Contents

Preface

Code refactoring directly improves your codebase and reduces your costs.

A gap exists in the practicality of code refactoring.

Do a Google search for the phrase "code refactoring." Go on and do it right now. Notice how many resources are available to you about this topic. Notice how people explain what it is. Do you understand what they are saying?

There is a gap between the notion of refactoring and the ability to put it into action. Many people struggle with making the leap from theory to implementation.

Our problem is getting started.

I wrote this book and its code challenge workbook to help you bridge the gap. You will learn about code refactoring and then I will give you tweaks that you can do right now. This book will teach you:

- How to spot a refactoring opportunity.
- How to think about it.
- And then how to do it.

The gap is knowing the practicality of refactoring and the ability to put it into action. Our problem is getting started.

This book helps you to bridge the gap.

I'll cover the most common opportunities as these are easy wins that you can do right now.

Listen to me. When you do these refactoring tweaks, here is what you get:

- Your codebase and product(s) will improve.
- You will reduce the lifecycle costs.
- You will be more profitable, as lower costs increase the bottom line.

Let's start cleaning up your code.

Who Should Read This Book?

This book is for anyone who wants to build better code.

Do you build software solutions? Then code refactoring is a necessary skill that propels you forward. There is a significant difference between code that works right now and quality code that you can maintain and reuse for years.

Regardless of your title or years of experience, refactoring is a continuous learning process.

If you build solutions in code, then this book and the code challenges workbook are for you.

How to Read This Book

I intentionally laid the book out in order to incrementally progress you forward from where you are right now to doing. Start at the introduction and read all of it.

The Secrets of Refactoring

This section opens your mind to the practice of refactoring. You need to know what the heck it is and it what it does for you.

Why Should You Care

Why should you care about refactoring? Why should you spend even a second on improving your code? What do you get out it? Refactoring takes work, as you have to think about it, make sure you don't break anything or change behavior, and then go do it. Why should you take your valuable time to do this process? What's in it for you?

Refactoring Clues

Clues guide you. They are your bloodhound sniffing out problematic areas in your codebase. You get started by using these clues to locate an area to improve.

Refactoring Tweaks

Now we get to the fun part: the actual refactoring tweaks. Each chapter presents one tweak. I will explain what it is, why it's problematic, and how to spot it. Then I'll give you an actionable step-by-step process to implement.

Have your code open on your computer while you read this section. It will help you to connect what I'm showing you to your own code.

Call-out Key

Throughout the book, I provide asides and call-outs to share additional information or to emphasize an important point. This key will help you to quickly identify the intent.

Master Tip

The bullseye is a master tip or insight that I'm sharing with you. Each of these will give more of *the why* to further solidify your understanding and implementation.

Additional Information

The coffee cup provides you with additional information, such as a definition or clarification. For example, if I'm teaching you about the context of post type, I may explain why it exists and how it's used to classify content.

Question or Thought Experiment

The question circle challenges you to think and consider *why*. These are meant to inspire you to rethink how you do stuff.

Doing It Wrong

The rocket ship plummeting towards the ground means "Whoopsie, you're doing it wrong". At times, you may think of going in a particular direction, but that direction has problems. This call-out helps you see *why* that direction will cause you woes.

Credit Time

The ringing bell is to give credit to someone who has contributed to this book, the way I think, or this profession.

Code Examples

Throughout the book, code examples are provided. Refactoring is language and platform independent. In other words, these strategies you will learn work for PHP, JavaScript, Perl, Python, C#, Visual Basic, etc. Refactoring is a process. It doesn't care what high level language you use to express your code.

The code examples are real code snippets taken from various codebases. To keep it concise, a `..` is used as a placeholder for code that has been removed for brevity.

Terminology

WordPress has two different meanings for the term `post`:

- All content that is stored in the database table `wp_posts` is considered a `post`.
- Each `post` gets a more descriptive classification called a post type. Post types can be post, page, revision, attachment, navigation menu, or custom post type.

Ah confusing, I know. To avoid confusion, let's change the first definition and term it `content`. Therefore, the `content` is a record out of the `wp_posts` database table and can be any post type.

The Secrets of Refactoring

There is a difference between code that works and quality code that is readable, reusable, and maintainable.

It Works

Do you want to step out and maintain it?

Quality

Reusable and maintainable

We build solutions for our clients. We pour ourselves into the details, designing code that works. The deeper we go into our code, the more *we focus on getting it to work*.

What we create in code does what it's supposed to do, with the needed functionality and cool features. But at what cost?

Writing code is expensive. It takes time to craft a solution. Time costs money for everyone involved.

Your time is valuable and in limited supply. Working smarter allows you to focus and have more time for the fun stuff, like the custom bits that are specific to this project.

Refactoring is a process to help you work smarter.

Working smarter means you are thinking about how to build a codebase that you can reuse over and over again with minimal effort. It's figuring out how to get the code to tell you what is going on so you can find errors and make changes faster.

Wait, stop. Think about what I just said for a moment. Imagine code that tells you what's going on. You'll bang your head on your desk less often and keep more of your hair. Imagine starting a project where the bulk of the code is already done and tested. That's the power of reusable code. Think about what you could do with all that extra time.

Let's say WordPress - or your favorite plugin - releases an update which requires you to change your code. It's probably been months since you looked at that project. See if you can relate to this struggle.

You open up the files and it takes you a while to find the right function or area. Then, you have to figure out what's going on and why. You'll read the code and shake your head, wondering what the heck you were thinking. Even worse, you're afraid to touch it as it might break or get worse.

Getting code to "just work" is like building the less desirable bridge. It's a bridge, but will it support you regularly stepping out there to fix a board in the middle?

Refactoring can turn the whole bridge into a well-designed, sound structure that you are proud of and can safely use again and again. When a board needs to be repaired or upgraded, you feel confident to step out onto that bridge and make it better.

What is Refactoring?

Refactoring is the process of rewriting your code to make *more* readable, reusable, and maintainable. It is taking the working code and incrementally improving it to reduce your costs over the life-cycle of the code.

Additional Information

Refactoring does not change the behavior of your code. It merely improves it. Make sure you understand the difference. Your code is working now. You are just cleaning it up and making it better.

I wrote this book to give you tools to make your code better. It starts with this understanding: all code needs improvement. Yours. Mine. Everyone's code. Period.

Why Should You Care?

Code refactoring improves the quality of your code. Quality code directly impacts your bottom line.

Refactoring reduces your costs. It helps you to *change working code* into *quality, cost-effective working code*, without changing the behavior.

You are a software professional. You call yourself a programmer, web developer, engineer, or architect. You solve business problems using software technologies. You are a professional and this is how you make your living.

Refactoring changes working code into quality, cost-effective working code, without changing the behavior.

Let's explore why you should care.

Professional Swagger

You pour yourself into every project you build. The code is a reflection of you, your talents, and your creativity.

No one wants someone to frown upon their work or discount it because it fails to comply with software principles or standards. We all want our craftsmanship to be highly regarded. You want your code to perform and be of the highest quality. You want bragging rights and the professional swagger to say "Hey, I built that. It's awesome!"

This is our art and profession.

Refactoring improves your code and earns you professional swagger.

Cost Effective

Being a professional means you get paid to do the thing you love: using your expertise to provide unique solutions.

What makes you "in demand" and marketable? Delivering high-quality, cost-effective solutions that deliver real return-on-investment (ROI).

Hum, think about what I just said.

The cost of software construction is expensive. There are many variables. Our profession requires us to reduce costs. Costs drive profits. All businesses must be sustainable to survive.

Doing It Wrong

The fact is: far too many businesses with excellent products or services fail because they aren't sustainable or making enough money.

You are a professional. This is a business. Business demands that you make money. Reducing your costs helps to do just that. That cycle keeps you doing what you love and getting paid.

A savvy developer uses techniques to streamline, improve efficiency, reuse code, and reduce bugs and wonky behavior. An "in demand" developer knows that quality matters.

Quality Matters

Quality matters in all industries. When comparing two similar things to buy, the driving factors are their perceived quality and value.

Quality directly affects profitability. In software, quality breaks down to readability, reusability, and maintainability. Why? These three concepts drive the cost of software over its entire life-cycle.

Readability

Readability is your code's way of *telling* you what it's doing and why. It provides the means to quickly: find what you are looking for, understand what is happening, and do your work. It is readable today, six months from now, and years down the road.

Your code should tell you what is going on and why without comments. It should be readable and expressive.

Any team member can open your code and navigate through it. When they look at a function, they know what it does just by reading the code.

The DocBlock comments provide additional information on how to interact and what to expect when using the code. But the code itself is self-documenting, deterministic, and purposeful.

You want readable code so that you can open it, read it, and know what it does.

Reusability

The strategy of reusing code modules provides compelling benefits. It reduces your build time, test time, and project risks. It frees up your time so you can redirect your efforts to the custom, unique bits of each project.

The reuse practice dictates _reuse without modification_. These modules are complete. They are well-tested. Mechanisms are built-in for loading the unique implementation and dependencies. Each does a specific thing and does it well.

You want to reuse code over and over again to let you focus on the fun, custom parts of your project. This reduces costs and increases profits.

Maintainability

Maintaining code is expensive.

When a problem occurs, it takes you away from other paid work while you find the root cause, rewrite the code, test it, and close the issue. It's distracting and frustrating. The more time it takes, the higher the costs.

Maintaining code has risks too, as fixing a bug may cause another problem. Picture in your mind our bridge analogy. Imagine negotiating that rickety bridge, fixing one board, and then watching two other boards falling into the raging river below. The process of being in the code might cause another problem.

Readability, reusability, and maintainability all work together to mitigate risks and reduce costs.

You want your codebase to be maintainable so you can quickly get in, get out, and get back to what you love.

Wrap it Up

You should care about code refactoring for all the reasons above. The "why" of refactoring comes down to quality, costs, risks, and professionalism.

Let's get to work honing your refactoring skills.