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 xiii

About the Technical Reviewer xv

Acknowledgments xvii

Introduction xix

Chapter 1: Learning to Program 1

Objectives 1

Introduction 1

Bad Ways to Learn Programming 1

Take a Pass on Practicing 1

Avoid Inspiration 2

Be a Script Kiddie 3

Do It Alone 4

Bad Ways to Choose Your Tools 4

Choose Inappropriately While a Beginner 5

Obsess Far Too Much over Your Choices 6

Be a Fashion Victim 7

Chapter 2: Layout and Structure 9

Objectives 9

Prerequisites 9

Introduction 9

Make Spacing Poor and Inconsistent 10

On the Level 10

Spaced Out 13

Tabs and Spaces 14

Clutter the Code 15
Unused Stuff 16
Dead Stuff 16
Disabled Stuff 17
Write Bad Comments 18
No Comment! 18
Code Parroting 19
Out of Sync 21
Avoid Structured Programming 22
Jump Around 25
Routine Work 26

Chapter 3: Variables 31

Objectives 31
Prerequisites 31
Introduction 31
Use Obscure Names—Thinking Up Meaningful Labels Isn't Worth the Effort 32
All Meaningless 32
Vowel Movements 34
Lazy Naming 35
Treat Variable Declaration Like a Waste of Time 35
Be Confusing 35
Be Contrarian 36
Maximize the Scope of Variables 37
Broad Scopes 37
Going Global 40
Thoroughly Abuse the Type System 42
Turn Numbers into Secret Codes 43
Strings Are Magic—They Can Pretend to Be Any Type 44
Mix Things Up 46

Null—The Harbinger of Doom 47
Null Checks 47
Seeding Disaster 48

Chapter 4: Conditionals 51

Objectives 51
Prerequisites 51
Introduction 51
Forget the Alternatives 52
Or Else What? 52
The Normal and the Exceptional 54
Build a Ladder 58
Abuse Expressions 60
Tortuous Expressions 60
Not Being Not Non-negative . . . Not 63
Include Gaps and Overlaps 65
Thumbs Down! 66

Chapter 5: Loops 67

Objectives 67
Prerequisites 67
Introduction 68
Choose the Wrong Type 68
Collections 68
Ranges 70
Arbitrary Iterations 71
Have Fun with Infinite Loops 74
Citing the Masters 74
Taking Precautions 77

Make Inappropriate Exits 79
Break Out 80
Make 'em Looooong and Complex 82
Long Loops 82
Complex Loops 84

Chapter 6: Subroutines 87

Objectives 87
Prerequisites 87
Introduction 87
Super-Size Your Subroutines 88
Thumbs Down! 89
Put Up Barriers to Understanding 90
Bad Naming 90
High Complexity 91
Too Many Purposes 94
(Ab)use Parameters 96
The More the Merrier 97
Being Defensive 98
Surreptitious Subroutines 100
Screw with Return Values 101
Return of the Harbinger 101
Fun with Output Parameters 102

Chapter 7: Error Handling 107

Objectives 107
Prerequisites 107
Introduction 107
Assume Everything Will Always Go Well 108
Don't Check 108
Don't Assert 109
Don't Catch 112

Send Problems Down the Memory Hole 113
Disappearing Exceptions 113
Reporting Problems Is Doubleplusungood 114
Kick the Can Down the Road 116
Using Error Codes 117
Baffle and Bamboozle 118
Make a Mess 120
Cleaning Up and How Not to Do It 121

Chapter 8: Modules 125

Objectives 125
Prerequisites 125
Introduction 125
A Note on Terminology 126
Make Importing Messy 126
Import All the Things! 127
Clutter and Mess 128
Prevent Reuse 130
Shopping-List Subroutines 130
Mono-focused Modules 133
Create Strong Dependencies 135
Exposing Your Innards 136
The Public Face of a Module 140

Chapter 9: Classes and Objects 145

Objectives 145
Prerequisites 145
Introduction 146
Have Questionable Motives for Creating Classes 146
Data Classes 146
God Classes 148
Utility Classes 149

Make Objects Inflexible 150
Objects Obeying Orders 150
Rigid Relationships 153
Avoid Polymorphism 156
Thumbs Down! 158
Overuse and Abuse Inheritance 160
Going Deep 161
Quick and Dirty Reuse 163

Chapter 10: Testing 169

Objectives 169
Prerequisites 169
Introduction 170
Be Protective of Your Code 170
Keeping It to Yourself 171
Doing the Bare Minimum 171
Thwarting Efforts 176
Set Traps in Your Tests 177
Machine-specific Tests 178
Expansive Focus 180
Chaos 183

Chapter 11: Debugging 189

Objectives 189
Prerequisites 189
Introduction 189
Investigate Unsystematically 190
Guesswork 190
Biases 191
Chaos 192

Make Debugging Hard 194
Always Keep Your Mouth Shut 194
Keeping Records 196
Avoid Proper Fixes 198
The Hit 'n' Run Bug 198
Patch It Up 199
Bibliography **203**
Glossary **209**
Index **213**

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.