Bad Programming Practices 101

Become a Better Coder by Learning How (Not) to Program

Karl Beecher

Apress®

Bad Programming Practices 101

Become a Better Coder by Learning How (Not) to Program

Karl Beecher

Apress®

Bad Programming Practices 101: Become a Better Coder by Learning How (Not) to Program

Karl Beecher
Berlin, Germany

ISBN-13 (pbk): 978-1-4842-3410-5 ISBN-13 (electronic): 978-1-4842-3411-2
https://doi.org/10.1007/978-1-4842-3411-2

Library of Congress Control Number: 2018933065

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Chaim Krause
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, email orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please email rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484234105. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dedicated to all the writers who show that serious and fun are not mutually exclusive.

Table of Contents

About the Author

Karl Beecher lives a double life as a writer and software specialist.

When being a writer, he focuses on science and technology. He likes to take meaty, complex ideas and present them in ways that are easy to understand.

As a software specialist, Karl has worked as a software engineer, earned a PhD in computer science, and co-founded a company specializing in management of large-scale IT operations.

About the Technical Reviewer

Chaim Krause presently lives in Leavenworth, Kansas, where the U.S. Army employs him as a simulation specialist. In his spare time, he likes to play PC games, and occasionally he develops his own. He has recently taken up the sport of golf to spend more time with his significant other, Ivana. Although he holds a BA in political science from the University of Chicago, Chaim is an autodidact when it comes to computers, programming, and electronics. He wrote his first computer game in BASIC on a Tandy Model I Level I and stored the program on a cassette tape. Amateur radio introduced him to electronics, while the Arduino and the Raspberry Pi provided a medium to combine computing, programming, and electronics into one hobby.

Acknowledgments

I'd like to thank my editors, Mark Powers and Steve Anglin, as well as all the others at Apress who made this book both possible and a pleasure to produce.

And, as ever, thank you to my wife, Jennifer, for her love, support, and invaluable feedback.

Introduction

So, you're a programmer, or at least a programmer-in-training.

You want to improve your programming skills. You want to become more productive as soon as possible.

You'll be working with colleagues who want their project to be successful and their code bug-free. They'll examine the code you write and serve as gatekeepers, either accepting or rejecting your contributions. Your colleagues want you to write code that's up to scratch.

The question is: how should you go about learning to do all this? One idea would be to read up on what the best programming practices are and then apply them in your work. However, the matter of how best to program is a touchy subject.

One of the easiest ways in the world to get an argument started is to ask a group of coders about good practices. Like the old jibe about economists,[1] if you ask three programmers what the best practice is on a particular topic, you'll get three different answers (and a fair few raised voices). Typical questions might be:

- Should the use of `goto` be allowed?
- What's the best policy for naming variables?
- What's an acceptable level of complexity for a subroutine?
- What is the maximum size for a class?
- How much code should be covered by tests?

In a perfect world, we'd have easy answers to these questions, but a world that gives us five *Pirates of the Caribbean* movies is far from perfect. The truth is that questions like these often have complex answers that depend on multiple factors. In any situation, there could be many acceptable solutions. A best practice rarely applies in all contexts.

This book helps you by taking a different approach.

[1] It goes something like, "Ask a question of three different economists, and you'll get four different answers."

In my experience, programmers tend to agree much more readily on matters of how *not* to program. Ask them, for example:

- Should I write code with absolutely no comments?
- Should I prefer global variables over local variables?
- If a pointer might be null, should I avoid checking its value?

To these three questions, you'd find a much stronger agreement among the answers: no, no, and *f*** *no*!

Many bad programming practices exist, practices that make experienced coders grow red-faced with anger or break out in sweaty, shivering fear. The truth is, you will occasionally write code that causes reactions like this, particularly in the early stages of your career. A key to accelerating your development as a programmer will be to learn which practices are bad and then avoid them.

This book doesn't focus on how you should program. After all, competing best practices suit various projects differently. What's more, the field of programming develops constantly. New approaches are found, and existing techniques are improved all the time. A list of good practices won't remain current for very long.

Instead, this book advises you how *not* to program. It takes advantage of the fact that oodles of code has been written in the preceding decades and a lot of things have already been tried out. A combination of experience and research exists that shows which stuff works badly and is generally to be avoided.

Avoiding the bad practices listed in this book will give you a head start in becoming a better and more productive programmer. After that, you can go on to argue the issue of good practice to your heart's content.

A Note on the Style

You might have already observed that the style of this book is rather tongue-in-cheek. It gives advice as if the reader is seeking to become a failure: a programmer who ignores the rules and follows the worst practices, a programmer whose contributions are regularly rejected or (on the rare occasions they make it through review) create nasty bugs in once-functioning software. I think this makes the book a fun and enjoyable read.

Occasionally, a reasonable voice interjects and explains why programmers view a particular practice as bad. It might be because of a consensus among professional programmers or because of some empirical research. In any case, that reasonable voice appears in sections bearing the heading *Thumbs Down!*

What I Mean by *Programming*

A bad way to begin learning how to program is to mistake what *programming* actually means. Therefore, let me make clear what *I'm* using the term to mean.

After more than a decade working with software (and a misspent youth spent learning to code), I view programming as problem-solving. Roughly speaking, a programmer begins their work at some starting point, A, with a problem statement. The programmer's job is to chart a path to the goal point, B, which results in a software-based system that solves the original problem acceptably.

The journey from A to B can be long and may include many complex steps along the way. The nature of the work involved depends on how broadly you define *programming*. For the purpose of this book, I'll distinguish two types of programming:

- Programing in the *narrow sense*: By this, I refer to what many others call *coding*. Problems in this sense are problems of missing or broken software, and the solution is to write code that fixes them.
- Programming in the *wider sense*: A fuller appreciation of programming acknowledges that coding is only part of the job. The larger job is to provide a solution that is complete, high-quality, and acceptable to the user.[2] This is much more than coding. It involves other activities, like requirements analysis, system design, or acceptability testing. It also includes *lots* of communication and collaboration, not just among the programming team but with the users too. Naturally, this requires skills beyond writing good code.

This book focuses on programming in the narrow sense. That's not because the stuff involved in the wider sense is less important—far from it. I've chosen to keep the focus narrow because of who the book is aimed at. The intended audience—students,

[2]Sometimes called *software engineering*.

apprentices, junior developers—usually focuses on coding-related activities and should master those before they shift their attention to the wider issues.

That said, bits of stuff from the wider sense get an occasional look-in throughout the book. What's more, one of the final chapters focuses on testing, a topic that moves the discussion away from purely coding and toward coding a solution that's acceptable to the user.

Nevertheless, don't mistake this book for one that deals in wider issues of software engineering.

CHAPTER 1

Learning to Program

Objectives

In this chapter, you'll learn:

- How to mess up your approach to learning programming
- Poor ways to choose your tools

Introduction

This chapter starts things off by discussing how to learn programming and gather your tools in preparation for writing code.

Naturally, since you want to be a bad programmer, it tells you to make a mess of these tasks.

Bad Ways to Learn Programming

If you choose programming as your career, you'll probably never stop learning. Software is a fast-moving field. Radical new tools and advances come along more often than Stephen King books.

This section will show how you can scramble around trying to learn programming while taking in barely a thing.

Take a Pass on Practicing

Your Spanish teacher always told you that you'd never learn the language by reading it from a book. Practice, practice, practice, they told you. It's the only way you'll learn to speak a foreign language.

K. Beecher, *Bad Programming Practices 101*, https://doi.org/10.1007/978-1-4842-3411-2_1

But everyone knows that programming languages and spoken languages are different, right? Surely, just like math or science, you can learn programming by reading it in a book (God knows there are enough programming books). After all, you didn't learn Newtonian mechanics by building a giant centrifuge—you learned formulas from a textbook.

In short: read the programming manual but ignore the "Exercises" section.

Thumbs Down!

Actually, it turns out that learning Spanish and learning a programming language are somewhat similar, specifically in that book learning needs to be complemented by practice.

Teachers and students alike testify that practical approaches to learning programming—such as participating in quizzes, performing textbook exercises, or doing coursework—are extremely helpful when learning (Lahtinen et al., 2005). Just like foreign-language speakers, experienced programmers preach the same advice: if you want to get good, then practice!

There's just something about practical application that helps make what we learn stick with us in the long term (Dunlosky, 2013). A book can tell you what a variable is, a lecturer can show you how a `for` loop works, but to really come to grips with programming, you need to write your own programs.

Benefits include the following:

- Long-term retention of knowledge is improved, making it less likely you'll forget stuff.
- You'll habitually make fewer mistakes.
- An easier grasp of the underlying principles of programming is achieved.

Avoid Inspiration

In spite of the previous advice, you might find yourself wanting to practice your programming skills. If you don't have the self-control to resist these urges, then the question arises: what form of practice should you choose?

Whatever you do, don't waste any time being choosy or imaginative. Just grab the first exercise you can find, preferably one that deals with a topic that doesn't interest you.

I say this because when you start to think about what you'd like to build, there's a risk that you will become inspired, and inspired is a very dangerous state to be in for those intent on doing badly. It's all too easy to get swept up by enthusiasm.

Thumbs Down!

Seymour Papert, a pioneer in both computer science and education, knew the power of inspiration. One of the central principles he championed was "project before problem." This advocates letting your own interest direct your learning (Papert, 1996). Instead of being given formal problems to solve by someone else, he recommended you look inside yourself and come up with your own projects. Discover what interests you, what you would enjoy creating, and what would delight you to see built, even if just for the intrinsic pleasure of doing it. Working on something because it inspires you creatively is a powerful impulse, one that can disguise the fact that you're actually learning at all.

Be a Script Kiddie

Assuming you have a practice problem to solve, the next step is to formulate a solution, preferably a bad one.

Of course, you're not going to actually think about the problem yourself. As mentioned earlier, applying your knowledge risks increasing your retention and improving your skills.

It's far better to scour the web for snippets of code that solve the same problem. You can simply copy and paste them blindly[1] and claim that you've written a solution. You might even succeed in convincing yourself that you really learned something.

[1]The pejorative name for someone doing this copy-paste style of coding is *script kiddie*.

However, if you're going to take this approach, then be careful. It's not as poor a practice as you might hope. In fact, many educators will encourage you to study good examples written by others, as long as you put in the effort to understand what makes them good. So, whatever you do, don't accidentally learn how the copied solutions actually function.

Do It Alone

A good way to slow your growth as a programmer is to learn alone. If you collaborate with other learners, you leave yourself open to the following risks:

- Exposing yourself to other ideas and perspectives that can improve your own.
- Reinforcing your own understanding by explaining the material to others.
- Increasing your understanding by having material explained in terms you're more likely to understand.
- Having a greater tendency to speak up and ask questions.
- Gaining a taste of what a real team project is actually like.

Bad Ways to Choose Your Tools

Of lesser importance, but still worth consideration, is your choice of tools. This is important whether you're a beginner or a veteran programmer. There are many types of tools, just a few examples of which are featured in Table 1-1.

Table 1-1. *Examples of Programming Tools*

Type	Purpose	Examples
Programming language	Write instructions for a computer to execute	Java, Python, C++
IDE[2]	Integrates many software development tools (e.g., code editor, compiler, and debugger)	Eclipse, IntelliJ, BlueJ
Source-code generator	Automatically write programs, usually requiring the programmer to fill in certain details afterwards	A built-in feature of many IDEs, such as Eclipse and IntelliJ
GUI[3] builder	Build a graphical user interface via drag-and-drop instead of writing source code	Eclipse WindowBuilder, IntelliJ GUI Designer, Qt Creator
Version-control system	Manage the history of changes made to files in a software project	Git, Subversion, Mercurial
Code review	Submit code to be inspected and approved by colleagues	Gerrit, Review Board

Choose Inappropriately While a Beginner

We all have different levels of understanding, and so different tools can serve each of us better at any one time. As a beginner, you're still learning a lot, and learning is a balance between not being overwhelmed on the one hand and being challenged on the other.

Of course, there are obvious benefits to avoiding challenge: it's a lot less effort, and you don't learn anything new. Therefore, you should ensure that the tools you choose are overly simplistic for your current level and automate a lot of things you don't understand.

But, if you're intent on sabotaging your own learning, there are also benefits to being overwhelmed. Choosing a tool aimed at pros (like a heavy-duty IDE) might frustrate you, but be assured that you'll impress people who see you using it, and your learning progress will be reduced to a crawl.

[2]Integrated Development Environment

[3]Graphical user interface

Thumbs Down!

To get the balance right instead of wrong, choose tools appropriate to your abilities. If you're very early in your learning career, you might still struggle with basic concepts like variables or loops, in which case a drag-and-drop environment or a visual programming tool, like Stratch or Alice, might serve you better (Powers et al, 2006).

Once you're accustomed to a tool, you can consider exploring something more complex like Visual Basic, BlueJ, or even an IDE like IntelliJ if you're ready for it. The more complex tools tend to be more powerful.

A tool's job is usually to make a task more convenient, often by automating some things. Automation can be helpful in two scenarios:

- You don't understand a task and rely on the tool to get the task done at all.
- You do understand a task, but use the tool to get the task done quicker.

Make sure you know which of these is the case for you. For example, a code generator is all very well, but any good teacher will ask you if you understand the generated code. If not, then how do you expect to change it later if it requires adaptation? In which case, play around with the generated stuff to see how it works.

Obsess Far Too Much over Your Choices

Do you have a 48-inch TV but feel sure that a 50-inch screen would really make all the difference? Do you have an iPhone 7 but lie awake at night obsessing over the iPhone 8 and the colossal difference to your life that having one might make?

Then this piece of advice is for you: obsessing over the choice of tools as though they make a huge difference is a great way to waste time and effort. For example:

- If your project must target Java version 7, then waste hours of everyone's time arguing in favor of Java 8 because its new Streams API will somehow give you superpowers.
- If Git is your team's version-control system, stamp your feet for days on end and demand switching to Mercurial because it has magical unicorns, or something.