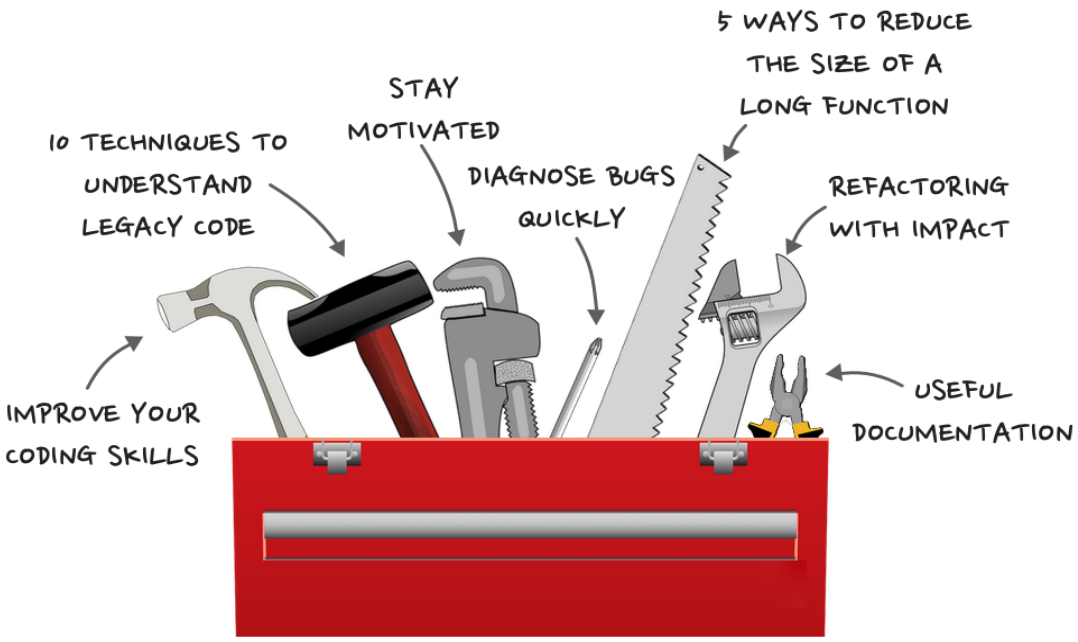
THE LEGACY CODE PROGRAMMER'S TOOLBOX

PRACTICAL SKILLS FOR SOFTWARE PROFESSIONALS WORKING WITH LEGACY CODE



JONATHAN BOCCARA

FOREWORD BY KEVLIN HENNEY

The Legacy Code Programmer's Toolbox

Practical skills for software professionals working with legacy code

Jonathan Boccara

This book is for sale at http://leanpub.com/legacycode

This version was published on 2022-02-22

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To Elisabeth

Contents

Foreword

What we have built is in the past. Where we are is the present. Where we want to be is the future.

Welcome to code. Welcome to the human condition. Welcome to the joys and frustrations of code, to the complexities of the human condition and to what happens when they meet and find themselves drawn into an intimate long-term relationship. Legacy code is as much about the people who surround and work in the code as it is about the code itself.

The default developer response to legacy code is primal. The fight-or-flight response kicks in, logic gives way to emotion, the question "What does the code need?" is replaced by "How can I spend as little time in this code as possible?" This is the thinking of short-term survival, not of long-term living, the habit of defend and attack, not of peace and construction.

To calm our response, we need first to appreciate that legacy code was not created arbitrarily or maliciously. Inasmuch as it may have been written in ignorance, this is because we are always operating with incomplete knowledge – and yes, that *we* includes you right now.

No matter what we may think of it, code is created systematically. Perhaps not always with a system of thinking and practice that is deliberate or appreciated, but we should understand that it is not arbitrary. The code was written by people – in which we must include our past selves – who were responding to a

situation with the tools – and the emotions and the knowledge and… – that they had available to them at the time. Start from there. There are no winners in a blame game.

To understand the system of code we should acknowledge the system of people and practices that created it, we should journey to retrieve the meaning in the legacy and make ways forward possible. There is an underappreciated ingenuity here, an application of time and tools and humanity that deserves our attention. This is what Jonathan is offering you in this book.

But rather than keep legacy at arms length, Jonathan brings it close. Legacy is very real and no one legacy is exactly like another, so you need a toolbox, not a silver bullet. And he understands and wants to help you. This is a book of compassion and intelligence.

If development in legacy code is a journey, Jonathan is your patient and loyal companion. He has stories for the road, ideas to help you on your way and suggestions for the route. He's also brought his tools along and invited you to share.

It is easy to forget that outside the world of software development, the word *legacy* has another meaning. A positive meaning, a gift of wealth from the past to the present for the future. This book will help you reclaim the word.

Kevlin Henney

Acknowledgments

Writing a book is quite some work, and not something one does alone. Several people have helped me shape this book, sometimes influencing it profoundly. And I suspect some of them aren't completely aware of the impact they had on it.

Now is my chance to express my gratitude.

My first thanks go to my dear wife, Elisabeth. Your invariable support has given me the means to stay motivated in the project of writing what I know about legacy code. I'm very grateful for your patience, for the interest you took in the project of this book as well as your kind understanding for the time it demanded. Your wise advice, that makes the most intricate matters look simple, was of an invaluable help. From all my heart, thank you.

I thank Patrice Dalesme who was my manager for many years. Patrice picked me up in his team as a young sprout who knew next to nothing, and undertook to teach me professional programming. On the top of caring about writing good code, Patrice has taught me how to create value and have a fulfilling professional life while working with existing code, either explicitly or simply be having him as a role model (some call him "King Patrice" or even "Magic Patrice"). Without him, I wouldn't have had much to say in this book. Few people have the chance to work with a developer and manager like you. You rock.

When I was considering taking on the enterprise of writing *The Legacy Code Programmer's Toolbox*, I started by crafting an

introduction along with the list of topics I intended to address. I showed it to several people, in order to test it and decide for a go or no-go for the project. Their warm enthusiasm for this project made it a full-speed go, and I'm grateful they encouraged me to carry on. These people are Ben Deane, Bryan St. Amour, Clare Macrae, Eric Pederson, Frédéric Tingaud, Ivan Čukić, Kate Gregory, Marco Arena, Philippe Bourgau and Ólafur Waage.

I also want to thank Kevlin Henney for his inspiration, feedback and support, and for writing the foreword of this book.

And last but not least, my gratitude goes to all the people who reviewed the book, that spent time and sweat in its construction along with me. The feedback that each of you gave has transformed the book in its way, and this makes you part of it. A big thank you to Arne Mertz, Avi Lachmish, Bartłomiej Filipek, Barney Dellar, Bilel Belhadj, Clare Macrae, Eric Roussel, Feng Lee, Francisco Castillo, Ivan Čukić, Javier Sogo, Jean-François Lots, JFT, Kobi, Kyle Shores, Marc-Anton Böhm, Matthieu Poullet, Miguel Raggi, Nikolai Wuttke, Rakesh UV, Riccardo Brugo, Swarup Sahoo, Thomas Benard, and Yves Dolce.

Finally, amongst the reviewers, I want to dedicate a special thanks to Phillip Johnston, Ricardo Nabinger Sanchez and Tim van Deurzen, for the impressive amount of work in their reviews and the meticulous analyses they performed to produce them.

Introduction: There is a lot of legacy, out there

Hello, brave sailor in the ocean of legacy!

If you're opening this book, I'm assuming that you have to deal with legacy code on a regular basis, if not every day, in your job as a software developer. And I am also assuming that you are resolved to face it, learn from it, thrive in it, and live on a happy life.

If those assumptions are correct, then you've come to the right place. This book will give you insights on how to perceive legacy code, how to act on it, how to act on your fellow developers, and how to live a happy life (well, to the extent that legacy code is concerned, that is!).

Legacy code

First things first. Let's start by agreeing on what we call legacy code. You have a fair idea of what that is, I do too, but let's make sure that we're in line.

There are several definitions of legacy code out there. For example, in his book *Working Effectively with Legacy Code*, Michael Feathers defines legacy code as code without tests.

Wikipedia gives a more formal definition, that is "source code that relates to a no-longer supported or manufactured operating system or other computer technology."

There are also less formal definitions, for example some people call legacy code the code that was written before they arrived in a company.

For our purpose, we will use a definition that tries to stick to the meaning of legacy code in usage, the one you probably thought about when reading this book's cover.

When we talk about legacy code, we will mean code that:

1. is hard for you to understand,
2. you're not comfortable changing,
3. and you're somehow concerned with.

Note that under this definition, someone's legacy code can be someone else's normal code. Indeed, according to part **1)**, a piece of code can be hard to understand for you, but easier for another person (the person who wrote it, for instance).

In practice, there can be spots in a codebase that constitute legacy code for pretty much every developer of the team.

Also note that part **2)** of the definition ties up with the "code without tests" definition. If there are no tests around a piece of code, you won't be comfortable changing it. Our definition is not quite the same as Michael Feathers's though, because even if a piece of code has *some* tests around it, it doesn't necessarily mean that you're comfortable changing it; for instance if only few cases are covered.

Finally, for it to be legacy for you, a piece of code must concern you. There is plenty of code on the planet that you'd have trouble to understand and would feel quite uncomfortable changing. And it's not legacy code for you. It's code that you don't care about. Hence part **3)** in the definition.

You didn't become a developer for this

Legacy code can be hard to take on, especially at the beginning of your career, when you come in on your first day, excited to be paid for spending your days programming.

Indeed, legacy code looks nothing like what schools teach. If you studied computer science, you typically got assigned to projects that you built from the ground up.

You understood them well because you designed sizeable parts, if not all, of their structure. You felt comfortable changing them for the same reason, and in fact you didn't have to change them so much, because you weren't maintaining them for years.

If you're a self-taught programmer, you had total freedom to choose what to work with, and you were driven by your interest. You could start an exciting project, interrupt it to build a library for it, interrupt it to build another library to implement a cool pattern you had just learned, and so on.

Programming small projects is extremely rewarding. You can see your program grow very quickly, you add features almost as fast as you think of them, and very rarely encounter the concept of "regression". Joy.

And one day you get into professional programming. While you still get to interact a lot with code, things work differently. There is existing code, to begin with. And to add features you need to figure out how this existing code works.

Then you need to be very cautious when you add a feature, because it could have an impact on this existing code. So you have to spend time writing tests, maybe even refactor existing code in order to write tests, deal with regressions, and accept that it's not so fast and easy any more.

And you face legacy code. It can be daunting. You need to understand someone else's writing, or plenty of people's writings combined. You have to think about why they did what they did. You have to understand the structures in the code and how they relate together.

All this understanding demands some intellectual effort, especially if it is legacy code (because of part **1)** of the definition), and it's much less of a smooth ride than writing your own code.

What's more, you have to modify it! As a developer, you're paid for changing code, not only for staring at it and trying to figure it out. And part **2)** of the definition of legacy code puts you in a difficult position.

Recognizing yourself in this yet? If so, then great, because this book is targeted at you.

There is a lot of legacy, out there

When you think about it, all this makes sense: you're not working on the same sort of application as when you were a young sprout learning to program.

The implementation you're making now is helping people on a day-to-day basis. It will be there for a long time hopefully, you make a living out of it, and your customers expect a level of quality and richness of features that you can't get out of a single developer. And even less out of a single developer hopping from project to project.

The first thing I want you to realize is that you and your company are **not the only ones facing legacy code**. Far from that.

I've worked with large codebases (in the tens of millions of lines of code), I'm an organizer of a Software Crafters meetup, I go to a decent number of conferences, and I run a popular blog about making code more expressive (fluentcpp.com[1]). In all those places, I get to meet people that deal with legacy code on a daily basis.

And I can tell you, legacy code is everywhere.

Some people don't realize this and feel like they're out of luck, because they ended up in a place where programming isn't as fun as it used to be for them.

The bright side for you is that if you face legacy code, it's normal. You don't have to flee your company just because of this, since the other one down the street will certainly also have legacy code.

And the even brighter side is that there is a way for you to be happy with the reality of programming. This is what this book is about.

Here is the program: we will start off with the right attitude to

[1] https://www.fluentcpp.com

deal with legacy code (Chapter 1), then we'll move on to how to use legacy code to improve your programming skills (Chapter 2) and where to find good code to get some inspiration (Chapter 3). This constitutes Part I of the book, about approaching legacy code.

Part II is about understanding the code itself. It covers 10 techniques to understand code you don't know. Those techniques are spread across three chapters: how to get an overview of the code (Chapter 4), how to become a code speed-reader (Chapter 5) and how to understand code in detail (Chapter 6).

Then in Part III we will dive into the topic of knowledge. We will see why in legacy code Knowledge is Power (Chapter 7), how to make knowledge flow around your team (Chapter 8) and focus on a specific practice to inject regular doses of knowledge: the Dailies (Chapter 9).

In Part IV, we'll see how to cut through legacy code, by jumping to the places that matter to find the source of a bug quickly (Chapters 10 & 11), deciding when to refactor the code to add value to it or when to leave it alone (Chapter 12), and clearing out long functions by making them shorter (Chapter 13)

Hoping to change your reality of programming with legacy code, let's begin.

Part I: Approaching legacy code

Chapter 1: The right attitude to deal with legacy code
Chapter 2: How to use bad code to learn how to write great code
Chapter 3: Why reading good code is important (and where to find it)

Chapter 1: The right attitude to deal with legacy code

Over the years I've had the time to experience how it *feels* to work with legacy code, and I have watched and discussed with many developers who also expressed their feelings about it.

A lot of developers have a hard time working with legacy code. Day in and day out, they trudge through their tasks by interacting with it.

But a few people seem to breeze through their tasks, even if those imply working with legacy code. It's not that they have super powers, but they seem to not mind working with legacy code, and even display a positive attitude in the face of tasks that would rebuke others.

I believe that there are two mindsets regarding legacy code: the natural mindset, and the effective mindset.

The people who win through are those that stick to the effective mindset. On the other hand, those that stick to the natural mindset have a harder time going on with their jobs dealing with software.

Let's see what those two mindsets are about.

The natural reaction: who the f*** wrote this

Picture yourself navigating a legacy codebase.

You're looking for something, and what you stumble across is a pile of tangled code that you can't make any sense of. Or some code that looks very poorly written. Or both.

How do you react?

One of the possible reactions is the natural - or primal - one. It consists in deciding that:

- This code is a pile of crap,
- The person who wrote it had no idea what they were doing,
- You would have done such a better job,
- You're way better than this, maybe you should find another place that deserves your skills.

Have you ever felt that way? I've seen many people do so. I've experienced it myself before I knew better.

And it's normal, because that's what this mindset is: *primal*. That's what we're wired to do, for whatever psychological reason that makes us feel good about it.

A primal attitude is fantastic though, if you're fighting against a gorilla in the jungle, or if you're being pursued by a serial killer in a dark alley.

But in software development, the primal attitude sucks.

As a software developer, you're paid for being rational, not for being primal. And if you look at legacy code the rational way, quite a few things start looking different.

I don't imply that the primal feeling always leads to the wrong conclusion. Maybe a particular piece of code is crap, maybe it was written by somebody who wasn't qualified, and maybe you ought to be somewhere else.

But often, putting on your rational hat offers a brand-new perspective on a piece of code. And if you're a software developer, it is a splendid rational hat that you're wearing. So let's dress up and put it on: let's see legacy code for what it really is.

A humble view of legacy code

Legacy code is not your enemy. In fact, when you think about it, we may go as far as saying that we can be software developers *thanks* to legacy code.

Legacy code made the application grow

How is that? The early stages of development of a piece of software are the crucial moments where it took off, captured clients, built on cash, and created a brand and image to its customers. This was done with a product, and behind it was the code.

Some of this code may be still around now. Some of it may not be around, but affected the architecture in a way that has left traces until today. But it is this code that performed the features that your customers liked your company for in the first place. This

is legacy code. This is your legacy, and without it you would probably not have a job today.

The amount of work put into a legacy codebase is often colossal, if you add up all the time invested by every developer that worked on it. Legacy code typically contains lots of features, and a myriad of bugfixes. If you were to start from scratch you'd have to go through most of the bug analyses again, and fix them.

Now let's try to look at a piece of legacy code for what it really is, by putting ourselves in the shoes of the person who wrote it.

Legacy code has time on its side

Legacy code is often more or less old. If you mentally travel back to the time it was written, do you think people knew as much as you do today?

Did the person who wrote it know the best practices that we are slowly putting together as a development community over the years? Did anyone know at that time?

Could they anticipate the direction that today's programming languages, Modern C++, the latest versions of Java, Python, JavaScript, and so on, were taking, and that is now so familiar to us?

Some legacy code that is around now has been written when some of us where in school (and for some in the early stages of school). Back then, technology wasn't what it is today, and conversely, a disturbing thought is that the best code we can write now might be laughed at in ten years.

So if we were in the actual position of the person who wrote that

code, it may well be that we wouldn't have done such a better job.

What's even more humbling is to take the time to think about how we would have solved the problem that some code is trying to solve.

Surveying a piece of legacy code at a glance, we can see things that seem absurd, and our reptilian brain starts sending out this feeling of superiority to its friend the higher brain: "had I been given the chance, I would have done so much better".

Well, maybe we would have done better. But often, when we actually try to solve the problem, with all the technical and functional constraints surrounding it, we become aware of certain things that prevent us from doing an elegant design. Things we hadn't seen at first sight.

You never really know until you actually try it. And even then, you may not know until some tests (or in the worst case, some customers) bring up an issue coming from the design.

Seeing legacy code for *who* it really is

Still being rational, that picture of an evil and under-qualified person writing ridiculous code to make your life hard doesn't fit the reality for at least one simple reason: it's not just one person.

Legacy code becomes tangled and difficult to understand because of an inconsistent accumulation of changes, made by many people, who sometimes weren't even employed by the company at the same time.

Consider the very first version of the code. Maybe it made some sense, but didn't express its intentions well. Then the developer

who took it on next may have understood it a bit differently, adding a change that wasn't exactly in line with what the code was designed for. But it still wasn't too bad. And then the next person arrived, gave yet another new direction to the code, and so on.

When you add this all up with many people over many years, what you get is a chunky piece of legacy code (this is why being expressive is such a determining characteristic of success for code).

Therefore, the code you see today that made you - primally - want to hit someone with a club over their head, doesn't have one culprit. To be really fair, you would have to go find many people, some of them off doing other projects, and gently tap each head with your club, over which you would have placed a cushion before. This way, you would spread your punishing blow out equitably. Alternatively, you could pick someone at random to bash, but there is no way that could be called fair.

Finally, let's ask ourselves a challenging question: didn't we also write legacy code? When you look back at the code you wrote a while ago, do you find it crystal clear and elegant?

First of all, there is a context that we have when we're "in the zone" when coding, that we lack when looking at code just out of the blue.

But above all, you should almost *hope* that you don't like your past code too much. Indeed, if you look at code you wrote a year ago and don't find it could be improved, it means you haven't learned much over the past year - and that is definitely not what you want.

The efficient approach: taking ownership

Now that you see legacy code with a rational eye, what can you do in practice to quit the primal mindset and join those in the efficient mindset?

The first thing to do is: **don't complain if you are not intending to improve the code**.

Complaining for the sake of it is just making noise, and a harmful one. Moaning about how you don't understand a piece of code, or how terribly designed it is, or how you don't like it in any way won't get you anywhere.

Worse, it is a self-fulfilling prophecy that won't help you handle legacy code better at all, quite the contrary. It puts you in the position of a victim, not the one of an active player.

More than just you, this affects the people around you. Someone who criticizes the code just for the sake of it sets an example, and little by little the team gets used to this. It creates an atmosphere where people see around them that it's the normal thing to just blame the code.

This is particularly contagious to the younger persons of the team. If you have young developers around you, or if you are a manager of younger developers, choose to be a model for them in terms of attitude. It's a bit like watching your language around kids so that they don't get a bad vocabulary.

And if you do have people that make non-constructive complaints around you, make an active effort to not get sucked in. Don't criticize the code just for the sake of it. Be particularly

careful to this if you're towards the beginning of your career, so that you grow positive habits.

The second aspect of the efficient mindset is to **consider that the code you're working on is *your* code**.

Whether you wrote it yourself or not, however good or bad you think it is, this is your code and you have responsibility over it.

Note that taking ownership is not the same as taking the blame - for whatever bad design somebody had to come up with before you even graduated. It means acknowledging that you are the one in charge now. Indeed, even if you're not a manager, you are in charge of at least some portion of the code.

Entering this mindset transforms the way you see legacy code. It's no longer something that some people, distant in time and space, have written and that you can criticize at will to show how bad it is and how good we are. It is your code, and you are here to make the most out of it.

When I came across this mindset (thanks to my fantastic manager Patrice Dalesme) I became motivated to do whatever was in my power to understand my code, improve it, and create business value out of it. Several years later, I'm still just as motivated to do so.

Having a role model

Having a model helps a lot to settle into the right mindset. Try to identify a developer from your company that is in the effective mindset, and get inspired by them, as with a mentor. It's practical

if it is your manager, but it doesn't have to be. That person may even not know that you're taking inspiration from them.

How to identify which developers are in the right mindset around you? Look at who gives the impression not to mind tackling hard legacy pieces of code!

Even if you can't change the past and how code was written, you have the power to control your attitude, and this will affect the future of how you work. Choosing the right mindset makes a difference, be there in your efficiency as a developer or in your happiness as a person in day-to-day life.

It's always a good time to enter the efficient mindset, and the perfect moment to do this is… now. Take responsibility, take ownership.

Key takeaways

- Suspect that writing the code you're reading must have been harder than it looks.
- Realize that code difficult to read is not the work of one evil character.
- Avoid complaining if you don't intend to improve the code.
- Take ownership of the part of the codebase you're working on.
- Find a role model to get inspired by their attitude.

The Legacy Code Programmer's Toolbox

Liked what you've just read?

The rest of the book The Legacy Code Programmer's Toolbox contains a ton more on how to be efficient, create value, and have a fulfilling professional life while working with legacy code! Get the rest of the book here: leanpub.com/legacycode[2]!

Here are the contents of the book:

Foreword

Introduction There is a lot of legacy, out there

Part I - Approaching legacy code

Chapter 1. The right attitude to deal with legacy code

Chapter 2. How to use bad code to learn how to write great code

Chapter 3. Why reading good code is important (and where to find it)

Part II - 10 techniques to understand legacy code

Chapter 4. 3 techniques to get an overview of the code

Chapter 5. 4 techniques to become a code speed-reader

Chapter 6. 3 techniques to understand code in detail

[2] https://www.leanpub.com/legacycode

Part III - Knowledge

Chapter 7. Knowledge is Power

Chapter 8. How to make knowledge flow in your team

Chapter 9. The Dailies: knowledge in regular doses

Part IV - Cutting through legacy code

Chapter 10. How to find the source of a bug without knowing a lot of code

Chapter 11. A case study to find the source of a bug quickly without knowing a lot of the code

Chapter 12. What to fix and what *not* to fix in a legacy codebase

Chapter 13. 5 refactoring techniques to make long functions shorter

Conclusion The legacy of tomorrow

Here is what other readers say about *The Legacy Code Programmer's Toolbox:

> "This is a warm and reassuring book that will equip you to read, understand, and update legacy code in any language.
>
> The advice is immediately actionable, and you can start to use it right after reading the chapters. The experience of the author is clearly hard-won; he generously shares it to save you a lot of trouble.
>
> The material will leave you ready to take on whatever legacy code you encounter, with a smile on your face. I happily endorse it."

Kate Gregory

"Let's face it - legacy code is everywhere! We can complain or... make it our friend. And this is exactly what Jonathan is offering is his book.

With a vivid language, lots of examples and use cases the text will shift your attitude towards legacy code. You'll be equipped with a lot of tools to make your daily job much fun and rewarding."

Bartlomiej Filipek

"*The Legacy Code Programmer's Toolbox* gives actionable advice about how to deal with the sometimes harsh reality of our work. You'll learn how to understand and when to refactor legacy code, and what attitude keeps you sane and productive when facing legacy code.

This book is a great read for everyone: Junior developers wondering what is coming for them and how to face it, and seniors still wondering what could have been done differently when that old project came to a screeching halt."
Arne Mertz

"Wow! I read the book in one day. For two reasons. First, it is quite entertaining. Second, it is even more enlightening.

Jonathan Boccara wrote a unique book about our day to day life as a professional software developer: Working with legacy code. He shows with many examples, how we should approach, understand, and improve legacy code if necessary.

You should read it, because Jonathan's book will give you new, critical insight."

Rainer Grimm

"As I read Jonathan's book I found a lot of comfort knowing that it will be a lot easier for many developers coping with understanding & working with legacy code.

The book helps you get in the right mindset to deal with legacy code and explores various techniques and tools to help you along the way, with lots of carefully crafted code examples.

I enjoyed this book a lot and learned some handy tricks along the way. "Jonathan's toolbox" just became my top recommendation on this subject."

Victor Ciura

Get the full book on leanpub.com/legacycode[3]!

[3] https://www.leanpub.com/legacycode

Chapter 2: How to use bad code to learn how to write great code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Don’t like the code? Elaborate, please.

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

What this technique will bring you

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Share your findings

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The vaccine against bad code is bad code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The vaccine analogy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Be aware of what good code looks like

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 3: Why reading good code is important (and where to find it)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The importance of reading good code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Where to find good code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Your standard library

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Where to get some practice

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Your almost-standard library

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The next version of the language

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Open source projects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Libraries implementations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Your legacy codebase

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Become more efficient with legacy code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Part II: 10 techniques to understand legacy code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 4: 3 techniques to get an overview of the code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

1) Choosing a stronghold

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

2) Starting from the inputs and outputs of the program (and how to find them)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

3) Analysing well-chosen stacks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Flame graphs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 5: 4 techniques to become a code speed-reader

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

1) Working your way backwards from the function's outputs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

2) Identifying the terms that occur frequently

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Locating the important objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Understanding how inputs are used

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Intensive uses of an object

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Continuing the analysis of `loadAlignments`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Combining techniques

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

3) Filtering on control flow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

4) Distinguishing the main action of the function

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 6: 3 techniques to understand code in detail

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

1) Using “practice” functions to improve your code-reading skills

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

2) Decoupling the code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

A refactoring example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

What refactoring this code taught us

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

3) Teaming up with other people

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

It gets easier with practice

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Part III: Knowledge

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 7: Knowledge is Power

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Where did the knowledge go?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The Knowledge Handicap

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Knowledge goes away

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Little knowledge is next to no knowledge

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The lack of knowledge leads to chaos

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 8: How to make knowledge flow in your team

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Writing precious documentation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Writing documentation that is read

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

What topic to document

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Writing style matters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

A common location

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Writing documentation that doesn't get (too much) out of sync with the code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Writing documentation is cool

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Telling your tales: acquiring knowledge in Eager mode

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Knowing who to ask: getting knowledge in Lazy mode

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Pair-programming and mob-programming

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

External sources of knowledge

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Make the knowledge flow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 9: The Dailies: knowledge injected in regular doses

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

What are Dailies?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Monthly sessions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The major benefits of Dailies

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

There is plenty of content out there

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Be the one who spreads knowledge

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Part IV: Cutting through legacy code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 10: How to find the source of a bug without knowing a lot of code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The slowest way to find the source of a bug

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The quickest way to find the source of a bug

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Step #1: Reproduce the issue

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Step #2: Perform differential testing to locate the issue

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Step #2a: Start with a tiny difference

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Step #2b: Continue with larger differences

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Step #3: Formulate and validate a hypothesis

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

A binary search for the root cause of a bug

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

A case study

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 11: The Harmonica School: A case study in diagnosing a bug quickly in an unfamiliar code base

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Lesson subscriptions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The bug report

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Let's find the source of that bug, quickly

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Step #0: Don't start by looking at the code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Step #1: Reproduce the bug

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Step #2: Perform differential testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Step #3: Formulate hypotheses and validate them in the code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The more time you spend in the application, the less total time you spend debugging

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 12: What to fix and what *not* to fix in a legacy codebase

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Legacy code is a bully

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The value-based approach (a.k.a. “Hit it where it hurts”)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The costs of a refactoring

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The cost of understanding the existing code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Fixing regressions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Adding tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Handling conflicts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The value of a refactoring

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Where does it hurt?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Slice up a big function

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Slice up a big object

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Make side effects visible

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Use names that make sense

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Don't Repeat Yourself

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Get value now, pay the cost later

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Use the value-based approach

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Chapter 13: 5 refactoring techniques to make long functions shorter

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The birth of a Behemoth

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Identifying units of behaviour

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

1) Extract `for` loops

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

`for` loops representing algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

`for` loops performing a simple iteration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

2) Extract intensive uses of the same object

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

3) Raise the level of abstraction in unbalanced `if` statements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Error cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

4) Lump up pieces of data that stick together

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

5) Follow the hints in the layout of the code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

6) Bonus: using your IDE to hide code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The impact on performance

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Conclusion: The legacy of tomorrow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

The bigger picture of writing code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

How to deal with legacy code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

But you're also person A

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

Parting words

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.

References

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/legacycode.