methods, breaking the Interface Segregation Principle. The first creates a report for

an employee, and the second saves text to a file. This not only breaks the ISP but may lead to breaking the SRP. After all, any class implementing this interface will need to implement both of those methods, meaning it will have two responsibilities instead of one. And this is just what happens in the `EmployeeReportCreator` class. It deals with creating the report and exposes a public method for saving strings to a file. Technically, it does not write to a file itself but uses the built-in `File` class. Still, this is quite a low-level operation and should certainly not happen in this class.

Let's also understand the `CreateReportFor` method. It takes an employee and, depending on their position, creates either a basic `ReportGenerator` or a specialized `DeveloperReportGenerator`. That single decision breaks two SOLID principles.

The first is the Open-Closed Principle. Adding a new employee type would force us to come back here and add another `else` branch.

The second is the Dependency Inversion Principle. Those report generators are dependencies of this class, and `EmployeeReportCreator` names them directly. It should depend on an abstraction instead, and ideally receive it through dependency injection.

```
public class ReportGenerator
{
    public virtual string Generate(Employee employee)
    {
        return $"{employee.FirstName} {employee.LastName}, " +
               $"working as {employee.Position}, " +
               $"worked {employee.HoursThisMonth} hours this month.";
    }
}

public class DeveloperReportGenerator : ReportGenerator
{
    public override string Generate(Employee employee)
    {
        if (employee.Position != "Developer")
        {
            throw new InvalidOperationException(
                "This report can only be generated for developers");
        }

        return
            $"{employee.FirstName} {employee.LastName}, " +
```

```

        $" working as {employee.Position}," +
        $" worked {employee.HoursThisMonth} hours this month" +
        $" and fixed {(employee as Developer).BugsFixed} bugs.";
    }
}

```

Now the `ReportGenerator` and `DeveloperReportGenerator` classes. The first one is simple. It builds a string with the employee's data. Notice that its method is virtual, and that it is overridden in `DeveloperReportGenerator`.

The override starts by checking whether the employee's position is "Developer", and throws an exception if it is not. That breaks the Liskov Substitution Principle because the override rejects an `Employee` input that the base method accepts. It may also throw an exception where the base implementation would return a result.

The result is that any code working with `ReportGenerator` will break the moment it receives a `DeveloperReportGenerator` and an employee who is not a developer.

When the argument is a developer, the method works, but it duplicates what the base class already does and adds the number of fixed bugs on top. *Table 9.2* summarizes the design issues identified before the refactoring begins.

Problem	Principle involved	Why
<code>IEmployeeReportCreator</code> contains report creation and general file-writing operations	ISP / SRP	Clients and implementers are forced to depend on unrelated responsibilities.
<code>EmployeeReportCreator</code> selects generator types with an if/else chain	OCP	Adding a new employee/report type requires modifying existing selection logic.
<code>EmployeeReportCreator</code> constructs concrete generators	DIP	High-level report orchestration depends on concrete implementation classes.
<code>DeveloperReportGenerator</code> rejects non-developer <code>Employee</code> objects	LSP	A subtype cannot safely replace <code>ReportGenerator</code> for every valid <code>Employee</code> input.

Problem	Principle involved	Why
EmployeeReportCreator creates its own dependencies File.WriteAllText is called directly	DIP / testability	The concrete dependency cannot easily be replaced in tests or alternative environments.

Table 9.2: SOLID problems in the employee-report design

Improving the code

We can address these problems in two stages: first isolate file access, then simplify report generation.

Isolating file access behind an abstraction

Let's start with the low-hanging fruit, the simplest and fastest thing to fix. Here, it is the ISP violation in the IEmployeeReportCreator interface. It holds two methods that have nothing to do with each other.

Consider how odd it would be if someone needed to save text to a file and reached for an EmployeeReportCreator object to do it. That is what this code suggests. The class implements this interface, so it is obliged to provide a method for saving text to a file. That's why it's better to have two separate interfaces:

```
public interface IFile
{
    void WriteAllText(string path, string contents);
}

public interface IEmployeeReportCreator
{
    void CreateReportFor(Employee employee);
}
```


Let's also introduce a wrapper that will hide the details of the low-level `File.WriteAllText` method. A thin wrapper is appropriate here because the .NET File API is static and therefore difficult to substitute in tests:

```
public class FileWrapper : IFile
{
    public void WriteAllText(string path, string contents)
    {
        File.WriteAllText(path, contents);
    }
}
```

The wrapper still uses the static `File` class, but the unavoidable concrete dependency is isolated in one class with no business logic. `EmployeeReportCreator` can depend on `IFile` and receive a mock during unit tests:

```
public class EmployeeReportCreator : IEmployeeReportCreator
{
    private readonly IFile _file;

    public EmployeeReportCreator(IFile file)
    {
        _file = file;
    }

    public void CreateReportFor(Employee employee)
    {
        ReportGenerator reportGenerator;

        if (employee.Position == "Developer")
        {
            reportGenerator = new DeveloperReportGenerator();
        }
        else
        {
            reportGenerator = new ReportGenerator();
        }

        var report = reportGenerator.Generate(employee);
        SaveToFile(BuildFileName(employee), report);
    }
}
```

```
}

private void SaveToFile(string path, string contents)
{
    _file.WriteAllText(path, contents);
}

private string BuildFileName(Employee employee)
{
    return $"{employee.FirstName}_{employee.LastName}.txt";
}
}
```

This change improves ISP compliance by keeping `IEmployeeReportCreator` focused, and DIP compliance by replacing a static concrete dependency with an injectable abstraction.

We must also remember to adjust the code that creates `EmployeeReportCreator`.

```
var employeeReportCreator = new EmployeeReportCreator(new FileWrapper());
```

Moving report behavior to the objects that own the data

The next design problem is the pair of generator classes. The `ReportGenerator` is not bad on its own. It builds a report for an employee. The trouble starts with the `virtual` keyword. Once the method is virtual, derived classes can override its behavior.

That is what happens in `DeveloperReportGenerator`. Overriding the method forces it to keep the same signature, including the `Employee` parameter, even though this class only deals with developers. Hence the exception thrown when the argument is not one.

This is a typical misuse of inheritance and virtual methods. Look at the shape of the design. We have two employee types, one deriving from the other, and two matching generator types, also one deriving from the other. Notice as well how much each generator uses data from `Employee`. *That can be a sign that a method does not belong to the type it lives in, and should move to the type whose data it uses.*

So the `Generate` methods belong in `Employee` and `Developer`, and the generators can go.

```
public class Employee
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
```

```
public string Position { get; init; }
public int HoursThisMonth { get; init; }

public virtual string GenerateReport()
{
    return $"{FirstName} {LastName}, " +
           $"working as {Position}, " +
           $"worked {HoursThisMonth} hours this month.";
}

}

public class Developer : Employee
{
    public int BugsFixed { get; init; }

    public override string GenerateReport()
    {
        return base.GenerateReport().TrimEnd('.') +
               $" and fixed {BugsFixed} bugs.";
    }
}
```

EmployeeReportCreator no longer needs to inspect Position or construct generator classes. It simply asks the employee object to produce the appropriate report:

```
public class EmployeeReportCreator : IEmployeeReportCreator
{
    private readonly IFile _file;

    public EmployeeReportCreator(IFile file)
    {
        _file = file;
    }

    public void CreateReportFor(Employee employee)
    {
        var report = employee.GenerateReport();
        SaveToFile(BuildFileName(employee), report);
    }

    private void SaveToFile(string path, string contents)
```

```
{
    _file.WriteAllText(path, contents);
}

private string BuildFileName(Employee employee)
{
    return $"{employee.FirstName}_{employee.LastName}.txt";
}
}
```

That is the refactoring done. Let's go through the SOLID principles one by one and see what changed.

Single Responsibility Principle. Each class now has one responsibility. They are all small and focused.

Open-Closed Principle. There are no more if-else statements checking the employee type, so *adding a new kind of employee no longer means editing existing code*. It means adding a new type derived from Employee.

Liskov Substitution Principle. The inheritance between the report generators is gone. Generating the report string is now the job of Employee and its subtypes, and those subtypes can be used safely anywhere the base type is expected.

Interface Segregation Principle. The method for writing to a text file used to sit in an interface it had nothing to do with. It now has its own interface, and anyone who needs that functionality uses IFile.

Dependency Inversion Principle. The generator classes are gone, so EmployeeReportCreator no longer depends on them. It still needs to write to a file, but it does so through an interface rather than depending on File directly.

The result is easier to read. Nothing surprising happens in this code, and every class is simple enough to understand on its own.

Summary

This chapter completed the SOLID sequence with the **Dependency Inversion Principle**. We replaced concrete services with abstractions, saw how to create dependencies at runtime without tying high-level logic to any particular implementation, and drew the line between dependency inversion and dependency injection. The employee report case study at the end showed how several SOLID problems tend to feed one another, and how the right abstractions pull them apart.

Chapter 10 builds on this to look at static methods: when they are the right choice, and when a static member is hiding a dependency that should have been visible.

10

Static methods and dependencies

Static methods are tempting. A **static method** belongs to the class itself rather than to any of its instances, so there is nothing to create and nothing to pass around. Reach for the class name and the method is there. Static fields and properties work the same way, except that they hold state shared by the whole application.

Sometimes using them is the right choice. Sometimes it costs us more than it saves. This chapter is about telling the two cases apart. A *static call is a dependency that never shows up in a constructor or a signature*. Nothing declares it, so nothing can replace it. When the behavior behind that call needs to change, or needs to behave differently in a test, we find that we have no way in.

In this chapter, we will cover the following topics:

- Deciding when a method should be static
- Distinguishing private helpers from public dependencies
- Understanding why public static methods can weaken substitution and testability
- Wrapping static framework APIs when behavior must be controlled
- Recognizing hidden dependencies and global state
- Making the dependency graph explicit through constructors

By the end of this chapter, you will be able to decide when static methods are appropriate, recognize public static members that hide replaceable dependencies, wrap static framework APIs when tests require control, identify obscured dependency graphs and global state, and refactor classes so that required collaborators are explicit and substitutable.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Static methods at a glance

A static method belongs to the type rather than to any particular object. Since there is no instance behind the call, a static method cannot reach the instance fields or properties of its class. Whatever it needs has to come in through its parameters, or from static fields, which belong to the type as well.

An operation such as `Min` depends only on the values passed to it. It uses no state from the class it lives in, so nothing stops it from being static:

```
public static class Math
{
    public static int Min(int a, int b)
    {
        return a < b ? a : b;
    }
}
```

The `Add` method of a list makes a good contrast. It is not static, and it could not be, because it works on the instance's own items and count:

```
class List<T>
{
    private T[] _items = [];

    //Resizing omitted for brevity
}
```

```
public int Count { get; private set; }

public void Add(T item)
{
    _items[Count] = item;
    ++Count;
}
}
```

Deciding when a method should be static

Whether a method should be static depends not only on whether it uses instance state, but also on whether callers may need to replace or control its behavior.

Private methods

We know that a method using none of the class state can be made static. The question is whether it should be.

For private methods, the answer is often yes. Making such a method static states plainly that it does not touch instance data, so anyone reading it knows from the signature alone that the object's state is not part of the picture. Everything the method works with is either in its body or in its parameters, and there is no need to go looking at what fields the class happens to have.

Why public static methods need more caution

Making public methods static may be a bad idea. Let's understand why with the following piece of code.

```
public class DataAccess
{
    public List<int> GetData()
    {
        // connects to a DB and reads the data
    }
}

public class DataProcessor
{
    private readonly DataAccess _dataAccess;

    public DataProcessor(DataAccess dataAccess)
    {
```



```
        _dataAccess = dataAccess;
    }

    public int CalculateSum()
    {
        return _dataAccess.GetData().Sum();
    }
}
```

Here is a `DataAccess` class with a public `GetData` method. It opens a connection to an SQL database and reads the data from there. The method uses no instance data, so technically it could be made static.

`DataProcessor` depends on it. It receives a `DataAccess` object through the constructor and stores it in a field.

As we already know, this goes against the **Dependency Inversion Principle**. `DataProcessor` should depend on an abstraction rather than a concrete type. As it stands, switching to a different way of reading the data, from a file or an API, would mean changing `DataProcessor` itself, its constructor at the very least. Testing has the same problem. A unit test should not be opening a real database connection, and right now there is no way to substitute a mock.

Depending on an abstraction fixes both. In C#, that means an interface.

```
public interface IDataAccess
{
    List<int> GetData();
}

public class DataAccess : IDataAccess
{
    public List<int> GetData()
    {
        // connects to a DB and reads the data
    }
}

public class DataProcessor
{
    private readonly IDataAccess _dataAccess;

    public DataProcessor(IDataAccess dataAccess)
```

```
{
    _dataAccess = dataAccess;
}

public int CalculateSum()
{
    return _dataAccess.GetData().Sum();
}
}
```

Now `DataAccess` implements the `IDataAccess` interface, and `DataProcessor` is given an object of that interface rather than a concrete class. It has no idea what it actually received. It might be the `DataAccess` object talking to an SQL database, or something reading from a file, or a mock used in a test. Any of them can be swapped in without touching `DataProcessor`, and the two classes now communicate only through the interface.

This is a good design. The two classes are no longer coupled to each other, *the implementation can be replaced whenever the requirements call for it*, and testing `DataProcessor` is straightforward, since a mock goes in exactly where the real thing would.

Now let's go back to static methods. We said earlier that `GetData` uses no instance state, so it could be made static. Say we decide to do exactly that.

```
public class DataAccess
{
    public static List<int> GetData()
    {
        // connects to a DB and reads the data
    }
}
```

The call site has to change now. A static method is called on the type rather than on an object, so instead of `_dataAccess.GetData()` we write `DataAccess.GetData()`. The interface reference is gone, and with it the object that used to be passed in.

```
public class DataProcessor
{
    public int CalculateSum()
    {
        return DataAccess.GetData().Sum();
    }
}
```

```
}  
}
```

With that, everything the Dependency Inversion Principle gave us is gone. There is no way to switch the implementation to something else, and no way to substitute a mock in a test. The code is bound to this one class.

There is more. `DataContext` can no longer implement `IDataAccess` at all. Methods implementing an interface are implicitly virtual, and a virtual method cannot be static. The runtime uses the type of an object to decide which implementation of a virtual method to invoke, and a static method is not invoked on an object in the first place. It is invoked on the type. So a method can be virtual or static, never both. Which means `DataContext` can no longer be used polymorphically.

That is the reason to avoid making public methods static. A static method always behaves the same way, and there is no object to call it on, so there is no way to substitute different behavior. The code gets rigid, and testing gets hard, because static methods are difficult to replace through ordinary interface-based mocking.

So a public static method that represents replaceable behavior deserves caution. If there is any chance the implementation will need to be swapped or mocked, making it static takes that option away.

Virtual methods

```
//may be DataContext  
//or mock of IDataAccess  
//or anything else implementing DataContext  
IDataAccess dataAccess = ???;
```

```
var numbers = dataAccess.GetData();
```

Will behave **differently** depending on the **actual type** of this object.

Static methods

```
var numbers = DataContext.GetData();
```

Always **the same** behavior. No object whose type can determine different behavior.

Figure 10.1: Comparison between runtime dispatch for virtual and static method calls

Let's see the few cases where a public static method is the right call.

When public static methods make sense

A public method can safely be made static when we always want the same implementation and will never replace it with another one. That includes replacing it with a mock. *If there is any chance we will want to mock the method, it should not be static.*

Simple as that sounds, it rules out most methods. Anything implementing a business requirement may need a different implementation eventually. Methods that provide data are the clearest case: reading from a database can become reading from a file or from cloud storage, and a console application can become one with a graphical interface. On top of that, methods that fetch data have to be mocked in unit tests. We do not want tests hitting a real data source or a real user interface, because then we no longer control what the test does.

So which methods will never need different behavior and never need mocking? The `Math` class is a good example. Basic mathematical operations do not have alternative implementations, because mathematics does not change, and there is no reason to mock them in a test. Running the real implementation is exactly what we want.

The same goes for simple text processing. A method that strips whitespace from a string takes a value, transforms it, and returns the result. There is no second way of doing it that we might want to swap in later, and a test has no reason to fake it, since running the real thing is fast and gives the same answer every time.

`Math` is a built-in .NET type that is static and should be. Not every static member of the standard library is such a good decision, though. Let's see how to deal with that.

Wrapping static framework APIs

Sometimes a static method we need comes from the standard library or an external package, and we have no say in the matter. `DateTime.Now` in C#, which returns the current date and time, is the classic example. It is static, so it cannot be replaced through ordinary interface-based mocking, and that becomes our problem as soon as we write a method that depends on it. Consider a method that describes the current day:

```
string GetCurrentDayDescription()  
{  
    DateTime today = DateTime.Now;  
    return $"Today is {today.DayOfWeek}";  
}
```

The output of this method changes with the clock. Run it on Monday and it says one thing, run it on Tuesday and it says another. *A unit test has to produce the same result every time it runs*, so a test for this method would need to fix what the current date is. That means replacing

`DateTime.Now` with something we control, and we cannot do it, because a static property cannot be replaced through ordinary interface-based mocking.

The usual way out is a wrapper. We define our own `IDateTime` interface and a class implementing it, whose job is to provide the same value the static property does, only in a non-static way that can be mocked. Here are the interface and the wrapper:

```
interface IDateTime
{
    DateTime Now { get; }
}

class DateTimeWrapper : IDateTime
{
    public DateTime Now
    {
        get
        {
            return DateTime.Now;
        }
    }
}
```

With the wrapper in place, a class that needs the current time takes an `IDateTime` in its constructor instead of reaching for `DateTime.Now` directly. In production it gets the real wrapper and behaves exactly as before. In a test it gets a mock returning a fixed date, so the result is the same on every run.

The dependency on the clock is also visible now. It sits in the constructor, where anyone reading the class can see it, rather than hiding inside a method body.

Note

`TimeProvider`, introduced in .NET 8, offers a built-in, testable alternative to `DateTime.Now` for newer projects. That said, `DateTime.Now` is likely to remain common in existing codebases for a long time. It is also only one example of a static dependency that needs wrapping. `File` and `Console` raise the same problem, and most static dependencies have no framework-provided alternative, so knowing how to wrap them yourself is a practical skill.

Keeping the dependency graph visible

We saw how a static method can lead to breaking the Dependency Inversion Principle. That is part of a larger problem, sometimes called obscuring the **dependency graph**. A dependency graph shows how the classes in a system depend on one another. It sounds abstract, but it is a simple idea and a useful one, and *it matters more than it might seem that we can see the full graph for a class without missing any part of it*.

Imagine joining a project and trying to find your way around the code. There is a `BankAccount` class that looks complicated, and it is not obvious how it works. To see it in action, you write a simple unit test.

```
public class BankAccountTests
{
    [Test]
    public void TransferTo_ShallRemoveAmountFromOneAccount_AndAddItToAnother()
    {
        var bankAccount1 = new BankAccount();
        bankAccount1.AddFunds(1000);

        var bankAccount2 = new BankAccount();
        bankAccount1.TransferTo(bankAccount2, 200);

        Assert.That(bankAccount1.Balance, Is.EqualTo(800));
        Assert.That(bankAccount2.Balance, Is.EqualTo(200));
    }
}
```

You expect this test to pass, as nothing seems suspicious in the code. However, when you run the test, it fails with a complicated error message. After some investigation and asking a senior team member, you learn that before calling the `TransferTo` method, you must initialize a `TransferServiceManager` class.

You update your test to include this initialization:

```
public class BankAccountTests
{
    [Test]
    public void TransferTo_ShallRemoveAmountFromOneAccount_AndAddItToAnother()
    {
        TransferServiceManager.Initialize();
    }
}
```

```
var bankAccount1 = new BankAccount();
bankAccount1.AddFunds(1000);

var bankAccount2 = new BankAccount();
bankAccount1.TransferTo(bankAccount2, 200);

Assert.That(bankAccount1.Balance, Is.EqualTo(800));
Assert.That(bankAccount2.Balance, Is.EqualTo(200));
}
}
```

But running the test again results in another exception. After another round of questions, you are told that you must also initialize `InternalServiceQueue`. Only after adding this line does the test finally pass:

```
public class BankAccountTests
{
    [Test]
    public void TransferTo_ShallRemoveAmountFromOneAccount_AndAddItToAnother()
    {
        InternalServiceQueue.Initialize();
        TransferServiceManager.Initialize();

        var bankAccount1 = new BankAccount();
        bankAccount1.AddFunds(1000);

        var bankAccount2 = new BankAccount();
        bankAccount1.TransferTo(bankAccount2, 200);

        Assert.That(bankAccount1.Balance, Is.EqualTo(800));
        Assert.That(bankAccount2.Balance, Is.EqualTo(200));
    }
}
```

It takes over twenty seconds to run, but a senior colleague tells you that is normal, because this is how long those services take to initialize.

Before we get to how bad this design is, one thing is worth saying. The example is simplified, but the problem it shows is not. Code that behaves like this turns up in real projects, and most developers run into it sooner or later.

So what is wrong here? *The class lies, or at least it does not tell the whole truth.* `BankAccount` needs those services to work, and it says so nowhere. It presents itself as something we can create and use right away, which is exactly what the first version of the test assumed, and that turned out to be false.

Those services are dependencies of this class, and they are not declared in the constructor. That leaves us no way of knowing they exist without reading the implementation. This is what obscuring the dependency graph means. The graph for `BankAccount` includes those two types, and nothing on the surface of the class admits it.

Why hidden dependencies create fragile code

Objects should be ready to use immediately after you create them, without requiring any secret initialization steps or special method calls. If you find yourself needing to perform some magic setup or call a particular method to make an object work, your design is flawed. *The only setup required before using an object should happen in its constructor.* If a type needs certain things to function, it should request them in the constructor and nowhere else.

When dependencies are hidden, it becomes easy to introduce serious bugs. For example, if you refactor your code and accidentally swap the order of two initialization lines, your tests may start failing. After spending time diagnosing the issue, you might discover that one dependency must be initialized before the other. This means the actual dependency graph is more complex than it appears in the code, and this lack of clarity can lead to mistakes.

But what does it have to do with static members? Well, static members are globally reachable through their type name. That reachability makes it easy for code anywhere in the application to acquire a dependency without declaring it in a constructor or method signature.

Look at the class below. It doesn't take anything in its constructor, so from the outside it looks like it depends on nothing at all. Then we open a method body and find `InternalServiceQueue` being called. The class depends on it, and nothing about its public surface says so:

```
class SomeClass
{
    public void SomeMethod()
    {
        InternalServiceQueue.Initialize();
    }
}
```

If a class exposes a public static method, any code in your project can call this method, creating a hidden dependency. If methods are not static, you need an instance of the type to call them.

This means you must provide the instance to the dependent class, typically through a constructor or method parameter, making the dependency explicit and ensuring the code will not compile unless the dependency is supplied.

```
class SomeClass
{
    private readonly IInternalServiceQueue _internalServiceQueue;

    public SomeClass(IInternalServiceQueue internalServiceQueue)
    {
        _internalServiceQueue = internalServiceQueue;
    }

    public void SomeMethod()
    {
        _internalServiceQueue.Initialize();
    }
}
```

A class that declares no dependencies may still have one. If some other type exposes an instance in a public static field, any code can reach it, and the dependency exists without ever appearing in a constructor. That is what makes public static fields dangerous. They create **global state**, meaning data reachable from any point in the application, and anything can come to depend on it without saying so. It is the same problem we saw with static methods, only now it applies to data rather than behavior.

The following code demonstrates how a hidden dependency can be introduced through a public static field:

```
class BankAccount
{
    public int Balance { get; private set; }

    public void AddFunds(int amount)
    {
        SomeOtherType.TransferServiceManager.SomeNonStaticMethod();
    }
}

class SomeOtherType
{

```

```
public static ITransferServiceManager TransferServiceManager;
}
```

If you use a public static field like `TransferServiceManager`, any code can access it at any time, creating a hidden dependency that is not declared in the constructor or interface of your class.

The point to remember is that public static methods and public static fields can both hide dependencies, and in many cases *the code is clearer when those dependencies are explicit*.

Making dependencies explicit

To solve the problem of static methods obscuring the dependency graph, declare dependencies explicitly. If `BankAccount` needs `TransferServiceManager`, it should require it as a constructor parameter. If `TransferServiceManager` needs `InternalServiceQueue`, it should require it too. If you forget to provide a dependency, you will get a compilation error, not a runtime failure or mysterious test failure. You won't need to rely on secret knowledge or special lines of code.

Initialization methods that must be called before an object becomes usable are another form of hidden prerequisite. *Prefer constructing objects in a valid, ready-to-use state and require essential collaborators through constructors.*

Constructor injection also creates a natural testing seam. Instead of initializing real infrastructure, a unit test can pass controlled test doubles and exercise `BankAccount` in isolation. After removing the static initialization methods and making all dependencies explicit, the code becomes easier to understand and maintain. The following code shows the improved approach:

```
class BankAccount
{
    private readonly ITransferServiceManager _transferServiceManager;

    public BankAccount(ITransferServiceManager transferServiceManager)
    {
        _transferServiceManager = transferServiceManager;
    }

    public int Balance { get; private set; }

    public void AddFunds(int amount)
    {
        // imagine a lot of complicated code
    }
}
```

```
public void TransferTo(BankAccount other, int amount)
{
    // imagine a lot of complicated code
}

class TransferServiceManager : ITransferServiceManager
{
    private readonly IInternalServiceQueue _internalServiceQueue;

    public TransferServiceManager(IInternalServiceQueue internalServiceQueue)
    {
        _internalServiceQueue = internalServiceQueue;
    }
}
```

Now `BankAccount` states that it needs an `ITransferServiceManager`, which in turn states that it needs an `IInternalServiceQueue`. Nothing is hidden.

Passing dependencies through the constructor brings another benefit: we can hand the class whatever we like, including a mock. Let's see how the test could look now:

```
using NUnit.Framework;
using Moq;

[TestFixture]
public class BankAccountTests
{
    [Test]
    public void TransferTo_ShallRemoveAmountFromOneAccount_AndAddItToAnother()
    {
        var bankAccount1 = new BankAccount(
            new Mock<ITransferServiceManager>().Object);
        bankAccount1.AddFunds(1000);

        var bankAccount2 = new BankAccount(
            new Mock<ITransferServiceManager>().Object);
        bankAccount1.TransferTo(bankAccount2, 200);

        Assert.That(bankAccount1.Balance, Is.EqualTo(800));
    }
}
```

```
        Assert.That(bankAccount2.Balance, Is.EqualTo(200));  
    }  
}
```

Now the test uses mocks instead of real infrastructure. Remember that earlier, this test took twenty seconds to pass? That was the real services initializing. With a mock in their place, the test finishes immediately.

Unit tests should generally avoid real external dependencies. A unit test tests one class in isolation, and once the real dependencies are in play, they are being tested too. At that point it is no longer a unit test. It also makes the tests slow, as with a real database connection. And it can cost money. It is not unheard of for a test suite to connect to real cloud services instead of mocking them, and every run adds to the bill.

With the current code, there are no `Initialize` methods to remember. No test performs real initialization, so nothing takes twenty seconds. A `BankAccount` object works as soon as it is created, and it cannot be created without its dependencies, because the code would not compile.

This code no longer lies. It tells us the truth, the whole truth, and nothing but the truth.

Summary

This chapter distinguished convenient static operations from static dependencies that make design and testing harder. You saw why private helpers and stable operations can be reasonable static methods, why public replaceable behavior deserves more caution, and how wrappers can provide control over framework APIs such as `DateTime.Now`. The `BankAccount` example demonstrated how global initialization obscures the dependency graph and how constructor injection makes required collaborators visible and mockable.

In *Chapter 11*, you will shift from dependency visibility to class size and learn how to identify natural boundaries inside classes that have accumulated too many responsibilities.

11

Designing smaller classes

A class can begin as a convenient home for a small amount of related behavior and slowly become the place where every new requirement is added. Over time it accumulates unrelated data, dependencies, business rules, low-level utilities, and special cases, until even a small change requires understanding code that has nothing to do with the task.

That is why *smaller classes are worth aiming for*. A class that fits in one's head can be read, changed, and tested without dragging everything around it into the picture.

But not at any cost. *A large class is a warning sign, not an automatic failure*, and splitting one apart to get under some line limit does more harm than leaving it alone. The goal is to find the cohesive responsibilities inside it and give each of them a home of its own. This chapter shows how to recognize those boundaries and refactor toward smaller classes without inventing abstractions that do not need to exist.

In this chapter, we will cover the following topics:

- Reviewing class-design fundamentals
- Understanding the problems caused by oversized classes
- Recognizing when a large class is nevertheless cohesive
- Avoiding superficial techniques for making code look shorter
- Finding class boundaries through data usage and dependencies
- Grouping behavior that changes together
- Separating high-level and low-level responsibilities
- Avoiding unnecessary class proliferation
- Refactoring an oversized class into smaller, focused classes

By the end of this chapter, you will be able to recognize when a class has accumulated too many responsibilities, distinguish justified size from god-class design, use naming, data usage,

change patterns, and abstraction levels to identify natural class boundaries, and refactor a larger class into focused, testable, reusable collaborators.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Reviewing class-design fundamentals

Before going further, a quick reminder of the terms we will be using.

In object-oriented programming, an object combines data with the behavior that operates on that data. A `Person` object may hold a name and a date of birth, and know how to calculate its own age. An `Address` object may hold a street, a number, and a city, and format them into a single line for a label. A `Person` can in turn hold a collection of `Address` objects. Building an application out of smaller concepts like these is what lets us avoid writing it as one large block of code.

A **class** is the definition from which those objects are created. It describes what data and what **methods** the **instances** will have. `Person` is the class, and the objects created from it are its instances, so John and Jane are two `Person` instances with different names and different dates of birth, sharing the same structure and the same behavior.

Some classes represent things from the real world, such as people, addresses, or orders. Others represent technical concepts, such as a database connector, a repository, a text reader, or a randomizer.

Public and private members

Classes also control which parts of themselves are visible to other code. **Public members** are the surface that other objects can use. **Private members** are implementation details, reachable only from inside the class. Keeping data and helper methods private stops unrelated code from touching details it has no business knowing about.

Fields and properties

Fields hold the state of an object. A Person object storing a name and a date of birth keeps them in fields.

Some languages, C# among them, also have **properties**. A property is a method for reading or setting a value, dressed up to look like a field. From the outside it is read and assigned the same way a field would be, but each access runs code the class controls. That lets the class decide what happens when someone reads or changes the value, or expose a value that can be read but not written. In languages without properties, the same thing is done with plain getter and setter methods.

Inheritance

Inheritance lets one class build on another. The **derived class** takes on the fields and methods of the **base class**, and it can override some of them to behave differently. This is how two types can share most of their behavior and differ only where it matters.

Interfaces work differently. An interface declares a set of operations without providing any implementation, and a class implementing it must supply the code. There is nothing to inherit, only a contract to satisfy. A class can implement as many interfaces as it needs, while in most languages it can inherit from only one base class.

Understanding the dangers of large classes. God class antipattern

Of all the properties that affect the quality of a class, size is arguably the one that matters most. Even when everything else is a mess, short code can usually be figured out. So let's start with why small classes beat large ones, and with the **antipattern** known as the god class.

A god class, sometimes called a god object, is a class that does too many things. These classes can be enormous. They carry multiple responsibilities and hold dozens or hundreds of fields, properties, and methods, and they seem to have a hand in everything the application does. A good indicator is when every functionality change, whatever it concerns, ends up touching this one class.

Classes like this reach thousands of lines, sometimes more than ten thousand. Nobody understands what such a class does, and nobody can. Testing it thoroughly is close to impossible, and changing it without breaking something is just as hard. In practice, people modify small parts of it and hope for the best.

The god class is one of the most dangerous antipatterns there is, and the problems it causes are the same as those of any large class.

The most obvious one is that reading the whole thing takes a long time, so nobody does. At best, programmers end up with a vague sense of a few fragments.

That is not to say we should read every line of a codebase to understand an application. Driving a car does not require understanding the engine. The same goes for classes: normally we read at a higher level of abstraction and get a rough idea of what the code does. But when everything is piled into one class, there is no way to tell a detail from something important, so there is no way to tell what is worth reading.

Maintenance is the next problem. A class with that many responsibilities touches so much of the application that any change to it can affect code somewhere else entirely.

Testing is where this hurts most. Classes like this are hard or impossible to test, so their tests are usually poor or missing. Without them, every change is a gamble, since nothing will warn us when something breaks. The risk of introducing a bug is far higher than with small classes.

Debugging is torture. Finding which of several thousand lines caused a problem can take a very long time.

Large classes are also difficult to reuse. Say one of a hundred methods reads data from a database, and we want that method elsewhere. It comes with the whole god object attached, and calling it means constructing something that probably takes dozens of constructor parameters and a complicated setup.

So how do we end up with small, readable classes? The main tool is the **Single Responsibility Principle**, since large classes almost always handle more than one responsibility.

Recognizing when a large class is acceptable

So far the message has been that large classes are a problem. But what about a class that follows the Single Responsibility Principle and is still large?

This happens most often with data structures. Take the `List` class from the C# standard library, which provides a collection that can grow and shrink as items are added and removed. Its source code runs to more than 1,200 lines. Is that a lot? Yes. Is it too much? Not necessarily, and it is worth going through why.

Start with the Single Responsibility Principle. `List` does not break it. Its responsibility is to provide a dynamic structure holding multiple items of one type, and that requires methods for manipulating those items, such as `Add` and `Remove`. There are many more methods than those two, and they are what make the class large, but they all serve the same responsibility.

Splitting it would feel forced. One class for adding items and another for removing them makes little sense. And those classes would need access to the `List`'s private data, which would mean

the data could no longer be private. If a `ListAdder` and a `ListDeleter` could reach inside the `List`, so could anything else.

So `List` is large, and it has good reasons to be. Its size comes from the number of operations a collection has to support, not from a mix of unrelated jobs that drifted into one place. That is the distinction worth making whenever we look at a big class: the question is not how many lines it has, but whether they all serve one purpose.

Avoiding helper classes that only hide size

Extracting a helper class is not a bad move in itself. It is usually a good one, especially when the extracted class represents a concept of its own. But extracting a class only to reduce the number of methods in another one rarely helps. If the methods we move still belong with the ones we left behind, the split is arbitrary. We have not separated two responsibilities, we have cut one in half, and now it takes two files to understand something that used to take one.

Naming gives this away. If we have a class called `Abc` and the best name we can find for the extracted one is `AbcHelper`, *the split is probably forced*.

Clean code needs the bigger picture. Focusing too hard on one measure, such as class size, tends to cost us somewhere else. A large class is a good reason to stop and consider whether something should be extracted from it. But if an honest look, ideally with a colleague or two, says the class has one responsibility, it can stay as it is.

One caveat before moving on. Large classes are sometimes justified, but rarely. The point here is not that large classes are fine, because in the vast majority of cases they are not. The point is that a class of more than two hundred lines is not automatically a failure to be shortened at any cost, and that forced shortening can do more damage than the length ever would.

Avoiding superficial ways of making code shorter

Short code beats long code, but not at any cost. Clarity comes first. If making a class or a method shorter also makes it harder to follow, the change is not worth it.

Refactoring toward better code often makes a codebase longer overall. We add classes, extract methods, and replace short names with descriptive ones. All of those are improvements, and the extra space they take in the source files is not a problem.

One thing that goes wrong when developers set out to shorten their code is what could be called a *clever hack*. Simple code that happens to take a few lines gets rewritten into something

more compact, which does the same job in fewer characters and takes far longer to understand. Here is an example.

```
int CalculateSumOfEvenNumbers(int n)
{
    int sum = 0;
    for (int i = 2; i <= n; i += 2)
    {
        sum += i;
    }
    return sum;
}
```

The same behavior can be compressed into a single expression:

```
int CalculateSumOfEvenNumbers(int n)
{
    return Enumerable.Range(2, n - 1)
        .Where(x => x % 2 == 0)
        .Sum();
}
```

The shorter version is not automatically cleaner. Different teams may prefer different idioms, but *the decision should be based on how easily the intended readers can understand the code*, not on the number of physical lines. Code review is a better guide than a contest to minimize line count.

Using class names to identify responsibilities

A class name is one of the most practical diagnostic tools for class design. A focused class is usually easy to name because its responsibility is clear. A class with several unrelated responsibilities often forces the developer into one of three naming mistakes: a very long compound name, an incomplete name that hides behavior, or a vague catch-all name.

Compound names reveal multiple jobs

Imagine a class that both confirms medical appointments and sends visit notifications. A literal name such as `AppointmentsConfirmerAndPatientNotificationsSender` accurately describes

the implementation, but the word "And" exposes the design problem: the class contains two responsibilities.

```
public class AppointmentsConfirmerAndPatientNotificationsSender
{
    public void ConfirmAppointment(
        DateTime appointmentDate,
        Patient patient,
        Doctor doctor)
    {
        // Implementation
    }

    public void NotifyAboutVisit(
        DateTime appointmentDate,
        Patient patient,
        Doctor doctor)
    {
        // Implementation
    }
}
```

A better design separates the responsibilities into `AppointmentsConfirmer` and `PatientNotificationSender`. Each class can then be named honestly and changed independently.

Incomplete names hide responsibilities

Renaming the same class `PatientNotificationSender` without removing appointment confirmation is worse. The name now suggests a narrower responsibility than the class actually has. Such names often appear when a class begins with one purpose and later accumulates new behavior without being restructured.

Vague names are a warning sign

Names such as `Manager`, `Handler`, `Helper`, `Service`, `Utility`, `Utils`, or `Processor` are not inherently wrong, but they deserve scrutiny when they are the only way to describe a class. A name such as `AppointmentsManager` may mean that the class confirms appointments, sends notifications, persists data, performs validation, and handles anything else related to appointments. The vagueness can be a symptom of a catch-all design.

Another warning sign is a name that combines different **abstraction levels**.

UserInterfaceStringProcessor mixes a high-level user-interface responsibility with low-level string processing. Those concepts can collaborate, but they do not need to live in the same class.

Signal	What it may indicate	Possible response
A long name containing "And"	The class performs multiple distinct jobs.	Extract each responsibility into a focused class
A name that omits existing behavior	Responsibilities were added without updating the design.	Move the extra behavior rather than hiding it behind the old name
Manager/Helper/Utility-style name	The class may be a catch-all.	List what the class actually does and look for cohesive groups
Concepts from different abstraction levels	High-level orchestration and low-level mechanics are mixed.	Move low-level behavior behind a dedicated collaborator
A class name changes whenever requirements change	The responsibility boundary is unstable.	Re-evaluate whether independent change reasons should be separated

Table 11.1: Naming signals that can reveal multiple class responsibilities

Finding boundaries through data and dependencies

Names provide an external clue, but the body of the class often contains stronger evidence.

Look at which fields and dependencies each method uses. If one group of methods consistently works with one set of data while another group uses a different set, the class may contain two independent concepts.

Separating personal data from account data

Consider a simplified `UserAccount` class that stores both personal information and login-related information:

```
public class UserAccount
{
    private string _firstName;
    private string _lastName;
    private DateTime _dateOfBirth;
    private string _username;
    private string _password;
    private DateTime _lastLogin;

    public string GetFullName()
    {
        return $"{_firstName} {_lastName}";
    }

    public int GetAge()
    {
        // Simplified for this example.
        return DateTime.Now.Year - _dateOfBirth.Year;
    }

    public bool ValidateCredentials(string username, string password)
    {
        return _username == username && _password == password;
    }

    public void UpdateLastLogin()
    {
        _lastLogin = DateTime.Now;
    }
}
```

`GetFullName` and `GetAge` operate only on the personal fields. `ValidateCredentials` and `UpdateLastLogin` operate only on the account fields. The two clusters are largely independent. Rearranging the members so that these groups sit together makes the separation visible and

suggests two classes: one representing a person and another representing account credentials and login state.

This technique is useful because it relies on actual dependencies rather than intuition. If two groups of methods do not share state or collaborators, ask why they belong to the same class.

Grouping behavior that changes together

Another way to discover responsibilities is to examine the history of change. When a particular group of methods is repeatedly modified for the same business reasons, those methods may represent a cohesive responsibility that deserves its own class.

Extracting billing behavior from customer data

Suppose a `CustomerService` class stores a customer's contact information and orders, but also calculates billing.

```
class CustomerService(  
    string firstName,  
    string lastName,  
    string email,  
    string phoneNumber)  
{  
    private string _phoneNumber = phoneNumber;  
    private List<Order> _orders = new List<Order>();  
  
    public void AddOrder(Order order)  
    {  
        _orders.Add(order);  
    }  
  
    public void UpdatePhoneNumber(string phoneNumber)  
    {  
        _phoneNumber = phoneNumber;  
    }  
  
    //changes happen frequently in this method  
    //it would be better to have it in a separate class  
  
    public decimal CalculateBilling()  
    {  
        //a complex algorithm using _orders
```

```
}  
}
```

The class may look reasonable at first, since billing works with the customer's orders. The change history says otherwise. Tax rules, discount thresholds, and billing policies all change often, while the customer data itself rarely does. If product owners keep asking for changes that only touch the billing logic, then billing has a reason to change of its own. Extracting a `BillingCalculator` makes that responsibility explicit:

```
public class BillingCalculator  
{  
    public decimal Calculate(IEnumerable<Order> orders)  
    {  
        //billing calculation  
    }  
}
```

Once billing is extracted, the remaining class can be renamed from the vague `CustomerService` to `Customer` because its role is simply to represent customer data and orders. The name becomes simpler because the **responsibility** became simpler.

Separating levels of abstraction

A class can also reveal multiple responsibilities when its methods operate at very different levels of abstraction. High-level methods describe what the application is trying to accomplish; low-level methods describe how individual technical operations are performed. Mixing the two can bury intent beneath implementation detail.

Moving string-cleaning details out of blog-post storage

Consider a `BlogPostsStorage` class with a high-level `SaveBlogPost` method. A reader can understand its intent from the method body: clean the content and save it. However, the private `Clean` method uses a **regular expression** to normalize repeated spaces. That is low-level string-processing logic.

```
public class BlogPostsStorage  
{  
    private readonly Regex _regex = new Regex("[ ]{2,}", RegexOptions.None);  
    private readonly IPostsRepository _postsRepository;  
  
    public BlogPostsStorage(IPostsRepository postsRepository)
```



```
{
    _postsRepository = postsRepository;
}

public void SaveBlogPost(string content, DateTime createdDate)
{
    var cleanedContent = Clean(content);
    _postsRepository.Save(cleanedContent, createdDate);
}

private string Clean(string content)
{
    return _regex.Replace(content, " ");
}
}
```

String normalization is not specific to blog-post storage and can be extracted into a lower-level string-processing component.

```
public static class StringExtensions
{
    private static readonly Regex Regex = new Regex("[ ]{2,}", RegexOptions.None);

    public static string RemoveExcessiveSpaces(this string content)
    {
        return Regex.Replace(content, " ");
    }
}
```

The high-level class can now read almost like a description of the use case:

```
public void SaveBlogPost(string content, DateTime createdDate)
{
    var cleanedContent = content.RemoveExcessiveSpaces();
    _postsRepository.Save(cleanedContent, createdDate);
}
```

The benefit goes beyond having fewer lines in `BlogPostsStorage`. The storage class now stays at one level of abstraction. And the string operation, which had nothing to do with storage in the first place, can now be called from anywhere in the application that needs it.

Avoiding unnecessary class proliferation

Refactoring toward smaller classes leaves us with more classes than we started with, and often with more lines of code in the project overall. That is expected, and it is a good trade. What we get in return is **granularity**.

A codebase made of small, well-named pieces can be read at a high level, without opening any of them. Consider an `AccountStorage` class working with `ICredentialsValidator`, `IUsersRepository`, and `IUserCommunication`. The workflow reads off the names alone: validate the credentials, save the account, tell the user what happened. Nothing stops us from opening any of those implementations if we want the details. The point is that it becomes a choice rather than a requirement.

```
public class AccountStorage
{
    private readonly ICredentialsValidator _credentialsValidator;
    private readonly IUsersRepository _usersRepository;
    private readonly IUserCommunication _userCommunication;

    public void CreateNewAccount(string username, string password)
    {
        if (_credentialsValidator.IsValidUsername(username) &&
            _credentialsValidator.IsValidPassword(password))
        {
            _usersRepository.Save(username, password);
            _userCommunication.ShowMessage("Account created successfully.");
        }
        else
        {
            _userCommunication.ShowWarning("Invalid user credentials.");
        }
    }
}
```

The goal is therefore not "as many classes as possible." Extract only when the new type expresses a coherent responsibility or abstraction. Avoid both extremes: one class that owns everything and dozens of tiny types that have no independent meaning.

Question to ask	Evidence for extraction	Evidence for keeping together
Do the methods use the same data?	Distinct groups use different fields or dependencies.	Most behavior works with the same cohesive state.
Do they change for the same reasons?	One group changes repeatedly for a separate business concern.	Changes normally affect the same set of members together.
Are they at the same abstraction level?	High-level orchestration and low-level mechanics are mixed.	Methods describe one coherent concept at a consistent level.
Can the new class be named clearly?	The extracted responsibility has an honest, specific name.	The best name is only <code>OriginalClassHelper</code> or another vague label.
Does extraction improve encapsulation?	The new class owns the data it needs or receives a clean abstraction.	Extraction requires exposing private implementation details.
Would the component be useful independently?	The behavior can reasonably be reused or tested in isolation.	It has no meaning outside the original class and exists only to reduce lines.

Table 11.2: Questions for deciding whether to extract a class

Refactoring case study: Decomposing `PizzaGenerator`

Let's apply the principles of smaller, focused classes to a `PizzaGenerator` that has gradually taken on more than one responsibility.

Understanding the code

We will work with the `PizzaGenerator` type. Its job is to put together a pizza from randomly chosen ingredients, given how many ingredients it should have and which diet it needs to fit:

```
public class PizzaGenerator
{
```

```
private const string IngredientsFile = "ingredients.txt";
private readonly IFileAccess _fileAccess;
private readonly IRandom _random;

public PizzaGenerator(IRandom random, IFileAccess fileAccess)
{
    _random = random;
    _fileAccess = fileAccess;
}
//...
}
```

It has two dependencies. `IRandom` generates random numbers, and `IFileAccess` reads all the lines from a text file.

Let's see its main method:

```
public Pizza Generate(int countOfIngredients, DietOption dietOption)
{
    var allIngredientsForCurrentDietOption =
        GetAllIngredients()
        .Where(ingredient => IsOkInDiet(dietOption, ingredient))
        .ToList();
    var selectedIngredients =
        Shuffle(allIngredientsForCurrentDietOption)
        .Take(countOfIngredients)
        .ToList();
    return new Pizza(selectedIngredients);
}
```

It gets all the possible ingredients, keeps only those allowed by the given diet option, shuffles them, and takes as many as it needs.

And here is the method that reads the available ingredients, along with the `Ingredient` type itself.

```
private List<Ingredient> GetAllIngredients()
{
    var result = new List<Ingredient>();
    var allLines = _fileAccess.ReadAllLines(IngredientsFile);

    foreach (var line in allLines)
```

```
{
    var split = line.Split(',');
    Enum.TryParse(split[1], ignoreCase: true, out DietOption dietOption);
    result.Add(new Ingredient(split[0], dietOption));
}
return result;
}

public class Ingredient(string name, DietOption dietOption)
{
    public string Name { get; } = name;
    public DietOption DietOption { get; } = dietOption;
    public override string ToString() => Name;
}
```

This method reads the ingredients from the ingredients file. Each line in that file holds one ingredient name, followed by a comma and the diet option it matches. GetAllIngredients reads the lines one by one, splits each on the comma, and builds an Ingredient object from the name and the diet option.

Let's also see the IsOkInDiet method:

```
private bool IsOkInDiet(DietOption dietOption, Ingredient ingredient)
{
    if (dietOption == DietOption.Omnivore)
    {
        return true;
    }
    if (dietOption == DietOption.Vegan)
    {
        return ingredient.DietOption == DietOption.Vegan;
    }
    return
        ingredient.DietOption == DietOption.Vegan ||
        ingredient.DietOption == DietOption.Vegetarian;
}
```

It checks whether a given ingredient is acceptable for a given diet option. An omnivore can eat anything, whether it is vegan, vegetarian, or meat. A vegan eats only vegan ingredients. A vegetarian eats typically vegetarian ingredients, such as cheese or eggs, and vegan ones as well.

Finally, there is a `Shuffle` method. The details of how it works are not important here. It randomizes the order of elements in a collection.

```
public ICollection<T> Shuffle<T>(ICollection<T> collection)
{
    var collectionCopy = collection.ToList();
    int n = collectionCopy.Count;
    while (n > 1)
    {
        n--;
        int k = _random.Next(n + 1);
        T value = collectionCopy[k];
        collectionCopy[k] = collectionCopy[n];
        collectionCopy[n] = value;
    }
    return collectionCopy;
}
```

Identifying the issues

There are a few red flags here suggesting that `PizzaGenerator` should be split into smaller classes.

The first is size. Not huge, but certainly not small.

The second is the pair of dependencies. A randomizer and file access have little to do with each other, and it becomes clearer once we look at what they are for: one reads files, the other shuffles collection items.

The third is that the class mixes levels of abstraction. `Generate` sits at a high level, while `Shuffle` is a low-level mechanism. It is easy to imagine wanting to shuffle a collection somewhere else, in a context that has nothing to do with pizza, and right now that is not possible. Even making the method `public` would not help, since any code calling it would need a `PizzaGenerator` object first.

All of which makes this class a good candidate for a split.

Improving the code

Let's practice splitting classes, starting with `GetAllIngredients`. This method belongs in a class of its own, so let's move it there. It has to become `public`, and the new class needs an interface, so that anything using it depends on an abstraction rather than a concrete type. The

IFileAccess dependency and the file name move across as well, since nothing else needs them.

```
public interface IIngredientsRepository
{
    List<Ingredient> GetAllIngredients();
}

public class IngredientsRepository : IIngredientsRepository
{
    private const string IngredientsFile = "ingredients.txt";
    private readonly IFileAccess _fileAccess;

    public IngredientsRepository(IFileAccess fileAccess)
    {
        _fileAccess = fileAccess;
    }

    public List<Ingredient> GetAllIngredients()
    {
        var result = new List<Ingredient>();
        var allLines = _fileAccess.ReadAllLines(IngredientsFile);
        foreach (var line in allLines)
        {
            result.Add(BuildIngredientFrom(line));
        }
        return result;
    }

    private static Ingredient BuildIngredientFrom(string line)
    {
        var split = line.Split(',');
        Enum.TryParse(split[1], ignoreCase: true, out DietOption dietOption);
        return new Ingredient(split[0], dietOption);
    }
}
```

One more improvement worth doing was extracting the `BuildIngredientFrom` method that builds a single ingredient from a line of the text file. The method is private and uses none of the class state, so it can safely be static.

The complete lack of error handling here is not ideal. The method does not anticipate a line without a comma, or one where the diet option is missing. But we are refactoring, and adding error handling would go beyond that, so it stays as it is.

PizzaGenerator will now take a dependency on `IIngredientsRepository`, so reading the file is no longer its concern. And since it will depend on an abstraction rather than a concrete implementation, we will follow the **Dependency Inversion Principle**. If we ever decide to read the ingredients from a different source, we can do it without touching `PizzaGenerator` at all.

Let's carry on splitting classes, and pick up a useful trick for working out which class a method really belongs to.

The next one to deal with is `IsOkInDiet`. Knowing whether eggs are allowed in a vegan diet is not the job of `PizzaGenerator`.

The trick is to look at the method's parameters. There is a good chance the method belongs to one of them. It will not be `DietOption`, which is just an enum, but `Ingredient` looks like a good candidate. It makes sense for an ingredient to know which diets it fits. So let's move the method there.

```
public class Ingredient(string name, DietOption dietOption)
{
    public string Name { get; } = name;
    public DietOption DietOption { get; } = dietOption;

    public bool IsOkInDiet(DietOption dietOption)
    {
        //implementation
    }

    public override string ToString() => Name;
}
```

The last method to address is `Shuffle`. It takes a collection, randomizes the order of its items, and returns the shuffled result. That is functionality we can easily imagine needing outside `PizzaGenerator`, so let's move it to a new `CollectionRandomizer` class.

```
public interface ICollectionRandomizer
{
    ICollection<T> Shuffle<T>(ICollection<T> collection);
}
```



```
public class CollectionRandomizer : ICollectionRandomizer
{
    Private readonly IRandom _random;

    public CollectionRandomizer(IRandom random)
    {
        _random = random;
    }

    public ICollection<T> Shuffle<T>(ICollection<T> collection)
    {
        //implementation
    }
}
```

We can now bring all of this together in the new and improved `PizzaGenerator` class:

```
public class PizzaGenerator
{
    private readonly IIngredientsRepository _ingredientsRepository;
    private readonly ICollectionRandomizer _collectionRandomizer;

    public PizzaGenerator(
        IIngredientsRepository ingredientsRepository,
        ICollectionRandomizer collectionRandomizer)
    {
        _ingredientsRepository = ingredientsRepository;
        _collectionRandomizer = collectionRandomizer;
    }

    public Pizza Generate(int countOfIngredients, DietOption dietOption)
    {
        var allIngredientsForCurrentDietOption =
            _ingredientsRepository.GetAllIngredients()
                .Where(ingredient => ingredient.IsOkInDiet(dietOption))
                .ToList();
        var selectedIngredients =
            _collectionRandomizer.Shuffle(allIngredientsForCurrentDietOption)
                .Take(countOfIngredients)
                .ToList();
    }
}
```

```
        return new Pizza(selectedIngredients);  
    }  
}
```

`PizzaGenerator` now has a clearly defined responsibility. It generates pizzas, and nothing else. It relies on other types to do that, but it depends on abstractions rather than concrete implementations, so the Dependency Inversion Principle holds.

`IngredientsRepository` does one thing as well: it builds a collection of ingredients from a given file. It depends on the abstract `IFileAccess`, which handles the low-level work of reading the text content of a file.

`CollectionRandomizer` shuffles a collection, and we can now use it anywhere that needs shuffling.

The result is a program built from small, focused types, each with a single responsibility.

Summary

This chapter was about breaking classes apart without treating line count as the rule that decides everything. We looked at god classes and at large classes that have good reasons to be large, at helper extractions that make a design worse rather than better. We also went through the signals worth watching: what the names suggest, which data a method actually uses, what tends to change together, and whether everything in a class sits at the same level of abstraction. The `PizzaGenerator` case study showed how a set of focused classes makes responsibilities, dependencies, testing, and reuse clearer than one class doing everything. In *Chapter 12*, you will look one level above individual classes and organize members, files, folders, and project conventions so that the codebase is easier to navigate.

12

Organizing classes and projects

A codebase can be full of perfectly reasonable classes and still be hard to find your way around. The problem is not how those classes behave. It is where things are. Members turn up in a different order in every class, one file holds several types, and the **folder structure** gives no clue about how the application is organized.

All of that is fixable, and it comes down to *putting things where people will look for them*. Inside a class, the public **methods** and the **dependencies** should be easy to spot. Across a project, the files and folders should tell us what the application is made of. And the team should agree on how it does these things, so that reading someone else's code holds no surprises. That is what this chapter is about.

In this chapter, we will cover the following topics:

- Ordering class members for readability
- Why one class per file should be the default
- Structuring project folders by concern or feature
- Maintaining consistent behavior and naming within classes
- Recording project-wide conventions
- Step-by-step approach to restructuring existing code

By the end of this chapter, you will be able to order **class members** predictably, choose sensible file boundaries, organize folders around meaningful concerns or features, apply project-wide **conventions** consistently, and reorganize a codebase without introducing unnecessary structural complexity.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Ordering class members for readability

The compiler may not care whether a field appears above or below a method, but the reader does. *A predictable member order reduces the amount of searching required to understand what a class exposes*, how it is created, what it depends on, and where its implementation details live. The guiding question is simple: what will a reader look for first? Put that information where it can be found quickly.

Public members normally deserve a prominent position because they describe the class from the outside. **Constructors** show how objects are created, and public methods reveal what the class can do. **Fields** are a separate case: they usually come first regardless of visibility, because they describe what the class is made of, and the reader needs the state before the behavior that acts on it. **Private methods** are **implementation detail**, so they can go further down. Anyone who wants them will scroll, but what a reader wants to see first is what the class offers.

Method order can also communicate flow. If one method calls another, placing the caller above the method it uses lets a reader move from high-level intent to lower-level detail. *These are recommendations rather than rigid rules. A team convention that is applied consistently is more valuable than repeatedly rearranging code to satisfy a personal preference.*

Member group	Typical placement	Reason
Fields and properties	Near the top	They describe what data the class holds.

Member group	Typical placement	Reason
Dependencies and other private fields	Near the top, grouped consistently	Dependencies are important context; grouping fields avoids surprising declarations later in the class.
Constructors	Near the data and dependencies they initialize	Readers can quickly see how an instance must be created.
Public methods	Before private implementation methods	They describe the behavior available to callers.
Private methods	Below the public behavior they support	Implementation details should follow the higher-level operation that uses them.
Nested types	Public near the public surface; private near the bottom	Accessibility should influence prominence.

Table 12.1: Practical guidelines for ordering class members

Organizing classes into separate files

A useful default is one type per file, with the file named after the type. A `DatabaseConnector` is easier to locate when it lives in `DatabaseConnector.cs` than when it is one of several unrelated types hidden in a general-purpose file. This convention improves both navigation and the ability to understand a project from its file list.

Separate files also make version-control history more informative. When a commit changes three class files, the changed file list immediately tells a reviewer which types were affected. If twenty classes live in one large file, version control can only report that the single file changed. Smaller files also reduce merge conflicts because teammates are less likely to edit the same physical file at the same time.

The rule should not become dogma. Very small types that clearly belong together can sometimes share a file without making anything harder to find. Suppose an application defines its own enums for days of the week, months, and seasons. Each one is a handful of lines, and none of

them will change; anyone looking for one of them would think to look for the others in the same place. Putting all three in a `DateEnums.cs` file does no harm.

Structuring project files and folders

Once each class has a file of its own, the project needs a structure that makes those files easy to find. There is no single folder layout that suits every application.

Frameworks often settle this for us. A new project created from a template arrives with folders already in place, and each of them has an established purpose that the framework's own conventions depend on. Following that layout is usually the right call. Anyone who has worked with the framework before will know where to look without being told, tooling and documentation assume the same structure, and in some cases the framework itself expects files in particular places.

Beyond what the framework dictates, the folder hierarchy should reflect the architecture of the application and the way the team thinks about it.

Two common approaches are organizing by **technical concern** and organizing by **feature**.

Approach	Example folders	Strength	Consideration
By concern	DataAccess, UI, BusinessLogic, Models	Makes architectural layers and technical responsibilities explicit.	A single feature may be spread across several folders
By feature	Login, ItemSearch, ShoppingCart	Keeps the code required for one user-facing capability close together.	Each feature folder may contain several technical concerns

Table 12.2: Organizing project folders by concern or by feature

Both structures can be clean. *What matters is picking the one that fits the application and staying consistent with it.* When a folder gets crowded, or starts holding things that have nothing to do with each other, subfolders help. A `DataAccess` folder might get `Database`, `Cloud`, and `FileSystem` inside it once each of those has enough code to be worth separating.

As with everything else in clean code, the right structure often emerges gradually rather than being designed up front, so there is no reason to hesitate about reorganizing later. One piece of advice, though: keep file moves in their own commits, separate from changes to the code inside

those files. Otherwise, the version-control history shows a large diff that is hard to read, and a change worth reviewing gets lost among files that only moved.

Test projects benefit from the same predictability as any other code. Keeping tests in a separate project is worth doing on its own merits, since test code should not be built or shipped along with the application. Mirroring the structure of the production project on top of that also means the test for a given class is easy to find, without learning a second scheme for where things live.

Maintaining consistency within classes

Organization is not only physical. A class also forms a small API, and its methods should behave in ways that do not surprise callers. Consider a `PersonalDataAccess` class in which three lookup methods report a missing person in three different ways.

The inconsistent behavior can be summarized by the following example:

```
public Person GetById(int id)
{
    var person = _peopleRepository.GetById(id);
    if (person is null)
    {
        throw new KeyNotFoundException($"No person with id {id} was found.");
    }
    return person;
}

public Person GetByNames(string firstName, string lastName)
{
    var person = _peopleRepository.GetByNames(firstName, lastName);
    if (person is null)
    {
        Console.WriteLine("Person not found.");
    }
    return person;
}

public bool GetByDateOfBirth(DateTime dateOfBirth, out Person result)
{
    result = _peopleRepository.GetByDateOfBirth(dateOfBirth);
}
```



```
        return result is not null;
    }
```

All three methods deal with the same situation, a person who is not in the repository, and each of them handles it differently. The first throws an exception, the second prints to the console and returns null, and the third returns false and passes the result through an output parameter.

That is a problem for whoever calls this class. Having learned how `GetById` behaves, they have every reason to assume the other two work the same way, and they are wrong. So each method has to be looked up separately, and each call site handles the same situation differently: a try-catch here, a null check there, and a bool check somewhere else. The inconsistency inside this one class turns into scattered complexity in every class that uses it.

The fix is to pick one convention that suits the application and apply it consistently within a type. The same goes for naming: `Get` and `Fetch` should not be used interchangeably for the same kind of operation unless the difference means something. Conventions like these are worth writing down when they settle a question that keeps coming up, and that includes conventions for naming unit tests.

Let's see the refactored version of the code:

```
public Person GetById(int id)
{
    var person = _peopleRepository.GetById(id);
    if (person is null)
    {
        throw new KeyNotFoundException(
            $"No person with id {id} was found.");
    }
    return person;
}

public Person GetByNames(string firstName, string lastName)
{
    var person = _peopleRepository.GetByNames(firstName, lastName);
    if (person is null)
    {
        throw new KeyNotFoundException(
            $"No person named {firstName} {lastName} was found.");
    }
}
```

```
        return person;
    }

    public Person GetByDateOfBirth(DateTime dateOfBirth)
    {
        var person = _peopleRepository.GetByDateOfBirth(dateOfBirth);
        if (person is null)
        {
            throw new KeyNotFoundException(
                $"No person born on {dateOfBirth.ToShortDateString()} was found.");
        }
        return person;
    }
}
```

Refactoring project and class structure

When an existing project has become hard to navigate, reorganize it in small steps that can be reviewed on their own, rather than mixing structural moves with changes to behavior.

Start inside the classes. Put the members in a predictable order: fields and properties first, since they describe what the class holds, then the constructors that set them up, then public methods, and the private implementation details further down. Do the same in every class, so that reading a new one holds no surprises.

Then look at the files. **One type per file**, named after the type it holds, so that finding a class means guessing its file name correctly on the first try.

At the project level, choose a folder model that matches the way the team thinks about the application, whether that is by technical concern, by feature, or by whatever the framework expects, and apply it consistently. Split a folder into subfolders once it grows crowded or starts holding unrelated things. Keep the test project separate and mirror the production structure inside it. Remember to *keep commits that move files separate from commits that contain code changes*, so that the version-control history stays readable.

Finally, compare the classes in the area you touched. Similar operations should act the same way, and the same kind of operation should carry the same name across classes rather than being a Get in one and a Fetch in another. Where a convention keeps settling the same recurring question, write it down.

Summary

This chapter focused on the physical and navigational organization of a codebase. You ordered class members so public behavior and dependencies are easy to find, used one-class-per-file as a pragmatic default, compared folder structures by technical concern and by feature, and examined how consistent APIs and naming reduce surprises. The final organization workflow brought these choices together without turning them into rigid rules.

In *Chapter 13*, you will move from where code lives to how classes relate to one another, balancing coupling, cohesion, reuse, duplication, and composition.

13

Balancing coupling, cohesion, and reuse

A system can be built from small, well-organized classes and still resist every attempt to change it.

Classes end up knowing too much about each other. Things that have nothing in common get grouped together. And in the effort to remove every repeated line, we sometimes build **abstractions** harder to follow than the **duplication** they were meant to replace.

The solution is to balance these relationships deliberately. **Loose coupling** limits what one class needs to know about another. **High cohesion** keeps related behavior together. **DRY** stops the same business knowledge from having two competing representations in the code.

All three need judgment in how far they are taken. Duplication is sometimes safer than an abstraction introduced too early, before we know whether the two pieces of code will really change together.

Reuse is a separate question. When behavior needs to be shared, composition is usually the better choice, because it does not lock that behavior into an inheritance hierarchy.

In this chapter, we will cover the following topics:

- Understanding coupling and learning how to reduce it
- Designing classes for high cohesion
- Applying the Law of Demeter
- Applying the DRY principle
- Recognizing when duplication is justified
- Understanding the BDUF, YAGNI, and KISS acronyms
- Understanding the "**Composition** over **inheritance**" guideline

By the end of this chapter, you will be able to identify **tight coupling** and **low cohesion**, distinguish duplicated business knowledge from merely similar code, avoid speculative abstractions, recognize rigid inheritance hierarchies, and refactor toward composition when interchangeable collaborators provide a cleaner design.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Understanding and reducing coupling between classes

Coupling is the degree to which one class depends on another class. It describes how much one piece of code needs to know about another in order to work with it, and it shows up as a measure of how much a change on one side forces changes on the other.

Some **dependency** is unavoidable, since useful classes collaborate. The design problem is not that a class depends on another one. It is that the consumer knows details that should have stayed private, or depends on a concrete type where an **interface** would do. When a small implementation change in one type forces edits in several unrelated consumers, those types are tightly coupled.

Consider a newsletter component that receives a `Subscribers` object. The initial design exposes the internal array directly:

```
public class Subscribers
{
    public Subscriber[] Items { get; }

    public Subscribers(Subscriber[] items)
    {
        Items = items;
    }
}
```

```
    }  
}  
  
public class NewsletterSender  
{  
    private readonly Subscribers _subscribers;  
  
    public NewsletterSender(Subscribers subscribers)  
    {  
        _subscribers = subscribers;  
    }  
  
    public void SendTo(bool premiumSubscribersOnly)  
    {  
        for (int i = 0; i < _subscribers.Items.Length; i++)  
        {  
            if (!premiumSubscribersOnly ||  
                _subscribers.Items[i].IsPremium)  
            {  
                Console.WriteLine(  
                    $"Newsletter sent to {_subscribers.Items[i].Email}");  
            }  
        }  
    }  
}
```

At first glance this looks fine. But notice how much `SendTo` knows about the inside of `Subscribers`. It knows the collection is an array, because it uses `Length` and an index. And since the array elements are reachable through the property, the sender could just as well assign null to one of them:

```
_subscribers.Items[i] = null;
```

If one class can null out the contents of another, those two classes are far too familiar with each other.

The fragility shows up as soon as anything changes. Replacing the array inside `Subscribers` with a `HashSet` would break `SendTo`, even though the **responsibility** of `Subscribers` has not changed at all. A change to the internals of one class should not reach into another one. Here it does.

The fix is for `Subscribers` to hide how it stores the data and expose only what its consumer actually needs, which in this case is the ability to go through the subscribers one by one. In C# that is what `IEnumerable<T>` describes: a sequence that can be iterated, with no promise about what holds it underneath. An array satisfies it, and so do a list, a set, and anything else the class might switch to later.

```
public class Subscribers
{
    private readonly HashSet<Subscriber> _items;
    public IEnumerable<Subscriber> Items => _items;

    public Subscribers(HashSet<Subscriber> items)
    {
        _items = items;
    }
}

public class NewsletterSender
{
    private readonly Subscribers _subscribers;

    public NewsletterSender(Subscribers subscribers)
    {
        _subscribers = subscribers;
    }

    public void SendTo(bool premiumSubscribersOnly)
    {
        foreach (var subscriber in _subscribers.Items)
        {
            if (!premiumSubscribersOnly || subscriber.IsPremium)
            {
                Console.WriteLine(
                    $"Newsletter sent to {subscriber.Email}");
            }
        }
    }
}
```

`NewsletterSender` no longer cares whether `Subscribers` uses an array, list, hash set, or another collection internally. The interface between the two classes is smaller, and the implementation detail can change without forcing a change in the consumer. Exposing `IEnumerable` also prevents `NewsletterSender` from replacing an item through index assignment, which protects the collection from inappropriate manipulation.

Warning sign	Why it matters	Typical response
A class depends directly on a concrete implementation	Replacing or changing the collaborator forces changes in the consumer	Depend on an interface or another suitable abstraction when variation is expected
A small change causes a cascade of edits	Several classes know details that should have remained local to one type	Hide volatile details behind a stable contract
Testing one class means setting up several others	The class cannot be exercised independently and probably carries excessive knowledge	Reduce responsibilities and inject focused collaborators
A consumer manipulates another object's internal data	Encapsulation is broken and implementation choices leak outward	Expose only the operations or read-only view the consumer needs
A class must inspect concrete subtypes to decide what to do	The abstraction is not carrying the behavior required by the client	Move behavior behind the abstraction or redesign the contract

Table 13.1: Warning signs of tight coupling

Applying the Law of Demeter

The **Law of Demeter**, often described as the principle of least knowledge, is another practical way to control **coupling**. *A class should collaborate with the objects it directly receives, owns, or creates rather than reaching through one object to navigate a chain of unrelated internals.*

The following **constructor** receives an entire application context only to extract one directory string:

```
public class FileAccess
{
```



```
private readonly string _mainDirectory;

public FileAccess(ApplicationContext context)
{
    _mainDirectory =
        context.GetAppData().GetDirectory().Main;
}

// Other file access methods
}
```

`FileAccess` now depends not only on `ApplicationContext` but also on the shape of the objects returned by `GetAppData` and `GetDirectory`. A change anywhere along that chain can break a class that only needed one string.

Pass the required value directly instead:

```
public FileAccess(string mainDirectory)
{
    _mainDirectory = mainDirectory;
}
```

The revised constructor exposes the real dependency. It is easier to reuse, easier to test, and unaffected by changes that have nothing to do with it. The general lesson is this: if a class has to walk through a chain of properties and method calls to reach the value it needs, something is wrong. Every link in that chain is a class it now depends on. A class that needs something should ask for that thing, not for something it can dig through to find it.

Designing for high cohesion

Cohesion describes how strongly the members of a class belong together. In a class with **high cohesion**, the data and the behavior inside it all serve one clear concept. **Low cohesion** is usually a sign that separate responsibilities have ended up in the same type.

Cohesion is not really something we actively achieve. It is something we should avoid breaking.

A small collection class is a useful example of high cohesion:

```
class PetsCollection
{
    private readonly List<Pet> _pets = new();
    public int Count => _pets.Count;
}
```

```
public void Add(Pet pet)
{
    _pets.Add(pet);
}

public IEnumerable<PetType> GetCurrentlyStoredTypes()
{
    return _pets
        .Select(pet => pet.PetType)
        .Distinct();
}

public bool Contains(PetType petType)
{
    return GetCurrentlyStoredTypes()
        .Any(type => type == petType);
}
}
```

Every member either works with the same `_pets` collection or builds on another operation in the class. There is no obvious boundary along which the type should be divided.

By contrast, imagine a `HousePricer` that calculates property prices and also sends those prices to the owners. The pricing part works with the area, the number of floors, and a price per square meter. The notification part needs a database of owners and something to send messages with. Neither half uses anything belonging to the other. They meet at exactly one point, where one of them hands over a number, and that is the seam this class should be split along.

```
public class HousePricer
{
    private IOwnersDatabase _ownersDatabase;
    private decimal _dollarsPerSquareMeter;

    public HousePricer(
        decimal dollarsPerSquareMeter,
        IOwnersDatabase ownersDatabase)
    {
        _dollarsPerSquareMeter = dollarsPerSquareMeter;
        _ownersDatabase = ownersDatabase;
    }
}
```

```
}

public decimal GetPrice(House house)
{
    return
        _dollarsPerSquareMeter *
        (decimal)house.Area *
        GetPriceMultiplierBasedOnFloors(house.Floors);
}

private decimal GetPriceMultiplierBasedOnFloors(int floors)
{
    return floors switch { 1 => 1m, 2 => 1.5m, _ => 1.6m };
}

public void SendPriceToOwner(House house)
{
    Console.WriteLine($"Sending price {GetPrice(house)}" +
        $" to {FindOwnerEmail(house.Address)}");
}

private string FindOwnerEmail(string address)
{
    return _ownersDatabase.GetEmailByAddress(address);
}
}
```

Refactoring this would give us two independent classes, each of them highly cohesive:

```
public class HousePricer
{
    public decimal GetPrice(House house) { ... }
}

public class OwnerNotifier
{
    public void SendPriceToOwner(decimal price, string address) { ... }
}
```

The refactored types are easier to name, because each name describes one cohesive purpose. High cohesion and the **Single Responsibility Principle** are closely related, but they are not the

same thing. Splitting every method into a class of its own would satisfy an overly literal reading of one responsibility, and it would destroy cohesion in the process.

The `List` class we looked at earlier is the clearest case. It is large, well past a thousand lines, and it still has one responsibility: providing a collection that grows and shrinks as items are added and removed. It is also highly cohesive, precisely because every one of its methods works on the same data and serves that same concept, whether it adds an item, removes one, or sorts them all. Separating `Add`, `Remove`, `Clear`, and `Sort` into four classes would leave each of them with nothing to hold on to.

Characteristic	High cohesion	Low cohesion
Private data usage	Most members use the same state or related collaborators.	Separate groups use unrelated fields or dependencies.
Functional relationship	Behaviors contribute to one concept.	Behaviors represent different concerns.
Ease of splitting	Natural split points are difficult to find.	Independent groups can be extracted cleanly.
Naming	A precise class name describes the whole type.	The name becomes vague, compound, or misleading.

Table 13.2: Characteristics of high and low cohesion

Applying the DRY principle to knowledge

The **Don't Repeat Yourself (DRY)** principle is often reduced to the instruction not to copy and paste code. Its more useful design meaning is broader: *a piece of knowledge should have one authoritative representation in the system*. **Business rules**, algorithms, and other decisions should not be independently encoded in several places that can drift apart.

Suppose an online store allows a product to be returned within 30 days of purchase. If the number 30 is embedded independently in several methods, the same business policy has several representations.

```
public DateTime GetReturnDateDeadline(DateTime purchaseDate)
{
    return purchaseDate.AddDays(30);
}
```

```
public bool IsAfterReturnDateDeadline(DateTime purchaseDate)
{
    return (DateTime.Now - purchaseDate).TotalDays > 30;
}
```

If the policy changes, every copy must be found and changed correctly. Represent the rule once instead:

```
private const int DaysForReturn = 30;

public DateTime GetReturnDateDeadline(DateTime purchaseDate)
{
    return purchaseDate.AddDays(DaysForReturn);
}

public bool IsAfterReturnDateDeadline(DateTime purchaseDate)
{
    return GetReturnDateDeadline(purchaseDate) < DateTime.Now;
}
```

Mechanical **code duplication** is related but not identical. For example, two validations may repeat the same null-or-empty check even though the business rule itself is not duplicated.

```
public void CommitOrder(Order order)
{
    if (string.IsNullOrEmpty(order.CustomerId))
    {
        throw new ArgumentException($"CustomerId must not be empty.");
    }

    if (string.IsNullOrEmpty(order.ProductId))
    {
        throw new ArgumentException($"ProductId must not be empty.");
    }

    //saving to database here...
    Console.WriteLine("Order committed and saved to database.");
}
```

In that case, extracting a small validation method can reduce maintenance work without changing the domain model.

```
private void Validate(string idToBeValidated, string propertyName)
{
    if (string.IsNullOrEmpty(idToBeValidated))
    {
        throw new ArgumentException(
            $"{propertyName} must not be empty.");
    }
}

public void CommitOrder(Order order)
{
    Validate(order.CustomerId, nameof(order.CustomerId));
    Validate(order.ProductId, nameof(order.ProductId));
    // Save to the database.
    Console.WriteLine("Order committed and saved to database.");
}
```

Knowing when duplication is justified

Removing every visual similarity can be as dangerous as copy-pasting indiscriminately. Two pieces of code can look the same today while representing different **business rules** that are likely to change independently tomorrow.

Back to the e-commerce example. Customers have 30 days to return the goods they bought. Now a new requirement arrives: once the goods come back, customer service has 30 days to issue the refund. We need to calculate the refund deadline and check whether it has passed.

```
private const int DaysForReturn = 30;
private const int DaysForRefund = 30;
public DateTime GetReturnDateDeadline(DateTime purchaseDate) =>
    purchaseDate.AddDays(DaysForReturn);
public DateTime GetRefundDateDeadline(DateTime returnDate) =>
    returnDate.AddDays(DaysForRefund);
public bool IsAfterReturnDateDeadline(DateTime purchaseDate) =>
    IsBeforeNow(GetReturnDateDeadline(purchaseDate));
public bool IsAfterRefundDateDeadline(DateTime returnDate) =>
    IsBeforeNow(GetRefundDateDeadline(returnDate));
```

```
private bool IsBeforeNow(DateTime dateTime) =>
    dateTime < DateTime.Now;
```

The code for that looks almost exactly like the code for the return deadline, only with a different starting date. Copy-pasting is something we have all been told not to do, so the tempting move is to have the refund **methods** call the return ones. The **duplication** would be gone and the code would still work. This is how such refactor could look like:

```
public DateTime GetRefundDateDeadline(DateTime returnDate) =>
    GetReturnDateDeadline(returnDate);
public bool IsAfterRefundDateDeadline(DateTime returnDate) =>
    IsAfterReturnDateDeadline(returnDate);
```

But it is the wrong call. Those two deadlines have nothing to do with each other. Returns and refunds are separate business areas, and the 30 days match by coincidence, not for any shared reason. Sooner or later one of them changes. The return window gets extended to 100 days, or it starts depending on the customer's status, and now we have to pull apart code that was written as if the two rules were one. In a small project that is annoying. In a real application it can be very hard.

So we keep them apart from the start. Two separate constants, even though they hold the same number for now.

This leaves a little **duplication** in the code. In exchange, two things that should stay separate for business reasons are separate, and when one of them changes, nothing has to be untangled first.

Situation	What is duplicated?	Preferred response
The 30-day return rule appears in several methods	Business knowledge	Create one authoritative representation and reuse it
Two ID checks contain the same null-or-empty validation	Implementation code	Extract the repeated mechanism when doing so improves clarity
Return and refund deadlines both happen to be 30 days	Possibly nothing; the policies may be independent	Keep the business rules separate unless they are truly the same rule

Table 13.3: Distinguishing duplicated knowledge from duplicated code

Avoiding premature abstraction

Temporary **duplication** can also be safer when the requirements are still uncertain. Two pieces of code look alike, so we assume they belong together, and we build an **abstraction** to cover both. Often we go further and design it to accommodate the cases we expect to arrive later. Then the requirements settle, and the similarity turns out to have been superficial. What we are left with is a hierarchy of **interfaces** and base classes serving a problem that never materialized, shaped around assumptions that were wrong.

Waiting costs less. *Let the requirements mature, and refactor once the shared parts have actually revealed themselves.*

Three common acronyms are useful here: **Big Design Upfront (BDUF)** names the trap: designing detailed abstractions before the problem is understood. **You Ain't Gonna Need It (YAGNI)** and **Keep It Simple, Stupid (KISS)** are the reminders that help us avoid it. None of them prohibits planning or **abstraction**. They push back against complexity that no current requirement supports.

Principle	Practical meaning	Question to ask
BDUF	Do not commit to detailed abstractions before the problem is understood well enough	Am I solving a current design problem or an imagined future one?
YAGNI	Do not implement extension points or features until a real requirement needs them	What evidence says this capability will actually be used?
KISS	Prefer the simplest design that preserves clarity and required capability	Can I remove complexity without making the code less expressive or less correct?

Table 13.4: Practical checks against speculative complexity

KISS does not mean avoiding advanced language features because a junior developer might not know them yet. Every language has constructs that take some learning, and using one is fine when it expresses the intent more directly than the alternative would. The goal is not to write at the lowest possible level of the language. It is to write the simplest code that says clearly what it does.

Deferring design decisions behind abstractions

Sometimes the uncertainty is about infrastructure rather than about an algorithm. Say the team has not decided whether the data will live on the company's own servers or in the cloud. The rest of the application does not have to wait for that. Define what it needs from a data source, put it behind an interface, and work against a simple test implementation until the decision is made. The real one can be written later, and nothing else will have to change when it arrives.

```
public interface IDataAccess
{
    IEnumerable<Record> GetAll();
}

public class DataAccessMock : IDataAccess
{
    public IEnumerable<Record> GetAll()
    {
        return new[] { new Record() };
    }
}
```

The **abstraction** is justified because the application already needs a data-access capability; only the implementation is undecided. This differs from inventing extension points for hypothetical capabilities that no current requirement needs.

Composition and inheritance

Two common ways to share behavior between classes are inheritance and composition. Although both can help us avoid duplication, they create different relationships between types. Understanding how each mechanism works will make it easier to see why one may be a better fit than the other.

Understanding both mechanisms

Inheritance is a mechanism that lets a class take on the **properties** and behavior of another class, with the ability to override some of them. It helps when we need several things that share most of their behavior and differ only in places. We create two types derived from the same base type, and each of them redefines only the part that should be different. The shared behavior is written once, in the base class.

Composition is a different approach. Instead of deriving from a base class, a class is given objects that provide the functionality it needs. It can serve a similar purpose to **inheritance**. When two functionalities are mostly the same but differ in one place, we can write one class that handles the shared part and hand it an object that handles the part that varies. Changing the behavior then means passing in a different object implementing the same **interface**.

Either mechanism gets us to a similar result, so which one to choose? *The composition over inheritance principle says composition is usually the better choice*, and the reason is that **inheritance** brings a lot of design problems with it. Let's understand them.

Issues of inheritance

To understand the issues **inheritance** brings, let's consider the following class:

```
class PersonalDataFormatter
{
    public string Format()
    {
        var people = ReadPeople();
        return string.Join(Environment.NewLine,
            people.Select(p => $"{p.Name} born in" +
                $" {p.Country} on {p.YearOfBirth}"));
    }

    private IEnumerable<Person> ReadPeople()
    {
        //Database-sourced implementation
    }
}
```

This class reads people's data from a database and formats it as a single string. It clearly breaks the Single Responsibility Principle, but let's set that aside for now.

One day the business requirements change. Sometimes the data will be read from the database, and sometimes from an Excel file, and the user decides which one at runtime.

There are two ways to approach this change: inheritance or composition. Let's start with inheritance, by making the `PersonalDataFormatter` class abstract. We will also make the `ReadPeople` method abstract, as it will be overridden in inheritors.

```
abstract class PersonalDataFormatter
{
    public string Format()
```

```

    {
        var people = ReadPeople();
        return string.Join(Environment.NewLine,
            people.Select(p => $"{p.Name} born in" +
                $" {p.Country} on {p.YearOfBirth}"));
    }

    public abstract IEnumerable<Person> ReadPeople();
}

```

We can now create two new classes derived from the `PersonalDataFormatter`. Each will override the `ReadPeople` method in its own way.

```

class DatabaseSourcedPersonalDataFormatter : PersonalDataFormatter
{
    public override IEnumerable<Person> ReadPeople()
    {
        //Database-sourced implementation
    }
}

class ExcelSourcedPersonalDataFormatter : PersonalDataFormatter
{
    public override IEnumerable<Person> ReadPeople()
    {
        //Excel-sourced implementation
    }
}

```

Now, switching between the data sources is just a matter of creating a proper `DataFormatter` type:

```

bool shallReadFromDatabase = false;
PersonalDataFormatter personalDataFormatter = shallReadFromDatabase ?
    new DatabaseSourcedPersonalDataFormatter() :
    new ExcelSourcedPersonalDataFormatter();
Console.WriteLine(personalDataFormatter.Format());

```

We got what we wanted. Personal data can now be formatted whether it comes from a database or from an Excel file, and there is no duplication, since the shared part lives in the base class.

Let's use this example to look at the problems inheritance brings.

First, `PersonalDataFormatter` is now tightly coupled to the classes that derive from it. Any change to the base class affects them. Adding a constructor to the base class, for instance, means adding the same constructor to every derived class.

The `ReadPeople` method is also stuck where it is. If we needed it elsewhere, we would have to reach for one of the `PersonalDataFormatter` classes, which makes little sense when what we need is reading data, not formatting it.

The relationship between these three classes is rigid. It was fixed at compile time, the moment we declared that the concrete classes derive from `PersonalDataFormatter`. A more flexible arrangement, one we could change without hardcoding it into the type declarations, would serve us better.

There is more. In C# and many other languages, a class can only inherit from one base class. There are good reasons for that restriction, but it is a restriction all the same. Derived types also tend to inherit more than they need, since the base class exposes its implementation details to everything below it.

Inheritance also makes class hierarchies grow fast. Suppose these classes need to differ not only in where the data comes from but also in how it is formatted, so that we can pick a short or a full format independently of the Excel or database source. The hierarchy jumps from three classes to five:

- `PersonalDataFormatter`
- `DatabaseSourcedPersonalDataShortFormatter`
- `DatabaseSourcedPersonalDataFullFormatter`
- `ExcelSourcedPersonalDataShortFormatter`
- `ExcelSourcedPersonalDataFullFormatter`.

Every new aspect that varies multiplies the hierarchy again, and it becomes unmanageable quickly.

Unit testing is the next problem. We have one abstract class and two concrete inheritors, and abstract classes cannot be instantiated, so there is no way to create a `PersonalDataFormatter` object for a test. That leaves two approaches, both flawed.

The first is to test both inheritors and accept that each test fixture also covers the shared logic from the base class. The tests end up partly duplicated, and a change in the base class means adjusting two fixtures.

The second is to test the inheritors while avoiding the base class logic as far as possible, then test that logic separately through a concrete type created only for testing. This is worse. Those tests exercise a type that does not exist in the application, so they prove less than they appear to. And even with the best intentions, it is hard to avoid touching the base class while testing its inheritors, so a change there is still likely to mean adjusting three fixtures.

One more reason, unrelated to the classes we have here but worth mentioning: inheritance is awkward with Object-Relational Mapping tools such as Entity Framework. These tools map objects to relational databases, letting us read records as objects and save objects back, and generating the **SQL** underneath. The tables typically mirror the types we defined in code. Databases do not have a natural equivalent of inheritance, so mapping a C# hierarchy onto flat tables is tricky and tends to produce a more complicated schema than we want.

All those issues can be solved by using composition instead of inheritance.

Composition over inheritance

Now let's refactor this code to use **composition** instead of **inheritance**.

The change is simple. Instead of an abstract method, `PersonalDataFormatter` takes a dependency on an object that can read the data. That object implements an `IPersonalDataReader` interface, and the class holds it in a field.

```
interface IPersonalDataReader
{
    IEnumerable<Person> ReadPeople();
}

class PersonalDataFormatter
{
    private readonly IPersonalDataReader _personalDataReader;

    public PersonalDataFormatter(IPersonalDataReader personalDataReader)
    {
        _personalDataReader = personalDataReader;
    }

    public string Format()
    {
```

```

        var people = _personalDataReader.ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
                $"{p.Country} on {p.YearOfBirth}"));
    }
}

class DatabaseSourcedPersonalDataReader : IPersonalDataReader
{
    public IEnumerable<Person> ReadPeople()
    {
        //Database-sourced implementation
    }
}

class ExcelSourcedPersonalDataReader : IPersonalDataReader
{
    public IEnumerable<Person> ReadPeople()
    {
        //Excel-sourced implementation
    }
}

```

This is what **composition** is. We put `PersonalDataFormatter` and `IPersonalDataReader` together, and working as a pair they give us exactly what we need. There is no **code duplication**, and the code can be extended easily.

Let's go through the problems inheritance caused, one by one, and see how composition deals with them.

Tight coupling. A minor change in the base class could force changes in the derived ones, which made the code brittle. Now `PersonalDataFormatter` and the reader classes know nothing about each other. They communicate through an interface, and interfaces change far less often than implementation details do.

Reusability. The people-reading logic used to live in an abstract method and its overrides, so reusing it meant dragging along a `PersonalDataFormatter`. The reader classes can now be composed with anything, including types that have nothing to do with formatting.

Rigid relationships. The link between the formatter and its subclasses was fixed at compile time. With composition, the formatter receives a reader through its constructor, so the choice

of reader can be made at runtime. It can depend on configuration or user input, and the formatter's code doesn't change.

One base type. Classes implementing `IPersonalDataReader` can implement as many other **interfaces** as they need.

Inheriting too much. With inheritance gone, the only point of contact between these classes is the interface, so there are no implementation details to leak.

Growing hierarchy. Adding short and long formatting used to mean multiplying the number of classes. Now it means one more interface with two implementations. However many traits we add, the structure stays flat: a set of interfaces with types implementing them, and nothing deeper than one level.

Unit testing. Testing `PersonalDataFormatter` is a matter of **mocking** the reader, and the reader classes can be tested on their own. A change in any of them affects one test fixture.

Object-Relational Mapping. With composition, one object holds another inside it, and that is something databases handle without trouble.

Every problem we identified with inheritance is gone. So whenever inheritance is tempting, the question worth asking is whether composition would serve better. Most of the time it will.

None of this means inheritance should never be used. **Frameworks** often require it, and deriving from a base class is simply how we tell the framework what kind of object we are building.

Custom exceptions are another case, since a new exception type is defined by inheriting from an existing one, and the resulting hierarchy is what lets callers catch errors at the level of detail they need. Inheritance also works well for small, data-focused families of types where the variants are known upfront and unlikely to grow. What it does not do well is serve as a general tool for sharing behavior, which composition handles better.

Refactoring case study: Reducing coupling in VacationPlanner

Before we start changing the code, let's first understand what it does and where its current design creates problems. This one brings together everything from this chapter: **coupling**, **cohesion**, **DRY**, and how a class gets hold of its **dependencies**.

Understanding the code

We will work with a `VacationPlanner`. Given a number of vacation days and a year, it looks for the start date that produces the longest continuous break, by combining those days with the weekends and public holidays around them.

Using it is straightforward:

```
VacationPlanner planner = new VacationPlanner();
int vacationDaysToUse = 5;
var optimalVacation = planner.FindOptimalVacation(
    vacationDaysToUse, 2027);
planner.ShowReport(optimalVacation, vacationDaysToUse);
```

Let's see the FindOptimalVacation method:

```
public VacationPlan FindOptimalVacation(int vacationDaysToUse, int year)
{
    var startDate = new DateTime(year, 1, 1);
    var endDate = new DateTime(year, 12, 31);
    DateTime bestStartDate = startDate;
    int maxHolidaysLength = 0;
    var publicHolidays = new PolishPublicHolidaysProvider().GetFor(year);

    for (var date = startDate; date <= endDate; date = date.AddDays(1))
    {
        int vacationLength = CalculateHolidayLength(date, vacationDaysToUse,
publicHolidays);
        if (vacationLength > maxHolidaysLength)
        {
            bestStartDate = date;
            maxHolidaysLength = vacationLength;
        }
    }
    return new VacationPlan(bestStartDate, maxHolidaysLength);
}
```

It takes the number of vacation days and the year. The algorithm is simple: for each day in the year, it calculates how long a vacation we would get by starting on that day, and it keeps the longest option found so far. Public holidays are taken into account: the method gets the list for the given year and passes it to CalculateHolidayLength. The result is a VacationPlan object holding the start date and the total number of days.

It uses a PolishPublicHolidaysProvider, a class that produces the list of public holidays in Poland for a given year. Notice how tightly the two classes are coupled. The provider is not injected either. It is created directly inside the method, so this class depends on a concrete type rather than an abstraction, which breaks the **Dependency Inversion Principle**. And if we

wanted to plan a vacation in any country other than Poland, there would be no way to do it without editing this code.

```
public class PolishPublicHolidaysProvider
{
    public List<DateTime> GetFor(int year)
    {
        return new List<DateTime>
        {
            new DateTime(year, 1, 1),    // New Year's Day
            GetEasterSunday(year).AddDays(1),
            //...produce all other public holidays dates
        };
    }
}
```

Now the CalculateHolidayLength method.

```
private int CalculateHolidayLength(
    DateTime vacationStart,
    int vacationDays,
    List<DateTime> publicHolidays)
{
    int usedVacationDays = 0;
    int totalVacationLength = 0;
    DateTime day = vacationStart;

    while (usedVacationDays < vacationDays)
    {
        if (day.DayOfWeek == DayOfWeek.Saturday || day.DayOfWeek ==
            DayOfWeek.Sunday ||
            publicHolidays.Contains(day))
        {
            totalVacationLength++;
        }
        else
        {
            totalVacationLength++;
            usedVacationDays++;
        }
        day = day.AddDays(1);
    }
}
```

```

    }

    // Keep adding weekend/public holiday days after vacation ends
    while (day.DayOfWeek == DayOfWeek.Saturday || day.DayOfWeek ==
DayOfWeek.Sunday ||
        publicHolidays.Contains(day))
    {
        totalVacationLength++;
        day = day.AddDays(1);
    }
    return totalVacationLength;
}

```

Starting from a given date, it keeps adding days to the vacation for as long as there are vacation days left. Weekends and public holidays do not consume any of them. And once the vacation days run out, the method still adds any weekend and holiday days that follow.

We also have the `ShowReport` method, which simply prints the information about the generated vacation plan.

```

public void ShowReport(VacationPlan vacationPlan, int vacationDaysToUse)
{
    Console.WriteLine(
        $"Vacation days used: {vacationDaysToUse}\n" +
        $"Optimal start date: {vacationPlan.Start.ToShortDateString()}\n" +
        $"Optimal end date: {vacationPlan.Start.AddDays(vacationPlan.TotalDays -
1).ToShortDateString()}\n" +
        $"Total vacation days: {vacationPlan.TotalDays}\n");
}

```

`VacationPlan` is a simple data structure:

```

public class VacationPlan
{
    public DateTime Start { get; }
    public int TotalDays { get; }

    public VacationPlan(DateTime start, int totalDays)
    {
        Start = start;
        TotalDays = totalDays;
    }
}

```

```
}  
}
```

Identifying the issues

The first thing to notice is that all those classes live in a single file.

Next, the `PolishPublicHolidaysProvider` class. On its own it is fine, but look at how it is used: no abstraction, no dependency inversion, no injection. `VacationPlanner` is tightly coupled to it. If we wanted an optimal vacation plan for a country with different public holidays or different weekend days, we would have a serious problem.

The formatting is messy, for example, some lines are too long to fit on the screen.

The order of methods is off as well. For example, `ShowReport` is public, yet it sits at the very bottom, below the private ones.

That same method breaks the Single Responsibility Principle. The `VacationPlanner` class is responsible for creating an optimal vacation plan, not for printing a report about the result.

It also hides another coupling and another **DIP** violation. The method calls a static method on the `Console` class, so the two are bound together. Showing the report in a graphical interface instead would mean rewriting all code related to data presentation.

The class breaks **DRY** too. Saturday and Sunday are named as weekend days in at least two places. That is a piece of **business logic** and it belongs in one place.

Finally, the naming is inconsistent. The code uses "holidays" and "vacation" interchangeably.

Let's start dealing with all those issues now.

Improving the code

The first step should be to reformat the code with the help of an IDE, break lines that are too long, adjust spacing between methods, and make other cosmetic changes.

We should settle on one word for time off work, either vacation or holidays, and use it consistently. Vacation is the better choice here, since holiday already means something else in this code: a public holiday. That means renaming `maxHolidaysLength` to `maxVacationLength` and `CalculateHolidayLength` to `CalculateVacationLength`.

Let's also make sure this code does not break the Don't Repeat Yourself principle. Right now there are two places defining Saturday and Sunday as weekend days. Imagine we are told one day that this code will be used in a country where the weekend falls on Friday and Saturday.

We would have to remember to change more than one place. Better to define it once, by moving the condition into a method:

```
private bool IsWeekend(DateTime dateTime)
{
    return
        dateTime.DayOfWeek == DayOfWeek.Saturday ||
        dateTime.DayOfWeek == DayOfWeek.Sunday;
}
```

Notice the alignment. Putting the two conditions directly one above the other makes the comparison easy to see at a glance.

Then we can deal with the DIP violation by representing calendar knowledge behind an **interface**:

```
public interface IPublicHolidaysProvider
{
    IEnumerable<DateTime> GetFor(int year);
    bool IsWeekend(DateTime day);
}
```

In this project, it will be implemented by `PolishPublicHolidaysProvider`. Another country can provide a different implementation without changing `VacationPlanner`. It's also a good idea to change the return type from a `List` to the more generic `IEnumerable`. This way, any type implementing the `IPublicHolidaysProvider` will decide for itself what collection type it wants to return.

Additionally, in the original code, it was odd that `VacationPlanner` is the class that knows whether a given day falls on a weekend. That is not its job. `IsWeekend` fits much better in the classes implementing `IPublicHolidaysProvider`. A class that knows which days are public holidays in a given country should also know which days are the weekend there. So let's move `IsWeekend` to `PolishPublicHolidaysProvider` and add it to the interface.

```
public class PolishPublicHolidaysProvider : IPublicHolidaysProvider
{
    public bool IsWeekend(DateTime day)
    {
        return day.DayOfWeek == DayOfWeek.Saturday ||
            day.DayOfWeek == DayOfWeek.Sunday;
    }
}
```

```
public IEnumerable<DateTime> GetFor(int year)
{
    //produce holidays dates
}
}
```

VacationPlanner will now depend on the abstract IPublicHolidaysProvider, and a concrete implementation will be injected via the constructor. With the new design, the holiday provider could easily be replaced in tests or at runtime, and the planner does not need to know how the dates are calculated or stored.

```
public class VacationPlanner
{
    private IPublicHolidaysProvider _publicHolidaysProvider;

    public VacationPlanner(IPublicHolidaysProvider publicHolidaysProvider)
    {
        _publicHolidaysProvider = publicHolidaysProvider;
    }
    //...
}
```

There is one more dependency to address, and it sits in the ShowReport method. It uses the static Console class to print information on the screen. What if we wanted a graphical interface instead, or a **mock** for testing? Switching the way we communicate with the user would not be easy.

We can make it easy by hiding the static dependency behind an interface:

```
public interface IUserCommunication
{
    void ShowMessage(string message);
}

public class ConsoleUserCommunication : IUserCommunication
{
    public void ShowMessage(string message)
    {
        Console.WriteLine(message);
    }
}
```

```
}  
}
```

And then, we can use it as a dependency of a new class, whose only role will be to print the vacation plan:

```
public class VacationPlanPrinter  
{  
    private IUserCommunication _userCommunication;  
  
    public VacationPlanPrinter(IUserCommunication userCommunication)  
    {  
        _userCommunication = userCommunication;  
    }  
  
    public void ShowReport(VacationPlan vacationPlan, int vacationDaysToUse)  
    {  
        var startDate = vacationPlan.Start.ToShortDateString();  
        var endDate =  
            vacationPlan  
                .Start  
                .AddDays(vacationPlan.TotalDays - 1)  
                .ToShortDateString();  
        _userCommunication.ShowMessage(  
            $"Vacation days used: {vacationDaysToUse}\n" +  
            $"Optimal start date: {startDate}\n" +  
            $"Optimal end date: {endDate}\n" +  
            $"Total vacation days: {vacationPlan.TotalDays}\n");  
    }  
}
```

With this design, if we ever switch to something other than the console to communicate with the user, all we have to do is inject a different object into the `VacationPlanPrinter` class. Now the printer doesn't even know that something like the console exists.

Let's see the final code that runs the application:

```
var planner = new VacationPlanner(  
    new PolishPublicHolidaysProvider());  
var printer = new VacationPlanPrinter(  
    new ConsoleUserCommunication());
```

```
var optimalVacation = planner.FindOptimalVacation(5, 2025);  
printer.ShowReport(optimalVacation, 5);
```

We should also organize all the files in a reasonable folder structure:

- ✓   DataStructures
 - >  C# VacationPlan.cs
- ✓   PublicHolidays
 - >  C# IPublicHolidaysProvider.cs
 - >  C# PolishPublicHolidaysProvider.cs
- ✓   UserCommunication
 - >  C# ConsoleUserCommunication.cs
 - >  C# IUserCommunication.cs
 - >  C# VacationPlanPrinter.cs
- >  C# Program.cs
- >  C# VacationPlanner.cs

Figure 13.1: Suggested project structure for the VacationPlanner application

Summary

This chapter looked at the relationships that decide whether a set of classes ends up flexible or fragile. We went through reducing coupling, applying the **Law of Demeter**, and using cohesion to judge where the boundaries between classes should fall. We also treated **DRY** as a rule about knowledge rather than about identical lines of text. The inheritance and composition examples, along with the VacationPlanner case study, showed how abstractions help when they reflect variation that actually exists. In *Chapter 14*, you will turn from design structure to source-code commentary and decide when **comments** add information and when they merely compensate for code that should be clearer.

14

Using comments effectively

We sometimes hear that the more **comments** a codebase has, the better. It sounds reasonable. Comments explain, and explanation helps. But it does not hold up in practice.

A comment can help a reader understand something the code cannot express on its own. It can also become a second description of the program, running alongside the first one and agreeing with it only for a while. *A comment that restates obvious code adds noise. A comment that has fallen out of date is worse*, because it tells the reader something that is no longer true.

The practical approach is to let the code carry as much of the intent as it can, and to reserve comments for what truly belongs outside the implementation. This chapter looks at comment patterns that do harm, shows when **refactoring** or **version control** solves the problem better, and points out the cases where a short, precise comment earns its place.

In this chapter, we will cover the following topics:

- Why comments should not compensate for unclear code
- Avoiding outdated and misleading comments
- Removing comments that merely restate the code
- Replacing navigation and **block-description comments** with clearer structure
- Why change history and commented-out code do not belong in comments
- Documenting complex algorithms and regular expressions
- Explaining temporary workarounds
- Using TO-DO comments carefully
- Using XML documentation comments for shared APIs
- Keeping required copyright comments concise

By the end of this chapter, you will be able to distinguish comments that add information from comments that repeat or contradict the code, refactor code to remove unnecessary explanations, use version control instead of changelist or commented-out code, and write focused comments for algorithms, regular expressions, workarounds, TO-DOs, and public **API documentation**.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Why comments should not replace clear code

Comments are often added with good intentions, but they should not compensate for unclear code. *If a variable or method needs a comment merely to explain what it means, improve the name.* If a method is too complex to understand without a running commentary, refactor the method before adding more prose.

A high comment count is not a quality metric. The code and the comments must be maintained separately, so every explanation creates another artifact that can drift away from the behavior it describes. Prefer clear names and structure whenever they can carry the same information more reliably.

When your first response to confusing code is to add a comment, ask whether **refactoring** would remove the need for that comment. A good comment adds information that the code cannot express cleanly; it should not serve as a translation layer for poor names or tangled control flow.

Outdated and misleading comments

Comments become dangerous when they no longer match the implementation. An IDE can rename symbols and detect many syntax problems, but it cannot reliably know that a prose explanation has become false after a behavior change. Treat the code as the source of truth and keep comments only where their additional information justifies the maintenance cost.

A comment that merely duplicates the code adds little value, while a comment that contradicts the code can send a maintainer in the wrong direction. The safest comment is one that explains information beyond the obvious mechanics of the implementation.

The following quote summarizes this idea:

"Don't comment bad code, rewrite it."

— *Brian W. Kernighan and P. J. Plauger*

Several recurring comment patterns are useful warning signs because they either restate visible code or preserve information that belongs elsewhere.

Removing comments that restate the code

A **"Captain Obvious"** comment restates what the code already says. Consider the following code:

```
//constructor of the Person class
public Person(string name, int yearOfBirth)
{
    Name = name;
    YearOfBirth = yearOfBirth;
}

//save file name
string saveFileName = "game";

//calculates the distance between points in kilometers
double CalculateDistanceInKilometers(
    Point point1, Point point2)
{
    // method body
}
```

Those comments add no information, as the identifiers already express what the code does. Remove such comments and let the code speak for itself.

Removing navigation comments

Another category of useless comments is **navigation comments**. Consider this for loop:

```
for (int i = 0; i < iterations; i++)
{
```

```
decimal x = i;  
decimal y = x * i;  
z += y;  
} //end of for Loop
```

Navigation comments such as this one usually indicate that formatting or method size is making the structure hard to see. Align the code consistently and extract oversized blocks rather than marking their boundaries with prose.

Replacing block-description comments with methods

Block-description comments often name a responsibility that should itself become a method. Let's see some examples:

```
if(ex == "txt")  
{  
    //read the IDs from TXT file  
    var txt = File.ReadAllText(file);  
    var ids = txt.Split(',');  
    foreach(var fileId in ids)  
    {  
        numbers.Add(int.Parse(fileId));  
    }  
}  
else if(ex == "json")  
{  
    //read the IDs from JSON file  
    var txt = File.ReadAllText(file);  
    numbers = JsonSerializer.Deserialize<List<int>>(txt);  
}
```

Those comments show that the developer noticed the separate responsibilities in those blocks of code but stopped short of acting on it. A better approach is to extract the logic into **methods** with descriptive names, eliminating the need for comments:

```
var fileContent = File.ReadAllText(file);  
  
if (ex == "txt")  
{  
    numbers = ReadIdsFromText(fileContent);  
}  
else if (ex == "json")
```

```
{  
    numbers = ReadIdsFromJson(fileContent);  
}
```

There is another advantage to this. A method name travels with the method. Wherever it is called, the name is right there, explaining what happens at that point. A comment stays where someone wrote it, and it appears again only if someone remembers to write it again. That makes a good name the more reliable documentation of the two.

Keeping change history in version control

Changelist comments store information that modern version-control systems already track more accurately. Dates, authors, renames, and cleanup history belong in repository history and commit messages rather than at the top of a source file.

Here is an example of changelist comments at the top of a file:

```
//02.10.2025 - File created by Krystyna  
//03.10.2025 - moved to Utils folder  
//04.10.2025 - renamed to IdsReader  
//02.11.2025 - renamed to IdsFromFileReader by Thomas  
//03.01.2026 - CheckFileExistence method refactored and renamed  
//04.02.2026 - removing unused code + cleanup  
//06.06.2026 - making the Read method public instead of internal
```

Instead, use your **version control** system to track changes.

Removing commented-out code

Commented-out code creates uncertainty because readers cannot tell whether it is obsolete, temporarily disabled, or still required. Delete code that is no longer used and rely on version-control history if an earlier implementation ever needs to be recovered.

Now that you have seen the worst examples of comments, you are ready to learn about scenarios where using comments may actually be appropriate.

When comments add value

Comments are justified when they carry context that the code itself cannot express, such as the reason behind a workaround, the description of an algorithm being implemented, or what a regular expression is actually matching. When a comment is necessary, make it precise, complete, brief, and easy for another developer to maintain. Before writing it, confirm that the same intent cannot be expressed more reliably through a better name, smaller method, or clearer type.

Prefer code that explains its own mechanics, and use comments for the additional context that remains.

Documenting complex algorithms

For example, if you implement a complex algorithm like **adaptive heap sort**, even the best variable naming may not help a developer unfamiliar with the algorithm understand your code. In this case, it is helpful to leave a short comment with a link to the algorithm's description, such as a Wikipedia article. This allows anyone reading your code to quickly find out what algorithm is being used.

The following code demonstrates how you might document the use of a complex algorithm with a comment and a reference link:

```
// Implements the Adaptive Heap Sort algorithm
// https://en.wikipedia.org/wiki/Adaptive_heap_sort
public static void Sort(int[] input)
{
    int length = input.Length;

    // Build the initial max-heap
    for (int i = length / 2 - 1; i >= 0; i--)
    {
        Heapify(input, length, i);
    }

    for (int i = length - 1; i > 0; i--)
    {
        // Swap the root element (maximum) with the last element
        int temp = input[0];
        input[0] = input[i];
        input[i] = temp;
        // Heapify the reduced heap
        Heapify(input, i, 0);
    }
}
```

Remember, this does not excuse you from *keeping the code itself as clean as possible*.

Explaining regular expressions

Another scenario where comments are appropriate is when you use **regular expressions** (**regexes**). Regular expressions are patterns for matching and manipulating strings, and they

can be difficult to read and understand. You should not force readers of your code to decipher regexes on their own. Instead, leave a comment that describes what the regex does and provides examples of matching and non-matching strings.

Here is an example of how to document a regex with a descriptive comment and examples:

```
public static bool IsValidEmail(string email)
{
    // Simple email validation pattern
    // This pattern checks for valid characters,
    // the "@" symbol, domain names, and top-level domains (TLDs).
    //
    // Examples of matching strings:
    // somebody@gmail.com
    // someone@yahoo.fr
    //
    // Examples of non-matching strings:
    // somebody@gmail
    // @yahoo.fr
    string pattern = @"^[\w-]+(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]{2,7}$";
    return Regex.IsMatch(email, pattern);
}
```

Documenting temporary workarounds

Sometimes code has to include a workaround for a problem that cannot be fixed right away. One real example from the author's experience: a library that threw an exception the first time an object was instantiated, but worked on the second attempt. With a deadline approaching and no time to investigate properly, the object was created twice, and a comment was left explaining why.

The following code shows how to document such a workaround with a clear comment:

```
SomeClass someObject;
try
{
    someObject = new SomeClass(); // this will fail
}
catch (Exception)
{
    // This library always throws an exception when
    // SomeClass constructor is called for the first time.
}
```

```
// This is a temporary workaround - to be addressed.  
// (12-09-2025, contact John Smith in case of questions)  
}  
someObject = new SomeClass(); // this will succeed
```

Using TO-DO comments carefully

Another reasonable use of **comments** is for to-do notes that describe what still needs to be done in the code. These are acceptable as long as you follow a few important conditions.

Larger work items usually belong in a dedicated issue tracker rather than in source comments. Use TO-DO comments only for small, short-lived tasks; keep their format consistent, make ownership and date visible when that helps the team, and use analyzers or repository searches so TO-DOs do not disappear indefinitely.

The following code demonstrates a properly formatted to-do comment, including the responsible person and date:

```
//to-do refactor this class  
// (Krystyna, 03.09.2023)  
public class FileManager  
{  
}
```

In contrast, the next example shows inconsistent and less effective to-do comment formats:

```
//to-do refactor this class  
//TO DO: fix bug when Length < 0  
//To-do add error handling
```

A consistent format, as shown below, is preferred, as it will enable easy search by automated tools:

```
//to-do refactor this class  
//to-do fix bug when Length < 0  
//to-do add error handling
```

Using structured documentation comments

Most languages have a convention for documenting a public **API** in a structured form that tools can read: XML documentation comments in C#, Javadoc in Java, docstrings in Python. IDEs display them to whoever calls the method, so the documentation reaches the reader at the moment they need it.

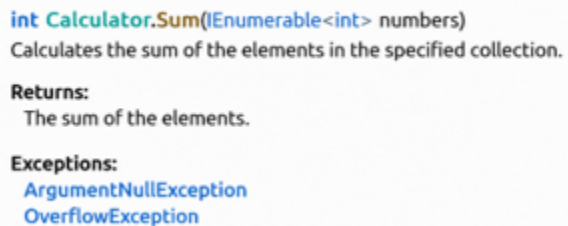
This kind of documentation has to be kept in step with the API it describes. That is worth doing where the audience justifies it, which usually means shared libraries and public interfaces. For code used only inside one project, the maintenance cost is rarely repaid.

The following code shows how to write summary comments in C#:

```
/// <summary>
/// Calculates the sum of the numbers in the specified collection.
/// </summary>
/// <param name="numbers">The numbers to sum.</param>
/// <returns>The sum of the elements.</returns>
/// <exception cref="ArgumentNullException">
/// Thrown if <paramref name="numbers"/> is <see langword="null"/>.
/// </exception>
/// <exception cref="OverflowException">
/// Thrown if the sum is greater than <see cref="int.MaxValue"/>
/// or less than <see cref="int.MinValue"/>.
/// </exception>
public static int Sum(IEnumerable<int> numbers)
{
    return numbers.Sum();
}
```

When a caller hovers over a documented method in the IDE, a popup with a summary along with parameter, return-value, and exception information will appear. This is useful when the documentation describes the contract rather than narrating the implementation.

```
var result = Calculator.Sum(new[] {1,2,3});
```

A Visual Studio tooltip for the `Calculator.Sum` method. The tooltip is a light blue box with a pointer to the `Sum` property access in the code above. It contains the following text:
int Calculator.Sum(IEnumerable<int> numbers)
Calculates the sum of the elements in the specified collection.
Returns:
The sum of the elements.
Exceptions:
[ArgumentNullException](#)
[OverflowException](#)

The tooltip uses bold text for the signature, return value, and exceptions, and blue text for the exception names which are likely hyperlinks in the IDE.

Figure 14.1: Visual Studio tooltip displaying method summary, return value, and exceptions for a documented method.

Keeping required copyright comments concise

Some organizations require copyright or licensing notices in source files. Keep them as concise as the policy allows and prefer a link to authoritative legal text when appropriate.

The recurring rule is simple: add a comment when it contributes information that the code, naming, structure, tooling, or version-control history cannot communicate more reliably.

Summary

This chapter set out a practical standard for comments: *the code stays the primary source of truth*, and a comment earns its place when it adds something that names, structure, or version control cannot carry more reliably. We went through the comments worth removing, from redundant restatements of the code to navigational markers, changelogs, and commented-out code. We also looked at the cases where a comment does help: complex algorithms, regular expressions, temporary workarounds, TO-DOs, documentation comments on public APIs, and copyright notices that a project is required to carry. In *Chapter 15*, you will bring the book together by designing production code for testability and maintaining tests with the same care as the code they protect.

15

Writing testable code and clean tests

Software can look clean and still be risky to change. A method may be short and a class well organized, yet a small modification can still introduce a defect that nobody notices until after release. Manual testing helps, but repeating the same checks after every refactoring gets slow, and people start skipping steps.

The practical answer is to make testability part of the design. **Automated tests** give us a safety net that we can run as often as we like, mocks let a class be exercised without real infrastructure behind it, and **dependency injection** creates the seams that make both possible.

Test code needs the same care as any other code. A confusing or brittle test suite becomes something the team maintains rather than something that helps them, and eventually something they stop trusting.

In this chapter, we will cover the following topics:

- Understanding unit-testing fundamentals and isolated tests
- Using mocks to control dependencies
- Using tests as a safety net for refactoring and as design feedback
- Applying test-driven development
- Treating test code as production code
- Writing readable, focused, and maintainable tests
- Avoiding testability problems caused by static dependencies and concrete types
- Using dependency injection effectively

- Keeping constructors simple and free of hidden work
- Refactoring a class and its tests into a clean, consistent suite

By the end of this chapter, you will be able to design code that can be tested in isolation, use mocks and dependency **injection** to control collaborators, write readable and focused unit tests, recognize common testability pitfalls, and refactor both production code and test code without losing the safety net that supports continuous improvement.

Technical requirements

You can follow the discussion and code examples in this chapter without running them. If you want to compile, execute, or refactor the examples yourself, use the following setup:

- .NET 10 with C# 14. The examples use standard C# constructs and are also compatible with .NET 8 and later versions.
- An IDE with C# tooling, such as Visual Studio Community 2026 or Visual Studio Code.
- Basic familiarity with C# syntax, classes, methods, collections, and Boolean expressions, as well as fundamental knowledge of object-oriented programming.
- The example-code repository for the book hosted at <https://github.com/KrystynaSlusarczykLearning/PragmaticCleanCode>.

Understanding unit testing

A **unit test** is an automated check of one small piece of functionality. In practice that usually means calling one method on one class with a fixed set of inputs and verifying the result.

"Automated" means we do not run them by hand. The whole suite should run from a button or a single command, and it should finish fast, within seconds for a small project. That is not something we get for free. It is something the tests have to be written and kept fit for, because a suite that takes ten minutes is a suite people stop running.

The same goes for isolation. A test for one class should only fail when that class changes. Modifying *class A* and watching the tests for *class B* go red is a sign that the two are more entangled than they should be, and it is worth stopping to look at the design rather than at the test. Isolation is something we design for, mostly by keeping **dependencies** explicit and substituting them in tests.

Let's consider this simple mathematical example:

```
class Math
{
    public static int Multiply(int a, int b)
```

```
    {
        return a * b;
    }

    public static int Add(int a, int b)
    {
        return a + b;
    }
}
```

Here is what a sample test for the Add method could look like. When we run the tests for the whole project, the framework finds them on its own and executes them, then reports which ones passed and which did not.

```
[TestFixture]
class MathTests
{
    [Test]
    public void Add_ShallGive_8_ForArguments_3_And_5()
    {
        var sum = Math.Add(3, 5);
        Assert.That(sum, Is.EqualTo(8));
    }
}
```

One test does not prove that the whole Math class is correct. Other inputs and other behaviors need tests of their own. What this example does show is the shape every **unit test** takes: arrange the data the test needs, execute one behavior, and verify the result, all without anyone clicking anything.

There are many testing frameworks, and the examples here use NUnit. The details differ between them, but the structure above is the same everywhere, so switching from one framework to another is mostly a matter of learning new names for the same things.

Characteristic	What it means	Why it matters
Automated	The test can be discovered and run by a framework without manual steps	The same checks can be repeated after every code change

Characteristic	What it means	Why it matters
Focused	The test verifies one small piece of behavior	A failure is easier to diagnose and understand
Fast	The test completes quickly enough to run frequently	Developers can use the test suite as continuous feedback
Isolated	The result is controlled by the code under test rather than external systems	Failures are predictable and are less likely to be caused by unrelated components
Repeatable	The same inputs produce the same expected outcome	The test remains dependable across machines and repeated runs

Table 15.1: Characteristics of effective unit tests

Using mocks for isolated tests

Isolation becomes more interesting when the class under test depends on another component. Consider `PersonalDataProcessor`, which asks a repository for people and returns the greatest age:

```
class PersonalDataProcessor
{
    private readonly IPeopleRepository _peopleRepository;

    public PersonalDataProcessor(IPeopleRepository peopleRepository)
    {
        _peopleRepository = peopleRepository;
    }

    public int FindMaxAge()
    {
        var allPeople = _peopleRepository.GetAll();
        return allPeople.Max(person => person.Age);
    }
}
```

A unit test should not need a real database holding a particular set of records. That would make the test slower, harder to run on another machine, and dependent on what happens to

be in the database. It would also mean the test exercises the repository as well as `PersonalDataProcessor`.

This is what *mocks* are for. A mock is an object that replaces a real dependency in a test. We decide how it behaves, so the test doesn't depend on the real implementation. It has no real logic behind it. We simply state what its methods should return when the code under test calls them. Most languages have mocking libraries that create these classes for us, so we rarely write them by hand.

Here the mock stands in for the repository, and the test decides exactly what `GetAll` returns:

```
[TestFixture]
public class PersonalDataProcessorTests
{
    [Test]
    public void FindMaxAge_ReturnsTheAgeOfOldestPerson()
    {
        var peopleRepositoryMock = new Mock<IPeopleRepository>();
        peopleRepositoryMock
            .Setup(repo => repo.GetAll())
            .Returns(new List<Person>
            {
                new Person("John", "Doe", 25),
                new Person("Jane", "Doe", 80),
                new Person("Max", "Smith", 30)
            });
        var processor = new PersonalDataProcessor(
            peopleRepositoryMock.Object);
        var result = processor.FindMaxAge();
        Assert.That(result, Is.EqualTo(80));
    }
}
```

The test is now independent of the real repository and its database. Mocks are also useful when the collaborator performs an action rather than returning a value. For example, a mock user-communication object can verify that `ShowMessage` was called with the expected argument even when the method under test has no useful return value.

What unit tests do and do not prove

Automated **unit tests** are the first line of defense after a change. When the covered behaviors still pass, we know those pieces work as they did before, and we know it in seconds rather than after repeating the same checks by hand.

A good test suite also works as documentation. Opening the tests for a class is often the fastest way to understand what it does, since each test states one behavior in isolation, along with the inputs it applies to and the result it produces. And unlike a comment or a wiki page, this description cannot quietly fall out of date. If the class stops behaving the way a test says it does, the test fails.

Unit tests do not prove the whole application works. Because unit tests deliberately isolate classes from one another, they say nothing about how those classes behave together. That is what integration and end-to-end tests are for. The role of a unit test suite is narrower: fast feedback about small pieces of behavior, and a way of making design problems visible whenever isolation turns out to be difficult.

Much of what this chapter says about writing tests well applies to those other kinds too. A clear name, one reason to fail, no logic in the test body, no dependence on the order tests run in: none of that is specific to unit tests. An integration test that nobody can read is as much of a liability as an unreadable unit test, and probably more, since it usually takes longer to run.

Using tests to protect and improve code quality

Unit tests do not make production code clean by themselves. A bad design can be covered by a large test suite and stay exactly as bad as it was. What tests do is make changing that code safer, so the design can be improved without fear. And when writing them turns out to be hard, they point at the parts of the design that resist isolation, which is usually where the improving needs to start.

Using tests as a safety net for refactoring

Getting from working code to clean code rarely happens in one go. First, we make the code work. Then we improve it through a series of small refactoring steps: renaming a variable, extracting a method, moving a responsibility to another class. Each step is easy on its own. What stops people from taking these steps is the risk. Every one of them changes code that currently works, and nobody wants to be the one who broke it.

This is where tests work as a safety net. After each step, we run them and know within seconds whether the code still behaves as before. If something broke, the fix is easy, because only one small thing changed.

Without tests, teams tend to leave working code alone, and it keeps getting worse because nobody dares to touch it. With tests, refactoring becomes an ordinary part of the day rather than something to schedule and worry about.

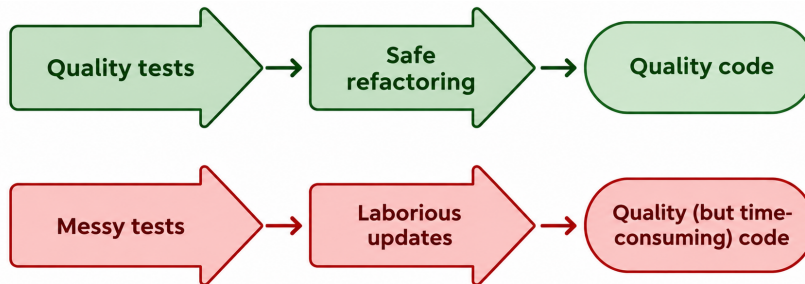


Figure 15.1: How test quality influences refactoring and the resulting code quality

The figure captures an important feedback loop. Clean tests support safe refactoring, and safe refactoring supports clean production code. Messy tests make changes expensive, while having no tests at all makes developers much less likely to refactor.

Using testability as design feedback

A class can look perfectly clean, with good names and a tidy structure, and still have a design that causes trouble. Usually we find out much later, when a new requirement arrives and the change turns out to be painful. But there is an earlier warning, and it comes when we sit down to write a test.

To test a class in isolation, its design has to include seams: boundaries along which the class can be cut out of the rest of the project. Without them the test drags half the application in with it, and whatever we end up writing is not really a unit test, because its result no longer depends on the class alone.

So the difficulty of writing a test is information in itself, and it arrives before the test has run even once. A class creating its own dependencies is the classic case. There is no way to test it without engaging everything it builds inside itself, and that struggle points straight at what needs to change.

That is not a coincidence, and it is the more interesting point. A class that a test can separate from the rest of the code is a class that anything can separate from the rest of the code. It can be swapped for a different implementation when the requirements change, even at runtime.

Interfaces and **constructor injection** create these seams, and a mock passed into a constructor in a test is the same mechanism as a different implementation passed in for real.

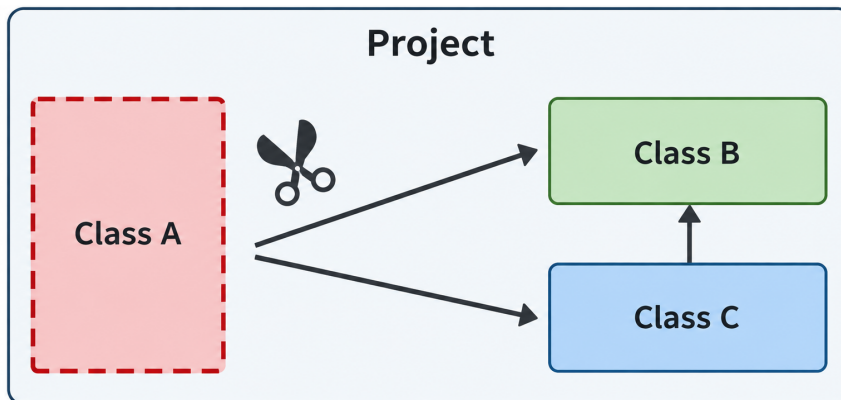


Figure 15.2: A seam separates Class A from collaborators that would otherwise be exercised with it

Once the dependency boundary is explicit, the class can be moved conceptually into a test context while the real collaborators are replaced or ignored:

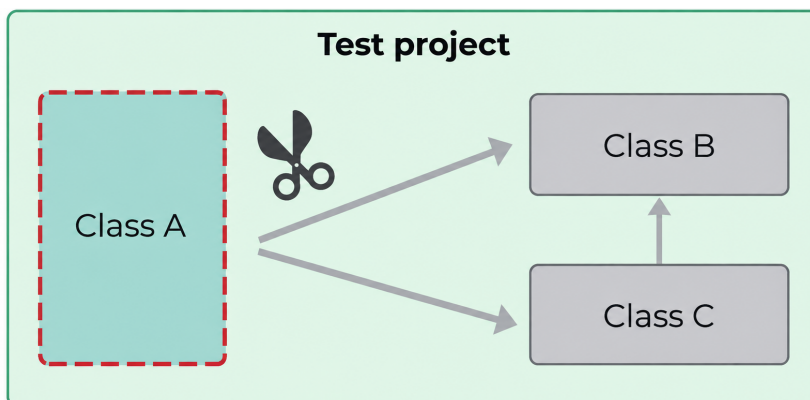


Figure 15.3: Class A tested in isolation while its production collaborators are excluded

This is why the act of writing tests can improve a design even before the test suite provides long-term **regression protection**. *Difficulty in testing is often a signal that dependencies are hidden, concrete, or too tightly connected.*

Using test-driven development

Test-driven development (TDD) changes the order in which code is written. Instead of implementing a whole feature and then adding tests, we start with a test for one small behavior that does not exist yet. Then we write just enough production code to make that test pass, improve the result while keeping it passing, and move on to the next behavior.

Say we are writing a `Divide` method. The first test states that dividing 8 by 4 gives 2, and we write the code that makes it true. The next test requires an exception when the divisor is zero, and only then does that case get implemented. Each requirement shows up as a test before it shows up as code.

This connects directly to clean code design. Writing the test first means describing how the class will be used before deciding how it works inside, so the signature, the parameter order, the return value, and the dependencies *are all settled from the caller's point of view*. And if the class needs data from a database, the question of how a test would control that data comes up immediately, which pushes us toward an abstraction and an injected dependency without having to think about it separately.

None of this happens because tests magically produce good code. It happens because the feedback arrives constantly. Every behavior gets a check as soon as it exists, and every implementation step ends with the safety net already in place, which is exactly the moment when improving the design costs the least.

Treating test code as production code

A test suite is part of the codebase and changes alongside the production code. If the tests are difficult to read or modify, developers spend more time maintaining them, and eventually they may start disabling tests that fail too often. Once that happens, the safety net loses its value. Test code therefore deserves the same attention to naming, structure, duplication, and readability as production code.

Writing readable and well-named tests

A test should say what it checks at a glance. Short tests do that. Long ones make the reader work for it.

One way to keep them short is to move repeated setup out of the test bodies. Most frameworks let us mark a method that runs automatically before each test in a class, and that is where the objects and configurations the tests share can be created:

```
[TestFixture]
class MathTests
{
    [SetUp]
    public void Setup()
    {
        // Code here runs before every test.
    }
}
```

```
    }  
}
```

Test names are especially important because the name is often the first information you see when a test fails. A useful name communicates what is being tested, the expected behavior, and the scenario. The exact order of those parts is a team convention; consistency matters more than choosing one universal pattern. Long names are acceptable when they make the failed behavior immediately understandable.

For example, a test checking that dividing 8 by 4 gives 2 could be called `Divide_ShallGive_2_WhenDividing_8_By_4`. The name has three parts: the method being tested, the expected result, and the scenario it applies to.

Notice the underscores. They go against how methods are normally named, and that is fine here. Test names are read in a report rather than called from code, so separating the parts makes them easier to pick out at a glance.

Separating arrange, act, and assert

Most **unit tests** contain three conceptual steps. Arrange prepares objects and data, Act executes the behavior under test, and Assert verifies the outcome. Keeping those steps visually distinct makes the test easier to follow. Empty lines are often enough; explicit **Arrange**, **Act**, and **Assert** comments are optional and should be used consistently if a team prefers them.

```
[Test]  
public void FindMaxAge_ReturnsTheAgeOfOldestPerson()  
{  
    var peopleRepositoryMock = new Mock<IPeopleRepository>();  
    peopleRepositoryMock.Setup(repo => repo.GetAll()).Returns(  
        new List<Person>  
        {  
            new Person("John", "Doe", 25),  
            new Person("Jane", "Doe", 80),  
            new Person("Max", "Smith", 30)  
        });  
    var processor = new PersonalDataProcessor(  
        peopleRepositoryMock.Object);  
    var result = processor.FindMaxAge();  
    Assert.That(result, Is.EqualTo(80));  
}
```

Keeping tests focused

Each test should verify one coherent behavior. A test that creates a collection, reverses it, orders it, filters it, and then checks a final value does not make it clear which operation failed. Split unrelated behaviors into separate tests so that a failure points to one responsibility.

Loops, conditionals, and exception-handling logic inside tests are warning signs because they introduce additional behavior into the test itself. A small loop can occasionally be reasonable, for example when making the same assertion for every item in a collection, but complicated control flow usually means the test should be simplified or divided.

Removing duplication with parameterized tests

Say we write a test checking that dividing 8 by 4 gives 2. Then we want the same check for another pair of numbers. The tempting move is to copy the test and change the values, which leaves two methods with identical bodies and different arguments. It works, but every fix to that test now has to be made twice.

Testing frameworks solve this with parameterized tests. We write the test body once, with the values as parameters, and then declare the sets of values it should run with. The framework runs the method once per set and reports each run separately, so a failure still tells us which case broke. Each framework has its own syntax for this, and NUnit uses `TestCase` attributes:

```
[TestCase(8, 4, 2)]
[TestCase(20, 5, 4)]
public void Divide_ShallGiveCorrectResult_ForValidInput(
    int dividend, int divisor, float expectedResult)
{
    var result = Math.Divide(dividend, divisor);
    Assert.That(result, Is.EqualTo(expectedResult));
}
```

Writing meaningful assertions

Assertions determine whether the test passes or fails, so they should verify the facts that matter to the behavior under test. As a practical default, keep a test centered on one logical outcome. Multiple assertions are reasonable when they describe one tightly connected result, such as the size and contents of a collection.

```
[Test]
public void Reverse_ShallFlipTheCollectionOrder()
{
    var input = new List<int> { 1, 2, 3 };
}
```

```

    input.Reverse();
    Assert.That(input.Count, Is.EqualTo(3));
    Assert.That(input[0], Is.EqualTo(3));
    Assert.That(input[1], Is.EqualTo(2));
    Assert.That(input[2], Is.EqualTo(1));
}

```

Do not assert details that the tested behavior does not promise. If a method is responsible only for selecting a day in a given year, asserting unrelated time components makes the test more brittle without increasing confidence in the intended behavior.

Practice	Prefer	Avoid
Length	Short tests with reusable setup where appropriate	Large tests that require scrolling through extensive setup
Naming	Descriptive names that state behavior and scenario	Generic names such as <code>Test1</code> or <code>BubbleSortTest</code>
Structure	Clear Arrange, Act, Assert phases	Interleaving setup, execution, and assertions
Responsibility	One coherent behavior per test	Combining several unrelated operations in one test
Duplication	Parameterized test cases or shared helpers	Copying nearly identical tests for different values
Assertions	Only facts relevant to the tested behavior	Checking incidental details that make the test brittle

Table 15.2: Practices for clean and maintainable tests

Designing production code for testability

Testability is a property of the production code. When a class is hard to test, the fix belongs there rather than in the test. Let's look at the design problems that cause it most often.

Avoiding static dependencies that must be replaced

Chapter 10 examined static methods and hidden dependency graphs in detail. Here, the focus is narrower: what happens to a unit test when the behavior it needs to control is locked behind a static call.

A class that receives an `IPeopleRepository` can be tested with a mock. If the same class calls a static repository method directly, the test loses that substitution point:

```
class PersonalDataProcessor_Static
{
    public int FindMaxAge()
    {
        var allPeople = PeopleRepository.GetAll(); //we can't substitute this in tests
        return allPeople.Max(person => person.Age);
    }
}
```

This does not make every static call a problem. Plenty of them are fine, because there is nothing we would ever want to substitute. A mathematical operation returns the same answer for the same arguments and has no alternative implementation. The same goes for simple text processing, such as trimming whitespace or changing case. Calls like these are fast, predictable, and have no reason to be replaced in a test, so letting the real code run is exactly right.

The problem appears when a static call reaches something we would want to control: a database, a file, the network, or the system clock.

Depending on abstractions rather than concrete types

Chapter 9 introduced the Dependency Inversion Principle as a design rule. In a test, the same abstraction becomes a substitution seam: the production collaborator can be replaced with a controlled test double.

The class below depends on `PeopleRepository`, a concrete type. Nothing about that signature invites a substitute, so a test for this class runs against a real repository, and through it against a real database. That makes the test slow, dependent on what the database happens to contain, and no longer a test of this class alone. Depending on an `IPeopleRepository` interface instead

would remove the problem, since the test could then pass in a mock and decide for itself what the repository returns.

```
class PersonalDataProcessor
{
    private readonly PeopleRepository _peopleRepository;

    public PersonalDataProcessor(PeopleRepository peopleRepository)
    {
        _peopleRepository = peopleRepository;
    }
}
```

Change the dependency to an abstraction when the collaborator is intended to vary or be replaced in tests:

```
class PersonalDataProcessor
{
    private readonly IPeopleRepository _peopleRepository;

    public PersonalDataProcessor(IPeopleRepository peopleRepository)
    {
        _peopleRepository = peopleRepository;
    }
}
```

The production class now states the capability it requires without committing to one implementation. The test can supply a mock, while the application can supply the real repository.

Some mocking libraries can fake a concrete class by generating a subclass at runtime, so it might look as though the interface is unnecessary. In practice, it only works for members the language allows to be overridden. For members the language does not allow to be overridden, the behavior depends on the mocking library. Some refuse outright, others accept the setup and never use it, leaving the real method to run while the test appears to be working with a mock. Mocking a class also ties the test to one particular implementation rather than to a contract, which is how a change inside one class ends up breaking the tests of another. Interfaces avoid both problems, and that is why mocking against them is the default approach.

Injecting dependencies instead of creating them internally

An interface is not enough if the class constructs the concrete implementation itself. The following constructor still prevents a test from supplying a mock:

```
class PersonalDataProcessor
{
    private readonly IPeopleRepository _peopleRepository;

    public PersonalDataProcessor()
    {
        _peopleRepository = new PeopleRepository();
    }
}
```

Constructor injection exposes the dependency and ensures that every valid instance receives it when the object is created. Making dependency fields publicly settable would also permit replacement, but it weakens encapsulation and allows the dependency to be changed to an invalid value later. Constructor injection keeps the object valid after construction.

Keeping constructors simple

One of the most common testability pitfalls is a constructor that does actual work. Creating an object should be cheap and predictable, and once it stops being either, every test that needs that object pays the price.

Here is a method that updates the owner of a house in a repository. It looks straightforward, and in production it works. Now we want a unit test for the `UpdateHouseOwner` method. The test would be simple enough: call the method and verify that `Save` was called on the repository mock with the right arguments.

```
public class HousesStateUpdater
{
    private readonly IHousesRepository _housesRepository;

    public HousesStateUpdater(IHousesRepository housesRepository)
    {
        _housesRepository = housesRepository;
    }

    public void UpdateHouseOwner(int houseId, Owner owner)
    {

```



```
        var house = new House(houseId, owner);
        _housesRepository.Save(house);
    }
}
```

But as it turns out, the test fails with some complicated message about a failed database connection, which makes no sense at first. The test uses a mock of the houses repository, so nothing here should be connecting to anything.

Tracking it down leads to the line where a `House` is constructed. The constructor turns out to do more than construct. It creates a `HousesRepository` object, which opens a database connection, which in turn expects global configuration that does not exist in the test environment. Hence the exception.

```
public class House
{
    public int HouseId { get; }
    public Owner Owner { get; }

    public House(int houseId, Owner owner)
    {
        var housesRepository = new HousesRepository();
        if (!housesRepository.DoesHouseExist(houseId))
        {
            throw new ArgumentException(
                $"A house with ID {houseId} does not exist.");
        }
        if (!housesRepository.DoesOwnerExist(owner))
        {
            throw new ArgumentException(
                $"An owner named {owner.Name} does not exist.");
        }
        HouseId = houseId;
        Owner = owner;
    }
}
```

A constructor doing this kind of work is wrong on several levels.

The first is expectation. Seeing a constructor call, nobody expects a database connection or a dependency on global configuration. Work like that should be visible, not hidden behind what looks like an ordinary constructor call.

The second is performance. Imagine a loop creating 1000 House objects. The constructor runs on every pass, and every time it builds a repository and opens a connection.

The third is that constructors run in places we do not always notice. Some types create objects on our behalf rather than when we ask for them. For example, C# has a type called `Lazy<T>`, a wrapper that holds an object of type `T` but does not create it until something first uses it. At that point the constructor of `T` runs, at a place in the program where no new appears anywhere nearby. If that constructor opens a database connection, both the cost and any resulting test failure are hard to trace back to their source.

The fourth brings us back to tests. A base class constructor also runs when we create an object of a derived type. This matters when we mock a concrete class, which is possible, although not recommended. A common way for mocking libraries to do this is to create a subclass of that class. In that case, creating the mock runs the base constructor. If that constructor does real work, such as opening a database connection, the mock does it too, and the test suddenly needs a working database. All we wanted was a simple stand-in. The same problem appears wherever derived types are involved, but tests are where it hurts most. It is one more reason to mock interfaces rather than classes.

A constructor doing complex work causes problems that are hard to trace, because they start at the moment an object is created and nobody thinks to look there.

So what belongs in a constructor? Very little. Assigning the arguments to fields, creating simple structures the object owns, such as an empty list, and rejecting arguments that are plainly invalid, like a `null` dependency. The goal is a valid object and nothing more.

What does not belong there is anything with a cost or a consequence. No database connections, no file reads, no calls to web services. No loops or branching business logic. And nothing expensive enough that we would want to see it happening in the code rather than have it triggered by a `new`. If an operation is worth thinking about when it runs, it should be a method call somebody makes deliberately, not a side effect of creating an object.

Pitfall	Why it hurts tests	Preferred response
Static dependency	The collaborator cannot be replaced through normal interface-based mocking	Use an injectable abstraction when the dependency must vary or be controlled

Pitfall	Why it hurts tests	Preferred response
Concrete dependency	The class is tied to one implementation and may be difficult to substitute	Depend on an interface or another suitable abstraction
Dependency created internally	The test cannot provide a mock even if an interface exists	Inject the dependency, normally through the constructor
Complex constructor	Object creation hides I/O, heavy work, or side effects	Keep constructors lightweight and move real work into explicit methods or collaborators

Table 15.3: Common testability pitfalls and their remedies

Refactoring case study: Cleaning Sorter and its tests

This last case study pulls together two threads from the book: comments are not a way to make up for code that does not explain itself, and test code deserves the same care as the code it covers.

Understanding the code

Before we start refactoring, let's look at the Sorter class and the tests around it to understand what the code currently does. We start with a Sorter class. It implements bubble sort, and it implements it correctly.

```
public class Sorter
{
    // impl of the bubble sort algorithm
    // O(n*n) complexity
    // Bubble sort repeatedly compares adjacent elements
    // and swaps them when needed.
    // implemented on 12.12.2024
    // modified on 20.12.2024 to not modify the input List

    public static List<int> BubbleSort(List<int> inputList)
    {
        // check if input is null
        ArgumentNullException.ThrowIfNull(inputList);
    }
}
```

```

List<int> sortedList = new List<int>(inputList);
int n = sortedList.Count; // the count of items
for (int i = 0; i < n - 1; i++)
{
    for (int j = 0; j < n - i - 1; j++)
    {
        if (sortedList[j] > sortedList[j + 1])
        {
            // swap if needed
            int temp = sortedList[j];
            sortedList[j] = sortedList[j + 1];
            sortedList[j + 1] = temp;
        }
    } // end of inner loop
} // end of outer loop
return sortedList;
}
//old version that modifies the input list
//public static List<int> BubbleSort(List<int> list)
//{
//    int n = list.Count;
//    the rest of the implementation
//}
}

```

This class comes with a suite of **unit tests**:

```

[TestFixture]
public class SorterTests
{
    [Test]
    public void BubbleSortTest()
    {
        // Arrange
        List<int> unsortedList1 = new List<int> { 4, 2, 3, 1 };
        List<int> expectedSortedList = new List<int> { 1, 2, 3, 4 };
        // Act
        List<int> resultList1 = Sorter.BubbleSort(unsortedList1);
        // Assert
        Assert.That(resultList1, Is.EqualTo(expectedSortedList));
        // Arrange
    }
}

```

```
List<int> unsortedList2 = new List<int> { 3, 4, 2, 1 };
// Act
List<int> resultList2 = Sorter.BubbleSort(unsortedList2);
// Assert
Assert.That(resultList2, Is.EqualTo(expectedSortedList));
// Arrange
List<int> unsortedList3 = new List<int> { 1, 2, 3, 4 };
// Act
List<int> resultList3 = Sorter.BubbleSort(unsortedList3);
// Assert
Assert.That(resultList3, Is.EqualTo(expectedSortedList));
}

[Test]
public void EmptyList_ReturnsEmptyList()
{
    List<int> emptyList = new List<int>();
    List<int> resultList = Sorter.BubbleSort(emptyList);
    Assert.That(resultList.Count, Is.EqualTo(0));
}

[Test]
public void ShallReturnSameList_ForOneItemList()
{
    // Arrange
    List<int> singleElementList = new List<int> { 42 };
    List<int> expectedSortedList = new List<int> { 42 };
    // Act
    List<int> resultList = Sorter.BubbleSort(singleElementList);
    Assert.That(resultList, Is.EqualTo(expectedSortedList));
}

[Test]
public void BubbleSort_ShallProduceTheSameList_FromAlreadySortedInput()
{
    // Arrange
    List<int> sortedList = new List<int> { 1, 2, 3, 4, 5 };
    List<int> expectedSortedList = new List<int> { 1, 2, 3, 4, 5 };
    // Act
    List<int> resultList = Sorter.BubbleSort(sortedList);
    // Assert
```

```
        Assert.That(resultList, Is.EqualTo(expectedSortedList));
    }

    [Test]
    public void BubbleSort_ListWithDuplicates_SortsCorrectly()
    {
        List<int> listWithDuplicates = new List<int> { 3, 5, 2, 2, 5, 1, 3 };
        List<int> expectedSortedList = new List<int> { 1, 2, 2, 3, 3, 5, 5 };
        List<int> resultList = Sorter.BubbleSort(listWithDuplicates);
        Assert.That(resultList, Is.EqualTo(expectedSortedList));
    }
}
```

Those tests cover many important cases: a few unsorted lists, an empty list, a list with one item, a list that is already sorted, and a list with duplicates. Each test builds an input, calls `BubbleSort`, and compares the result against the list it expects.

Identifying the issues

First, let's discuss the implementation. The problem is everything around the algorithm: comments restating what the code already says, a change history that belongs in version control, markers announcing where loops end, and old code left commented out in case someone wants it back.

Now, the tests. The suite works, but there is plenty to fix.

`BubbleSortTest` does too much. It packs three separate cases into one method, each with its own arrange, act, and assert block. When it fails, the name tells us nothing about which case broke, and neither does the method itself until we count the assertions.

One of those three cases is redundant. The third input in `BubbleSortTest` is already sorted, and `BubbleSort_ShallProduceTheSameList_FromAlreadySortedInput` covers exactly that.

The remaining two cases duplicate their code. The same three lines appear twice with different numbers, which is exactly what parameterized test cases are for.

The names follow no pattern. `BubbleSortTest` says nothing at all.

`EmptyList_ReturnsEmptyList` and `ShallReturnSameList_ForOneItemList` are informative but built differently from each other, and `BubbleSort_ListWithDuplicates_SortsCorrectly` uses a third arrangement again. Only three of the five start with the name of the method under test.

The arrange-act-assert comments add length without adding clarity. They are also applied inconsistently, with `EmptyList_ReturnsEmptyList` having none and `ShallReturnSameList_ForOneItemList` missing the one before the assertion.

Two behaviors are not tested at all. Nothing checks that a null input throws, and nothing checks that the input list comes back unchanged, which matters here because an earlier version of the method sorted the list in place.

Improving the code

Let's start with the algorithm. Nearly all of these comments should go. The code already shows that the input is checked for null, that `Count` gives the number of items, where each loop ends, and how the swap works. Change history belongs in version control, and so does the old implementation left commented out, which can be recovered from there if anyone ever needs it.

One comment is worth adding, though. Readers who want to understand the algorithm itself are better served by a single line linking to its description on Wikipedia than by a long explanation copied into the file.

With the comments gone, the method is easier to scan, because what remains on the screen is the code that actually runs:

```
public class Sorter
{
    //https://en.wikipedia.org/wiki/Bubble_sort
    public static List<int> BubbleSort(List<int> inputList)
    {
        ArgumentNullException.ThrowIfNull(inputList);
        var sortedList = new List<int>(inputList);
        int itemCount = sortedList.Count;
        for (int i = 0; i < itemCount - 1; i++)
        {
            for (int j = 0; j < itemCount - i - 1; j++)
            {
                if (sortedList[j] > sortedList[j + 1])
                {
                    int temp = sortedList[j];
                    sortedList[j] = sortedList[j + 1];
                    sortedList[j + 1] = temp;
                }
            }
        }
    }
}
```

```
    }  
    return sortedList;  
}  
}
```

Refactoring the overgrown test

Now, let's deal with the tests.

BubbleSortTest is the one that needs the most work. Three arrange-act-assert sequences in a single method, significant duplication, and a name that says nothing about what is being checked.

One of those three cases can go, since another test already covers a sorted input. The other two do the same thing with different numbers, which is exactly what a parameterized test is for. Here it is, with a name that says what the test expects:

```
[TestCase(new[] { 4, 2, 3, 1 }, new[] { 1, 2, 3, 4 })]  
[TestCase(new[] { 3, 4, 2, 1 }, new[] { 1, 2, 3, 4 })]  
public void BubbleSort_ShallProduceListSortedAscendingly_FromUnsortedInput(  
    int[] unsortedCollection, int[] expectedSortedCollection)  
{  
    List<int> resultList = Sorter.BubbleSort(unsortedCollection.ToList());  
    Assert.That(resultList, Is.EqualTo(expectedSortedCollection));  
}
```

The arrange-act-assert comments can go as well. They make each test a few lines longer without making anything clearer, since the structure is visible from the code itself: the variables are set up, the method is called, the result is checked. In a test this short, labeling those three steps adds nothing.

Next, the tests for the two special cases: an empty input list and a list with a single item. Neither is badly written, but their names can be better, and the comments can go the same way as before:

```
[Test]  
public void BubbleSort_ShallProduceEmptyList_FromEmptyInput()  
{  
    List<int> emptyList = new List<int>();  
    List<int> resultList = Sorter.BubbleSort(emptyList);  
    Assert.That(resultList.Count, Is.EqualTo(0));  
}
```



```
[Test]
public void BubbleSort_ShallProduceTheSameList_FromSingleItemInput()
{
    List<int> singleElementList = new List<int> { 42 };
    List<int> expectedSortedList = new List<int> { 42 };
    List<int> resultList = Sorter.BubbleSort(singleElementList);
    Assert.That(resultList, Is.EqualTo(expectedSortedList));
}
```

The test for input that is already sorted has a good name, so removing the comments is all it needs. The last one, covering input with duplicates, only needs a name that matches the rest:

```
BubbleSort_ShallProduceListSortedAscendingly_FromInputWithDuplicates
```

Adding missing behavior checks

Refactoring the existing tests is not enough if important branches remain uncovered. BubbleSort explicitly rejects null input, so the suite should verify that behavior:

```
[Test]
public void BubbleSort_ShallThrowException_ForNullInput()
{
    Assert.That(() => Sorter.BubbleSort(null!), Throws.ArgumentNullException);
}
```

The method is also intended to return a new sorted list without modifying the original input. That behavior deserves a dedicated regression test because an earlier implementation changed the input collection:

```
[Test]
public void BubbleSort_ShallNotModifyTheInputList()
{
    var inputList = new List<int> { 3, 2, 1 };
    Sorter.BubbleSort(inputList);
    Assert.That(
        inputList,
        Is.EqualTo(new List<int> { 3, 2, 1 }));
}
```

With the changes in place, the suite needs to be run, so we can be sure everything works as expected.

With the tests passing, both halves are done. The `Sorter` class holds the algorithm and nothing else. The comments restating the code are gone, along with the change history and the old implementation, and the one comment that remains points to a description of bubble sort for anyone who wants it.

The tests are in better shape too. Every test covers one case, every name follows the same pattern and says what the test expects, and two behaviors that were never checked before now have tests of their own. The suite also works as documentation of the class, describing exactly how it behaves. Most importantly, if we ever want to refactor this class, we can now do it safely. Any change that breaks existing behavior will make a test fail.

Problem	Refactoring action	Result
Comments restate visible code	Delete Captain Obvious and loop-navigation comments	The implementation is easier to scan
Change history is stored in comments	Rely on version control instead	Source files contain current behavior rather than historical clutter
Old implementation remains commented out	Delete it and use repository history for recovery	No uncertainty about whether dead code is still relevant
One test contains several scenarios	Use focused tests or parameterized test cases	Failures identify a specific behavior and duplication is reduced
Test names are generic or inconsistent	Apply one descriptive naming convention	The test explorer communicates failures more clearly
Important branches are untested	Add tests for null input and preservation of the original list	The suite protects behavior that could otherwise regress

Table 15.4: Refactoring actions in the `Sorter` case study

Summary

This chapter connected testability with clean design and clean tests. We went through what makes a unit test focused and isolated, how mocks let us control what a class collaborates with, and why a fast suite is what makes refactoring safe. We also saw that difficulty in writing a test is information in itself, pointing at hidden dependencies and missing boundaries. Along the way we covered test-driven development, readable test structure, parameterized tests, meaningful assertions, the trouble with static and concrete dependencies, and constructors that do work nobody asked for.

The Sorter case study brought all of it together, cleaning up both the production code and the tests: better names, better structure, no unnecessary comments, repeated scenarios parameterized, and the missing cases finally covered. That result is what this book has been aiming at throughout: *code that is easier to understand, safer to change, and backed by feedback that keeps the next improvement affordable.*

16

Unlock your exclusive benefits

Your copy of this book includes the following exclusive benefits:



DRM-Free PDF Version

Download DRM-free PDF and ePub copies of this book.



7-Day Packt Library Access

Get 7-day unlimited access to 8,000+ books and videos. No credit card required.

Available for first-time Packt+ trial users only.



Next-Gen Reader Access

Read this book on Packt Reader with progress sync, dark mode and note-taking.

Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this book's free benefits in 3 easy steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

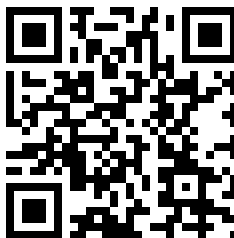
For more help on finding your invoice, visit <https://www.packtpub.com/en-us/unlock?step=1>.

Note

Note: If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

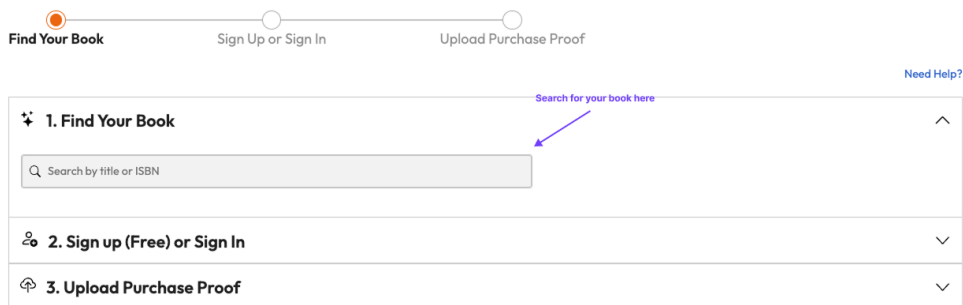
Scan the QR code or go to [packtpub.com/unlock](https://www.packtpub.com/unlock).



On the page that opens (similar to *Figure 16.1* on desktop), search for this book by name and select the correct edition.

Unlock Your Book's Free Benefits

Bought a Packt book from Amazon or one of our channel partners? Unlock your free benefits in 3 easy steps.



The image shows a desktop view of the Packt unlock landing page. At the top, there's a progress bar with three steps: 'Find Your Book' (active, orange circle), 'Sign Up or Sign In' (inactive, grey circle), and 'Upload Purchase Proof' (inactive, grey circle). Below the progress bar, there's a search bar with the placeholder text 'Search by title or ISBN'. A purple arrow points to the search bar with the text 'Search for your book here'. To the right of the search bar, there's a link that says 'Need Help?'. Below the search bar, there are three steps listed: '1. Find Your Book' (with a star icon), '2. Sign up (Free) or Sign In' (with a person icon), and '3. Upload Purchase Proof' (with a document icon). Each step has a chevron icon to its right.

Figure 16.1: Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need help

If you get stuck and need help, visit <https://www.packtpub.com/unlock-benefits/help> for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.



Note

Note: If you are still facing issues, reach out to customer@packt.com.



packtpub.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.

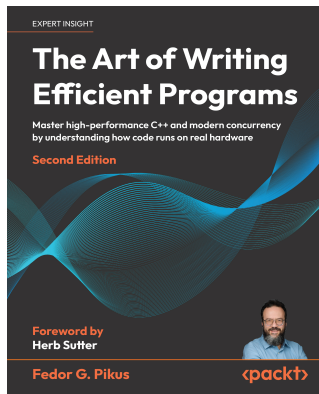
Why subscribe?

- 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

At www.packtpub.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:

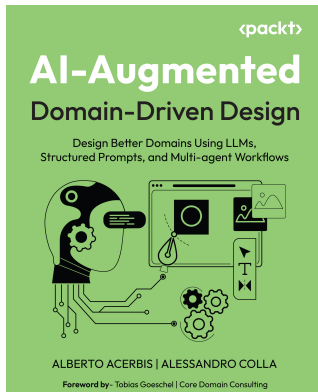


The Art of Writing Efficient Programs

Fedor G. Pikus

ISBN: 978-1-83546-518-9

- Analyze and optimize CPU resource usage
- Improve memory access patterns across architectures
- Design performant multithreaded and lock-free systems
- Use atomic operations and smart pointers effectively
- Apply memory order guarantees for correctness and speed
- Evaluate compiler decisions and leverage vectorization
- Implement coroutine-based execution models
- Build APIs that scale in concurrent environments



AI-Augmented Domain-Driven Design

Alberto Acerbis, Alessandro Colla

ISBN: 978-1-80742-105-2

- Understand how LLMs reason, where they help, and where they fail
- Apply prompt engineering to Domain-Driven Design workflows
- Use AI-assisted reasoning to refine domain events and bounded contexts
- Build rulebooks that guide and constrain LLM reasoning
- Apply AI to EventStorming and domain modeling activities
- Design AI agents for domain discovery and DDD workflows
- Orchestrate multi-agent systems for domain modeling
- Strengthen software architecture decisions while preserving domain clarity

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packt.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 *Pragmatic Clean Code*, we'd love to hear your thoughts! Scan the QR code below 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.



<https://packt.link/r/1808493796>

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

Index

A

AI coding assistants	13
abbreviations	
using	37
abstract method	3
abstract types	106, 107
abstraction	
concrete delivery service,	141 – 143
replacing with	
consistent levels,	60, 61
maintaining	
file access, isolating behind	155 – 157
keeping independent	143, 144
levels of abstraction,	187
separating	
account data	
personal data, separating	185, 186
from	
adaptive heap sort	240
antipattern	179
appropriate name length	
selecting	28, 29
arrange, act, and assert	254
assertions	255
automated formatting	79 – 81
tools	

B

Big Design Upfront (BDUF)	219
Boolean parameters	53, 54
Boolean values	
naming	21
Boolean-returning methods	
naming	21
Boy Scout rule	25
base class	179
billing behavior	

extracting, from customer data	186, 187
--------------------------------	----------

block-description comments	
replacing, with methods	238, 239
braces	82
bug	3
business rules	215 – 217

C

changelist comments	239
class members	
ordering	200, 201
class names	182
compound names	182
incomplete names	183
vague names	183, 184
class structure	
refactoring	205
classes	178
consistency, maintaining	203, 204
helper classes	181
large classes	179, 180
organizing, into separate files	201
clean code	2, 9, 181, 253
characteristics	9, 10
in age of AI-assisted	13, 14
development	
significance	4
versus poorly written code	10 – 12
clear method signatures	
designing	49, 50
client	132
code blocks	
structuring	82
code duplication	216
design decisions, deferring	220
justified	217, 218
premature abstraction,	219
avoiding	

code formatting	76	Dependency Inversion Principle	96, 140, 144, 145, 159, 164, 195, 257
code quality	8	Don't Repeat Yourself (DRY) principle	215, 216
code repository	4	delegates	49
code review	3	dependencies	
cohesion	212	creating	146 – 148
commented-out code		explicit	173, 174
removing	239	dependency graph	169 – 171
comments	236	derived class	179
benefits	239	E	
block-description	238, 239	effective communication	8
comments, replacing with		exception	3
methods		expressive names	
changelist comments	239	writing	27, 28
complex algorithms,	240	F	
documenting		Format Document command	79
misleading	236, 237	fields	179, 200
navigation comments,	237, 238	financial debt	7
removing		folders	
outdated	236, 237	structuring	202
regular expressions	240, 241	formatting	76 – 78
removing	237	style	78, 79
required copyright	244	function	2, 48
comments		G	
structured documentation	242, 243	global state	172
comments		god class	179
temporary workarounds,	241	good code	6
documenting		granularity	189
TO-DO comments	242	grouping behavior	186
composition	221	H	
over inheritance	224 – 226	Hangman game, formatting	
concrete data-provider dependency		refactoring case study	86 – 92
refactoring	148 – 150	Hungarian notation	
concrete delivery service		avoiding	36
replacing, with abstraction	141 – 143	helper classes	181
confusingly similar names		hidden dependencies	
avoiding	37	fragile code	171, 172
constructor injection	173, 251		
constructors	200		
context			
using	38		
coupling	208 – 211		
curse of knowledge	24		
D			
Definition of Done	6		
Dependency Injection	144, 145		

-
- high cohesion** 212 – 214
 - homonyms** 32
 - I**
 - Integrated Development Environment (IDE)** 4
 - Interface Segregation Principle** 96, 126, 132 – 137, 159
 - identifiers**
 - language, selecting for 35
 - implementation detail** 200
 - implementer** 132
 - indentation** 81
 - inheritance** 3, 179, 220
 - issues 221 – 224
 - interface** 106, 107, 251
 - interface pollution** 137
 - isolated tests**
 - mocks, using for 248, 249
 - K**
 - Keep It Simple, Stupid (KISS)** 219
 - L**
 - Law of Demeter** 211
 - Liskov Substitution Principle** 96, 120 – 127, 134 – 136, 159
 - lambda expressions** 3, 49
 - language**
 - selecting, for identifiers 35
 - large classes** 179, 180
 - acceptable 180, 181
 - large language models** 14
 - library** 4
 - long lines**
 - breaking 82 – 84
 - long test names** 29
 - low cohesion** 212
 - M**
 - meaningful names**
 - significance 18 – 20
 - meaningless words**
 - avoiding 35
 - method** 2, 48, 178
 - consistent levels of 60 – 62
 - abstraction
 - delegates 49
 - focused 57
 - independent 63
 - lambda expressions 49
 - naming 21, 49
 - ordering 62
 - responsibility 58, 59
 - signatures 49, 50
 - small 57
 - splitting, to reduce 51
 - parameters
 - task 57
 - method chains** 83
 - method design**
 - refactoring case study 66 – 73
 - misleading comments** 236, 237
 - mocks** 3
 - using, for isolated tests 248, 249
 - N**
 - naming** 181
 - appropriate name length 28, 29
 - Boolean values 21
 - Boolean-returning 21
 - methods
 - difficulty 26, 27
 - expressive names 27, 28
 - fundamentals 20
 - language conventions 22, 23
 - long test names 29
 - meaningful names, 18 – 20
 - significance
 - methods 21
 - parameters 21
 - project conventions 22, 23
 - reasons, for selecting poor names 24, 25
 - refactoring case study 39 – 44
 - review workflow 44
 - singular and plural forms, matching 34
 - types 21
 - variables 21
 - natural language**

using	34	professional responsibility	8
navigation comments		project files	
removing	237, 238	structuring	202
O		project structure	
Open-Closed Principle	96, 106, 107, 116, 125, 159	refactoring	205
account behavior,	110 – 112	properties	179
refactoring		public APIs	
practical limits	112 – 115	renaming	25, 26
violation	108, 109	public members	178, 200
object	178	public static methods	163 – 167
outdated comments	236, 237	pure functions	63 – 65
overly specific names		R	
avoiding	35	readability	9
P		readable tests	253
PizzaGenerator		refactoring	2, 181
refactoring case study	190 – 197	case studies	15
parameter lists	83	regular expressions (regexes)	187, 240
parameter-count anti-patterns		renaming	25
avoiding	55, 56	public APIs	25, 26
parameterized tests	255	report behavior	
parameters		moving, to objects owning data	157
Boolean parameters	53, 54	required copyright comments	244
naming	21	risk	8
ordering, logically	56	rubber duck method	
reducing	51	using	24
related parameters,	52, 53	S	
grouping		SOLID principles	96
selecting	50, 51	Dependency Inversion Principle	96, 159
personal data		Interface Segregation Principle	96, 132 – 137, 159
separating, from account data	185, 186	Liskov Substitution Principle	96, 120 – 128, 159
plural forms	34	Open-Closed Principle	96, 106 – 116, 159
polymorphism	106, 107	Single Responsibility Principle	96 – 103, 116, 159
poor code quality		SOLID problems	
consequences	5	in employee-report design	151 – 154
poorly written code			
versus clean code	10 – 12		
principle of least surprise	30, 31		
private members	178		
private methods	163		
production code			
designing, for testability	257 – 261		

Single Responsibility Principle	96, 97, 116, 136, 159, 180, 215	tight coupling	
applying	97–103	warning signs	211
checklist	103	type-based names	
Sorter class		avoiding	36
refactoring case study	262–269	types	
scope	82	naming	21
semantic correctness		U	
maintaining	31, 32	unit test	3, 246, 250
singular forms	34	characteristics	247, 248
static framework APIs	167, 168	unnecessary class proliferation	
static method	3, 162	avoiding	189
string-cleaning details		V	
moving, out of blog-post	187, 188	VacationPlanner	
storage		refactoring case study	226–233
structured documentation comments	242, 243	variables	
synonyms	33	naming	21
T		version control system	3
TO-DO comments	242	vertical space	
technical debt	6	code, organizing	84, 85
test code		virtual method	3
as production code	253–255	void functions	2
test-driven development (TDD)	252	W	
testability	256	well-named tests	253
using	251	whitespace characters	82
tests		working code	6
focused	255	Y	
using	250	You Ain't Gonna Need It (YAGNI)	219