O'REILLY®

Becoming a Better Programmer

A HANDBOOK FOR PEOPLE WHO CARE ABOUT CODE

Pete Goodliffe

Becoming a Better Programmer

Pete Goodliffe

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Becoming a Better Programmer
by Pete Goodliffe

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Brian MacDonald
Production Editor: Melanie Yarbrough
Copyeditor: Jasmine Kwityn
Proofreader: Sonia Saruba
Indexer: Pete Goodliffe
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Pete Goodliffe

October 2014: First Edition

Revision History for the First Edition:

2014-10-01: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491905531* for release details.

ISBN: 978-1-491-90553-1

[LSI]

Table of Contents

Also by Pete Goodliffe xi

Introduction xiii

1. Care About the Code 1
Adopting the correct approach and attitude to code

Part I. you.write(code);

2. Keeping Up Appearances 7
Code presentation: layout and naming

3. Write Less Code! 17
Avoiding unnecessary lines of code

4. Improve Code by Removing It 29
Identifying and removing dead code

5. The Ghost of a Codebase Past 37
Learning from the code you wrote in the past

6. Navigating a Route 45
How to start working with unfamiliar code

7. Wallowing in Filth 55
Dealing with unpleasant, messy code

8. Don't Ignore That Error! 63
Healthy attitudes for error handling

9. **Expect the Unexpected** ... 69
Writing robust code that considers all possibilities

10. **Bug Hunting** ... 75
How to find and fix bugs

11. **Testing Times** ... 87
Developer testing: unit, integration, and system tests

12. **Coping with Complexity** ... 103
Designing code well, to avoid unnecessary complexity

13. **A Tale of Two Systems** ... 113
The consequences of good and bad design

Part II. Practice Makes Perfect

14. **Software Development Is** ... 131
What is this software stuff?

15. **Playing by the Rules** ... 141
Inventing rules that define your development team

16. **Keep It Simple** ... 145
Striving for simplicity in our software

17. **Use Your Brain** ... 153
Programmers are allowed and encouraged to use their brain; don't be stupid!

18. **Nothing Is Set in Stone** ... 157
No code is sacred, everything changes

19. **A Case for Code Reuse** ... 165
The healthy way to reuse code

20. **Effective Version Control** ... 171
Using version control well

21. **Getting One Past the Goalpost** 183
Working effectively with the QA team

22. **The Curious Case of the Frozen Code** 195
Code freeze: what it is, and whether it is necessary

23. **Please Release Me** 203
Making software releases

Part III. Getting Personal

24. **Live to Love to Learn** 215
How to learn effectively

25. **Test-Driven Developers** 225
Driving as an analogy to programming: how do we learn and pass the test?

26. **Relish the Challenge** 231
How to find the right challenges to stay motivated and keep your skills sharp

27. **Avoid Stagnation** 237
Preventing your programming skills from going stale

28. **The Ethical Programmer** 243
Ethical issues in the developer's life

29. **A Love for Languages** 253
Learning many programming languages and loving the ones you use

30. **Posturing Programmers** 261
Improving the programmer's health: posture, eye strain, and keeping your spirits up

Part IV. Getting Things Done

31. **Smarter, Not Harder** 273
Working effectively: avoiding unnecessary work and solving the right problems

32. **It's Done When It's Done** 283
Defining your programming tasks and knowing exactly when you're done

33. **This Time I've Got It** 291
Avoiding a narrow focus: find the best way to solve a problem

Part V. The People Pursuit

34. **People Power** 299
How to position yourself alongside excellent programmers, and how to work well in a team

35. **It's the Thought That Accounts** 305
Accountability: how it improves you and your work

36. **Speak Up!** 313
Communication skills for the software developer

37. **Many-festos** 323
Software manifestors: what and why?

38. **An Ode to Code** 329
A cautionary tale of software mismanagement

Epilogue 333

Index 337

Also by Pete Goodliffe

Introduction

You care about code. You're passionate about programming. You're the kind of developer who likes to craft truly great software. And you've picked up this book because you want to do it *even better*. Good call.

This book will help you.

The aim is to do exactly what it says on the cover: help you become a better programmer. But what does that mean exactly?

Pretty early in any programmer's career comes the realisation that there's more to being a great coder than a simple understanding of syntax and a mastery of basic design. The awesome programmers, those productive people who craft beautiful code and work effectively with other people, know far more. There are methods of working, attitudes, approaches, idioms, and techniques you learn over time that increase your effectiveness. There are useful social skills, and a whole pile of tribal knowledge to pick up.

And, of course, you need to learn syntax and design.

That is exactly what this book is about. It's a catalogue of useful techniques and approaches to the art and craft of programming that will help you become better.

I won't pretend that this is an exhaustive treatise. The field is vast. There's always more to learn, with new ground being claimed every day. These chapters are simply the fruit of more than 15 years of my work as a professional programmer. I've seen enough code, and made enough mistakes. I won't claim I'm an expert; I'm just well seasoned. If you can learn from the mistakes I've made and garner inspiration from what I've experienced, then you'll gain a leg up in your own development career.

What's Covered?

The topics covered in this book run the whole gamut of the software developer's life:

- Code-level concerns that affect how you write individual lines of code, as well as how you design your software modules.
- Practical techniques that will help you work better.
- Illustrations of effective attitudes and approaches to adopt that will help you become both super effective and well grounded.
- Procedural and organisational tricks and tips that will help you flourish whilst you are incarcerated in the software factory.

There's no particular language or industry bias here.

Who Should Read This?

You!

Whether you're an industry expert, a seasoned developer, a neophyte professional, or a hobbyist coder—this book will serve you.

Becoming a Better Programmer aims to help programmers at any level improve. That's a grand claim, but there's always something we can learn, and always room for improvement, no matter how experienced a programmer you are. Each chapter provides the opportunity to review your skills and work out practical ways to improve.

The only prerequisite for making use of this book is that you must *want* to become a better programmer.

The Structure

The information in this book is presented in a series of simple, self-contained chapters, each covering a single topic. If you're a traditionalist, you can read them in order from front to back. But feel free to read chapters in any order you want. Go straight to what seems most pertinent to you, if that makes you most happy.

The chapters are presented in five parts:

you.write(code);
: We start right at the bottom, at the codeface, where programmers feel most comfortable. This section reveals important code-writing techniques, and shows ways to write the best code possible. It covers code writing, code reading, code design, and mechanisms to write robust code.

Practice Makes Perfect
: Stepping back from the codeface, this part covers the important programming *practices* that help make you a better programmer. We'll see healthy attitudes and

approaches to the coding task, and sound techniques that will help you craft better code.

Getting Personal
: These chapters dig deep to build excellence into your personal programming life. We'll look at how to learn effectively, consider behaving ethically, find stimulating challenges, avoid stagnation, as well as improve physical well-being.

Getting Things Done
: These chapters talk about practical ways to *get things done:* to deliver code on time without getting sidetracked or delayed.

The People Pursuit
: Software development is a social activity. These chapters show how to work well with the other inhabitants of the software factory.

More important than the order you consume these chapters is how you approach the material. In order to actually improve, you have to apply what you read practically. The structure of each chapter is designed to help you with this.

In each chapter, the topic at hand is unpacked in flowing prose with stark clarity. You'll laugh; you'll cry; you'll wonder why. The conclusion of each chapter includes the following subsections:

Questions
: A series of questions for you to consider, and to answer. *Do not* skip these! They do not ask you to regurgitate the information you've just read. They are there to make you think deeper, beyond the original material, and to work out how the topic weaves into your existing experience.

See also
: Links to any related chapters in the book, with an explanation of how the chapters fit together.

Try this…
: Finally, each chapter is rounded off with a simple challenge. This is a specific task that will help you improve and apply the topic to your coding regimen.

Throughout each chapter, there are particularly important *key points*. They are highlighted so you don't miss them.

> **KEY ➤** This is a key point. Take heed.

As you work through each chapter, please do spend time considering the questions and the *Try this…* challenges. Don't gloss over them. They're an important part of *Becoming a Better Programmer*. If you just flick through the information in each chapter, then it will be just that: information. Hopefully interesting. No doubt informative. But unlikely to make you a much better programmer.

You need to be challenged, and absorb what you read to your programming skillset. These closing exercises won't take you too long. Honestly. And they will really help cement each chapter's theme in your mind.

A Note for Mentors

This book has been designed to work as a valuable tool for mentoring fellow programmers. You can use it one-on-one or in a study group.

The best approach to this material is *not* to methodically work through each section together. Instead, read a chapter separately, and then get together to discuss the contents. The questions really work as a springboard for discussion, so it's a good idea to start there.

Safari® Books Online

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/becoming_a_better_programmer*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Writing a book is a surprisingly large undertaking: one that tends to take over your life and suck other people into the maelstrom on the way. There are many people who have in some way contributed to the state of this book, from the very early drafts of this material right through until it became the complete tome that rests on your (potentially digital) bookshelf.

My wonderful wife, Bryony, has patiently supported (and put up with) me whilst my finger has been in this pie, alongside the many other pies my other fingers find. I love you, and I appreciate you very much. Alice and Amelia have provided many welcome distractions; you make life fun!

Some parts of this book originated in articles I wrote over the last few years. Steve Love, the esteemed editor of ACCU's *C Vu* magazine, has contributed valuable feedback on many of these, and his encouragement and sage opinion has always been appreciated. (If you don't know about ACCU, it is an awesome organisation for programmers who care about code.)

Many friends and colleagues have contributed valuable inspiration, feedback, and critique. These include my Akai family: Dave English, Will Augar, Łukasz Kozakiewicz, and Geoff Smith. Lisa Crispin and Jon Moore provided insight from the QA perspective, Greg Law taught me facts about bugs, whilst Seb Rose and Chris Oldwood offered much-appreciated and timely reviews.

The technical reviewers—Kevlin Henney, Richard Warburton, and Jim Brikman—provided much valuable feedback and helped shape the text you're reading. I am grateful for their expert input.

The excellent O'Reilly team of editors and production geniuses have worked hard on this book, and I'm grateful for their skillful attention. In particular, Mike Loukides and Brian MacDonald's early formative work helped shape the material considerably.

Lorna Ridley drew a chicken, single-handedly preventing this book from being fowl.

CHAPTER 1

Care About the Code

From caring comes courage.

— Lao Tzu

It doesn't take Sherlock Holmes to work out that good programmers write good code. Bad programmers… don't. They produce elephantine monstrosities that the rest of us have to clean up. You want to write the good stuff, right? You want to be a good programmer.

Good code doesn't pop out of thin air. It isn't something that happens by luck when the planets align. To get good code you have to work at it. Hard. And you'll only get good code if you actually *care* about good code.

KEY ➤ To write good code, you have to *care* about it. To become a better programmer you must invest time and effort.

Good programming is not born from mere technical competence. I've seen highly intellectual programmers who can produce intense and impressive algorithms, who know their language standard by heart, but who write the most awful code. It's painful to read, painful to use, and painful to modify. I've seen more humble programmers who stick to very simple code, but who write elegant and expressive programs that are a joy to work with.

Based on my years of experience in the software factory, I've concluded that the real difference between mediocre programmers and great programmers is this: *attitude*. Good programming lies in taking a professional approach, and *wanting* to write the best software you can, within the real-world constraints and pressures of the software factory.

The code to hell is paved with good intentions. To be an excellent programmer you have to rise above good intentions and actually *care* about the code—foster positive perspectives and develop healthy attitudes. Great code is carefully crafted by master artisans, not thoughtlessly hacked out by sloppy programmers or erected mysteriously by self-professed coding gurus.

You want to write good code. You want to be a good programmer. So, you care about the code. This means you act accordingly; for example:

- In any coding situation, you refuse to hack something that only *seems* to work. You strive to craft elegant code that is clearly correct (and has good tests to show that it is correct).
- You write code that *reveals intent* (that other programmers can easily pick up and understand), that is *maintainable* (that you, or other programmers, will be able to easily modify in the future), and that is *correct* (you take all steps possible to determine that you *have* solved the problem, not just made it look like the program works).
- You work well alongside other programmers. No programmer is an island. Few programmers work alone; most work in a team of programmers, either in a company environment or on an open source project. You consider other programmers, and construct code that others can read. You want the team to write the best software possible, rather than to make yourself look clever.
- Any time you touch a piece of code, you strive to leave it better than you found it (better structured, better tested, and more understandable…).
- You care about code and about programming, so you are constantly learning new languages, idioms, and techniques. But you only apply them when appropriate.

Fortunately, you're reading this book because you *do* care about code. It interests you. It's your passion. You like doing it well. Read on, and we'll see how to turn this code concern into practical action.

As you do this, never forget to have fun programming. Enjoy cutting code to solve tricky problems. Produce software that makes you proud.

KEY ➤ There is nothing wrong with an emotional response to code. Being proud of your great work, or disgusted at bad code, is healthy.

Questions

1. Do you *care* about code? How does this manifest in the work you produce?
2. Do you want to improve as a programmer? What areas do you think you need to work on the most?
3. If you don't care about code, why are you reading this book?!

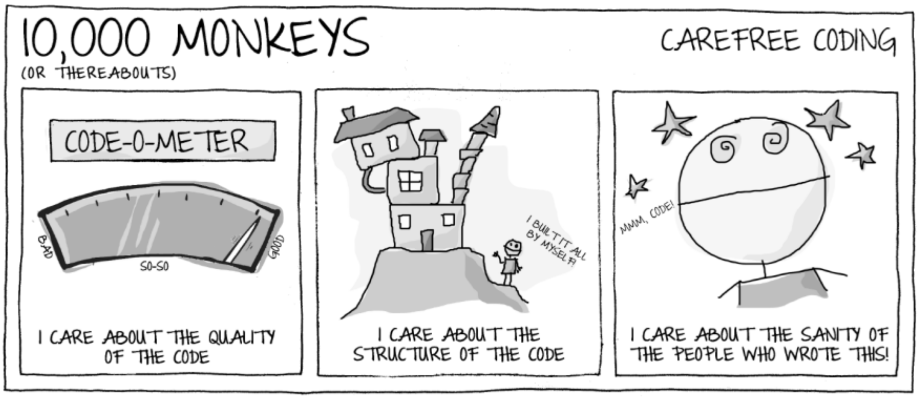
4. How accurate is the statement *Good programmers write good code. Bad programmers… don't*? Is it possible for good programmers to write bad code? How?

See also

- *Software Development Is…* What *is* this thing we care about?
- *Speak Up!* We care about working with good code. We should also care about working with good *people*.

Try this….

Commit now to improving your programming skills. Resolve to engage with what you read in this book, answer the questions, and attempt all of the *Try this…* challenges.


10,000 MONKEYS
(OR THEREABOUTS)
CAREFREE CODING
CODE-O-METER
BAD
SO-SO
GOOD
I CARE ABOUT THE QUALITY OF THE CODE
I BUILT IT ALL BY MYSELF!
I CARE ABOUT THE STRUCTURE OF THE CODE
MMM, CODE!
I CARE ABOUT THE SANITY OF THE PEOPLE WHO WROTE THIS!

PART I
you.write(code);

This first part deals with life on the front lines: our daily battle with code.

We'll look at low-level details that programmers revel in: how to write individual lines of code, how to improve sections of code, and how to plan a route into existing code. We'll also spend some time preparing for the unexpected: handling errors, writing robust code, and the black art of tracking down bugs. Finally, we look at the bigger picture: considering the design aspects of our software systems and investigating the technical and practical consequences of those designs.

CHAPTER 2

Keeping Up Appearances

Appearances are deceptive.

— Aesop

No one likes working with messy code. No one wants to wallow in a mire of jagged, inconsistent formatting, or battle with gibberish names. It's not fun. It's not productive. It's the programmer's purgatory.

We care about good code. And so we naturally care about code aesthetics; it is the most immediate determinant of how easy a section of code will be to work with. Practically every book about programming has a chapter on presentation. Oh look, this one does, too. Go figure.

Sadly, programmers care so much about code presentation that they end up bickering about it. This is the stuff that holy wars are made of. That, and which editor is best.[1] Tabs *versus* spaces. Brace positioning. Columns per line. Capitalisation. I've got my preferences. You have yours.

Godwin's law states that as any discussion on the Internet grows longer, the probability of a comparison to the Nazis or Hitler approaches one. *Goodliffe's law* (unveiled here) states that as any discussion about code layout grows, the probability of it descending into a fruitless argument approaches one.

Good programmers care deeply about good code presentation. But they rise above this kind of petty squabble. Let's act like grown-ups.

KEY ➤ Stop fighting over code layout. Adopt a healthy attitude to your code presentation.

1. Vim is. That is all.

Our myopic focus on layout is most clearly illustrated by the classic dysfunctional code review. When given a section of code, the tendency is to pick myriad holes in the presentation. (Especially if you only give it a cursory skim-read, then layout is all you'll pick up on.) You feel like you've made many useful comments. The design flaws will be completely overlooked because the position of a bracket is wrong. Indeed, it seems that the larger the code review, and the faster it's done, the more likely this blindness will strike.

Presentation Is Powerful

We can't pretend that code formatting is unimportant. But understand why it matters. A good code format is *not* the one you think looks prettiest. We do not lay out code in order to exercise our deep artistic leanings. (Can you hear the code-art critics? *Daaaahling, look at the wonderful Pre-Raphaelite framing on that nested switch statement.* Or: *you have to appreciate the poignant subtext of this method.* I think not.)

Good code is clear. It is consistent. The layout is almost invisible. Good presentation does not draw attention or distract; it serves only to reveal the code's intent. This helps programmers work with the code effectively. It reduces the effort required to maintain the code.

KEY ➤ Good code presentation reveals your code's intent. It is not an artistic endeavour.

Good presentation techniques are important, not for beauty's sake, but to *avoid mistakes* in your code. As an example, consider the following C snippet:

```
bool ok = thisCouldGoWrong();
if (!ok)
    fprintf(stderr, "Error: exiting...\n");
    exit(0);
```

You can see what the author intended here: `exit(0)` was only to be called when the test failed. But the presentation has hidden the real behaviour: the code will always `exit`. The layout choices have made the code a liability.[2]

Names have a similarly profound effect. Bad naming can be more than just distracting, it can be downright dangerous. Which of these is the bad name?

```
bool numberOfGreenWidgets;
string name;
void turnGreen();
```

2. This is not just an academic example to fill books! Serious real-life bugs stem from these kinds of mistakes. Apple's infamous 2014 *goto fail* security vulnerability in its SSL/TLS implementation was caused by exactly this kind of layout error.

The `numberOfGreenWidgets` is a variable, right? Clearly a counter is not represented by a boolean type. No; it's a trick question. They're all bad. The string does not actually hold a name, but the name of a colour; it is set by the `turnGreen()` function. So that variable name is misleading. And `turnGreen` was implemented thus:

```
void turnGreen()
{
    name = "yellow";
}
```

The names are all lies!

Is this a contrived example? Perhaps; but after a little careless maintenance, code can quickly end up in this state. What happens when you work with code like this? Bugs. Many, many bugs.

KEY ➤ We need good presentation to avoid making code errors. Not so we can create pretty ASCII art.

Encountering inconsistent layout and hodgepodge naming is a sure sign that code quality is not high. If the authors haven't looked after the layout, then they've probably taken no care over other vital quality issues (like good design, thorough testing, etc.).

It's About Communication

We write code for two audiences. First: for the compiler (or the language runtime). This beast is perfectly happy to read any old code slop and will turn it into an executable program the only way it knows how. It will impassionately do this without passing judgment on the quality of what you've fed it, nor on the style it was presented in. This is more a conversion exercise than any kind of code "reading."

The other, more important, audience is *other programmers*. We write code to be executed by a computer, but to be *read* by humans. This means:

- You right now, as you're writing it. The code has to be crystal clear so you don't make implementation mistakes.
- You, a few weeks (or months) later as you prepare the software for release.
- The other people on your team who have to integrate their work with this code.
- The maintenance programmer (which could be you or another programmer) years later, when investigating a bug in an old release.

Code that is hard to read is hard to work with. This is why we strive for clear, sympathetic, supporting presentation.

KEY ➤ Remember who you're writing code for: other people.

We've already seen that code can look pretty but obscure its intent. It can also look pretty, but be unreasonably hard to maintain. A great example of this is the "comment box." Some programmers like to present banner comments in pretty ASCII-art boxes:

```
/**************************************************
 * This is a pretty comment.                      *
 * Note that there are asterisks on the           *
 * righthand side of the box. Wow; it looks neat. *
 * Hope I never have to fix this tiypo.           *
 **************************************************/
```

It's cute, but it's not easy to maintain. If you want to change the comment text, you'll have to manually rework the right-hand line of comment markers. Frankly, this is a sadistic presentation style, and the people who choose it do not value the time and sanity of their colleagues. (Or they hope to make it so crushingly tedious to edit their comments that no one dare adjust their prose.)

Layout

If any man wishes to write a clear style, let him first be clear in his thoughts.

— Johann von Goethe

Code layout concerns include indentation, use of whitespace around operators, capitalisation, brace placement (be it K&R style, Allman, Whitesmith, or the like), and the age-old tabs *versus* spaces indent debate. In each of these areas there are a number of layout decisions you can make, and each choice has good reasons to commend it. As long as your layout choices enhance the structure of your code and help to reveal the intent, then they're good.

A quick glance at your code should reveal the shape and structure. Rather than argue about brace positioning, there are more important layout considerations, which we'll explore in the following sections.

Structure Well

Write your code like you write prose.

Break it up into chapters, paragraphs, and sentences. Bind the like things together; separate the different things. Functions are akin to chapters. Within each chapter may be a few distinct but related parts of code. Break them up into paragraphs by inserting blank lines between them. Do not insert blank lines unless there is a natural "paragraph" break. This technique helps to emphasise flow and structure.

For example:

```
void exampleFunction(int param)
{
    // We group things related to input
```

```
        param = sanitiseParamValue(param);
        doSomethingWithParam(param);

        // In a separate "paragraph" comes other work
        updateInternalInvariants();
        notifyOthersOfChange();
    }
```

The order of code revelation is important. Consider the reader: put the most important information first, not last. Ensure APIs read in a sensible order. Put what a reader cares about at the top of your class definition. That is, all public information comes before private information. Creation of an object comes before use of an object.

This grouping might be expressed in a class declaration like this:

```
class Example
{
public:
    Example();                  // lifetime management first
    ~Example();

    void doMostImportantThing(); // this starts a new "paragraph"
    void doSomethingRelated();   // each line here is like a sentence

    void somethingDifferent();   // this is another paragraph
    void aRelatedThing();

private:
    int privateStuffComesLast;
};
```

Prefer to write shorter code blocks. Don't write one function with five "paragraphs." Consider splitting this up into five functions, each with a well-chosen name.

Consistency

Avoid being precious about layout styles. Pick one. Use it consistently. It is best to be idiomatic—use what fits best with your language. Follow the style of standard libraries.

Write code using the same layout conventions as the rest of your team. Don't use your own style because you think it's prettier or better. If there is no consistency on your project then consider adopting a *coding standard* or *style guide*. This does not need to be a lengthy, draconian document; just a few agreed upon layout princples to pull the team together will suffice. In this situation, coding standards must be agreed on mutually, not enforced.

If you're working in a file that doesn't follow the layout conventions of the rest of your project, follow the layout conventions in that file.

Ensure that the entire team's IDEs and source code editors are configured the same way. Get the tab stop size the same. Set the brace position and comment layout options identically. Make the line ending options match. This is particularly important on cross-platform projects where very different development environments are used simultaneously. If you aren't diligent in this, then the source code will naturally become fractured and inconsistent; you will breed bad code.

War Story: The Whitespace Wars

I joined a project where the programmers had paid no attention to presentation. The code was messy, inconsistent, and unpleasant. I petitioned to introduce a coding standard.

All developers agreed that this was a good idea, and were willing to agree on conventions for naming, layout, and directory hierarchy. This was a huge step forward. The code began to grower neater.

However, there was one point we simply couldn't reach consensus on. You guessed it: *tabs or spaces*. Almost everyone preferred four-space indents. One guy swore tabs were superior. He argued, complained, and refused to change his coding style. (He probably still argues about it to this very day.)

Because we'd made some significant improvements, and in the interest of avoiding unnecessarily divisive arguments, we let this issue slide. We all used spaces. He used tabs.

The result was that the code remained frustrating and hard to work with. Editing was surprisingly inconsistent; sometimes your cursor moved one space at a time, sometimes it leapt around. Some tools would display the code reasonably well if you set an appropriate tab stop. Other tools (including our version control viewer and our online code review system) could not be adjusted and displayed ragged, awful looking code.

Names

When I use a word, Humpty Dumpty said, in a rather scornful tone, it means just what I choose it to mean—neither more nor less.

— Lewis Carroll

We name many things: variables, functions and methods, types (e.g., enumerations, classes), namespaces, and packages. Equally important are larger things, like files, projects, and programs. Public APIs (e.g., library interfaces or web service APIs) are perhaps the most significant things we choose names for, as "released" public APIs are most often set in stone and particularly hard to change.

A name conveys the identity of an object; it describes the thing, indicates its behaviour and intended use. A misnamed variable can be *very* confusing. A good name is descriptive, correct, and idiomatic.

You can only name something when you know *exactly* what it is. If you can't describe it clearly, or don't know what it will be used for, you simply can't name it well.

Avoid Redundancy

When naming, avoid redundancy and exploit context. Consider:

```
class WidgetList {
    public int numberOfWidgets() { ... }
};
```

The *numberOfWidgets* method name is unnecessarily long, repeating the word *Widget*. This makes the code harder, and more tedious, to read. Because this method returns the size of the list, it can simply be called `size()`. There will be no confusion, as the context of the enclosing class clearly defines what *size* means in this case.

Avoid redundant words.

I once worked on a project with a class called `DataObject`. That was a masterpiece of baffling, redundant naming.

Be Clear

Favour clarity over brevity. Names don't need to be short to save you key presses—you'll *read* the variable name far more times than you'll type it. But there is, however, a case for single-letter variable names: as counter variables in short loops, they tend to read clearly. Again, context matters!

Names don't need to be cryptic. The poster child for this is Hungarian Notation. It's not useful.

Baroque acronyms or "amusing" plays on words are not helpful.

Be Idiomatic

Prefer idiomatic names. Employ the capitalisation conventions most often used in your language. These are powerful conventions that you should only break with good reason. For example:

- In C, macros are usually given uppercase names.
- Capitalised names often denote types (e.g., a class), where uncapitalised names are reserved for methods and variables. This can be such a univerally accepted idiom that breaking it will render your code confusing.

Be Accurate

Ensure that your names are accurate. Don't call a type `WidgetSet` if it behaves like an array of widgets. The inaccurate name may cause the reader to make invalid assumptions about the behaviour or characteristics of the type.

Making Yourself Presentable

We come across badly formatted code all the time. Be careful how you work with it.

If you must "tidy it up" never alter presentation at the same time as making functional changes. Check in the presention change to source control as a separate step. *Then* alter the code's behaviour. It's confusing to see commits mixing the two things. The layout changes might mask mistakes in the functionality.

> **KEY ➤** Never alter presentation and behaviour at the same time. Make them separate version-controlled changes.

Don't feel you have to pick a layout style and stick with it faithfully for your entire life. Continually gather feedback from how layout choices affect how you work with code. Learn from the code you read. Adapt your presentation style as you gain experience.

Over my career, I have slowly migrated my coding style, moving ever towards a more consistent and easier to modify layout.

From time to time, every project considers running automated layout tools over the source tree, or adding them as a pre-commit hook. This is always worth investigating, and rarely worth using. Such layout tools tend to be (understandably) simplistic, and are never able to deal with the subtlties of code structure in the real world.

Conclusion

Stop fighting about code presentation. Favour a common convention in your project, even if it's not your personal preferred layout style.

But do have an informed opinion on what constitutes a good layout style, and why. Continually learn and gain more experience from reading other code.

Strive for conistency and clarity in your code layout.

Questions

1. Should you alter layout of legacy code to match the company coding standard? Or is it better to leave it in the author's original style? Why?
2. How valuable are code reformatting tools? How much does this depend on the language you're using?

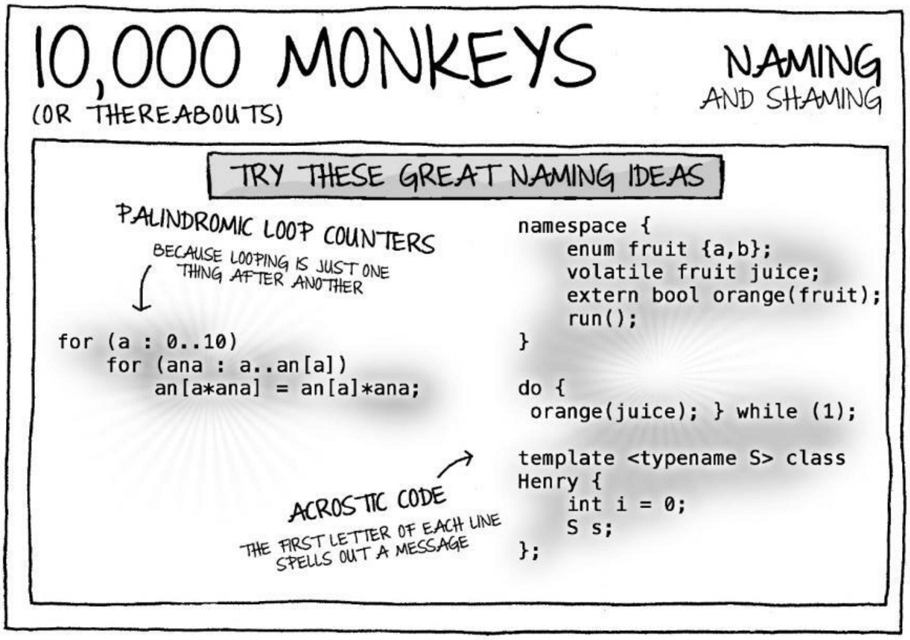
3. Which is more important: good code presentation or good code design?
4. How consistent is your current project's code? How can you improve this?
5. Tabs or spaces? Why? Does it matter?
6. Is it important to follow a language's layout and naming conventions? Or is it useful to adopt a different "house style" so you can differentiate your application code from the standard library?
7. Does our use of colourful syntax-highlighting code editors mean that there is less requirement for certain presentation concerns because the colour helps to reveal code structure?

See also

- *Speak Up!* Writing and presenting code is all about communication. This chapter discusses how a programmer communicates, in both code and the written word, and in speech.
- *The Ghost of a Codebase Past* Discusses how your programming style develops over time. Code presentation style is likely something you'll adapt as you gain experience.

Try this....

Review your layout preferences. Are they idiomatic, low ceremony, clear, and consistent? How can you improve them? Do you disagree with teammates about presentation? How can these differences be resolved?


10,000 MONKEYS
(OR THEREABOUTS)
NAMING
AND SHAMING
TRY THESE GREAT NAMING IDEAS
PALINDROMIC LOOP COUNTERS
BECAUSE LOOPING IS JUST ONE THING AFTER ANOTHER
ACROSTIC CODE
THE FIRST LETTER OF EACH LINE SPELLS OUT A MESSAGE

```
for (a : 0..10)
    for (ana : a..an[a])
        an[a*ana] = an[a]*ana;
```

```
namespace {
    enum fruit {a,b};
    volatile fruit juice;
    extern bool orange(fruit);
    run();
}

do {
 orange(juice); } while (1);

template <typename S> class
Henry {
    int i = 0;
    S s;
};
```

CHAPTER 3

Write Less Code!

A well-used minimum suffices for everything.

— Jules Verne
Around the World in Eighty Days

It's sad, but it's true: in our modern world there's just too much code.

I can cope with the fact that my car engine is controlled by a computer. There's obviously software cooking the food in my microwave. And it wouldn't surprise me if my genetically modified cucumbers had an embedded microcontroller in them. That's all fine; it's not what I'm obsessing about. I'm worried about all of the *unnecessary* code out there.

There's simply too much unnecessary code kicking around. Like weeds, these evil lines of code clog up our precious bytes of storage, obfuscate our revision control histories, stubbornly get in the way of our development, and use up precious code space, choking the good code around them.

Why is there so much unnecessary code?

Some people like the sound of their own voice. You've met them; you just can't shut them up. They're the kind of people you don't want to get stuck with at parties. *Yada yada yada.* Other people like their own code too much. They like it so much they write reams of it: `{ yada->yada.yada(); }`.

Or perhaps they're the programmers with misguided managers who judge progress by how many thousands of lines of code have been written a day.

Writing lots of code does *not* mean that you've written lots of software. Indeed, some code can actually negatively affect the amount of software you have—it gets in the way, causes faults, and reduces the quality of the user experience. The programming equivalent of antimatter.

KEY ➤ Less code *can* mean more software.

Some of my best software improvement work has been by removing code. I fondly remember one time when I lopped thousands of lines of code out of a sprawling system, and replaced it with a mere 10 lines of code. What a wonderfully smug feeling of satisfaction. I suggest you try it some time.

Why Should We Care?

So why is this phenomenon *bad*, rather than merely annoying?

There are many reasons why unnecessary code is the root of all evil. Here are a few bullet points:

- Writing a fresh line of code is the birth of a little life form. It will need to be lovingly nurtured into a useful and profitable member of software society before you can release a product using it.

  Over the life of your software system, that line of code needs maintenance. Each line of code costs a little. The more code you write, the higher the cost. The longer a line of code lives, the higher its cost. Clearly, unnecessary code needs to meet a timely demise before it bankrupts us.
- More code means there is more to read and more to understand—it makes our programs harder to comprehend. Unnecessary code can mask the purpose of a function, or hide small but important differences in otherwise similar code.
- The more code there is, the more work is required to make modifications—the program is harder to modify.
- Code harbours bugs. The more code you have, the more places there are for bugs to hide.
- Duplicated code is particularly pernicious; you can fix a bug in one copy of the code and, unbeknown to you, still have another 32 identical little bugs kicking around elsewhere.

Unnecessary code is nefarious. It comes in many guises: unused components, dead code, pointless comments, unnecessary verbosity, and so on. Let's look at some of these in detail.

Flappy Logic

A simple and common class of pointless code is the unnecessary use of conditional statements and tautological logic constructs. Flappy logic is the sign of a flappy mind. Or, at least, of a poor understanding of logic constructs. For example:

```
if (expression)
    return true;
```

```
else
    return false;
```

can more simply, and directly, be written:

```
return expression;
```

This is not only more compact, it is easier to read, and therefore easier to understand. It looks more like an English sentence, which greatly aids human readers. And do you know what? The compiler doesn't mind one bit.

Similarly, the verbose expression:

```
if (something == true)
{
    // ...
}
```

would read much better as:

```
if (something)
```

Now, these examples are clearly simplistic. In the wild we see much more elaborate constructs created; never underestimate the ability of a programmer to complicate the simple. Real-world code is riddled with things like this:

```
bool should_we_pick_bananas()
{
    if (gorilla_is_hungry())
    {
        if (bananas_are_ripe())
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    else
    {
        return false;
    }
}
```

which reduces neatly to the one-liner:

```
return gorilla_is_hungry() && bananas_are_ripe();
```

Cut through the waffle and say things clearly, but succinctly. Don't feel ashamed to know how your language works. It's not dirty, and you won't grow hairy palms. Knowing, and exploiting, the order in which expressions are evaluated saves a lot of unnecessary logic in conditional expressions. For example:

```
if ( a
     || (!a && b) )
{
    // what a complicated expression!
}
```

can simply be written:

```
if (a || b)
{
    // isn't that better?
    // didn't hurt, did it?
}
```

KEY ➤ Express code clearly and succinctly. Avoid unnecessarily long-winded statements.

Refactoring

The term *refactor* entered the programmer lexicon in the 1990s. It describes a particular kind of software modification, and was popularised by Martin Fowler's book, *Refactoring: Improving the Design of Existing Code.*[1]

The term, in my experience, is frequently misused.

It specifically describes a change made to the structure of existing code (i.e., its *factoring*) that *does not* change its exhibited behaviour. It's that last part that's often forgotten. A refactor is *only* a refactor if it is a transformation of the source code, preserving behaviour. An "improvement" that changes how the program reacts (no matter how subtly) is not a refactor; it's an improvement. A "tidy-up" that adjusts the UI is not a refactor; it's a tidy-up.

We refactor to increase the readability of the code, to improve the internal structure, to make the code more maintainable, and—most often—to prepare the code for some later functional enhancement.

There are catalogues of simple refactorings which can be applied in sequence to the code. Many language's IDEs provide automated support for these. Such transforms include: *Extract Class* and *Extract Method*, which break up functionality into better logic pieces, and *Rename Method* and *Pull Up/Pull Down*, which help move it to the right place.

Proper refactoring requires discipline and is greatly simplified by a good suite of unit tests that cover the code in question. These help to prove that any transformation has indeed preserved behaviour.

1. Martin Fowler, *Refactoring: Improving the Design of Existing Code* (Boston: Addison-Wesley, 1999).

Duplication

Unnecessary code duplication is evil. We mostly see this crime perpetrated through the application of *cut-and-paste* programming: when a lazy programmer chooses not to factor repeated code sections into a common function, but physically copies it from one place to another in their editor. Sloppy. The sin is compounded when the code is pasted with minor changes.

When you duplicate code, you hide the repeated structure, and you copy all of the existing bugs. Even if you repair one instance of the code, there will be a queue of identical bugs ready to bite you another day. Refactor duplicated code sections into a single function. If there are similar code sections with slight differences, capture the differences in one function with a configuration parameter.

KEY ➤ Do not copy code sections. Factor them into a common function. Use parameters to express any differences.

This is commonly known as the *DRY* principle: *Don't Repeat Yourself!* We aim for "DRY" code, without unnecessary redundancy. However, be aware that factoring similar code into a shared function introduces tight coupling between those sections of code. They both now rely on a shared interface; if you change that interface, both sections of code must be adjusted. In many situations this is perfectly appropriate; however, it's not always a desirable outcome, and can cause more problems in the long run than the duplication—so DRY your code responsibly!

Not all code duplication is malicious or the fault of lazy programmers. Duplication can happen by accident too, by someone reinventing a wheel that they didn't know existed. Or it can happen by constructing a new function when a perfectly acceptable third-party library already exists. This is bad because the existent library is far more likely to be correct and debugged already. Using common libraries saves you effort, and shields you from a world of potential faults.

There are also microcode-level duplication patterns. For example:

```
if (foo) do something();
if (foo) do_something_else()
if (foo) do_more();
```

could all be neatly wrapped in a single `if` statement. Multiple loops can usually be reduced to a single loop. For example, the following code:

```
for (int a = 0; a < MAX; ++a)
{
    // do something
}
// make hot buttered toast
for (int a = 0; a < MAX; ++a)
{
        // do something else
    }
```

probably boils down to:

```
    for (int a = 0; a < MAX; ++a)
    {
        // do something
        // do something else
    }
    // make hot buttered toast
```

if the making of hot buttered toast doesn't depend on either loop. Not only is this simpler to read and understand, it's likely to perform better, too, because only one loop needs to be run. Also consider redundant duplicated conditionals:

```
    if (foo)
    {
        if (foo && some_other_reason)
        {
            // the 2nd check for foo was redundant
        }
    }
```

You probably wouldn't write that on purpose, but after a bit of maintenance work a lot of code ends up with sloppy structure like that.

KEY ➤ If you spot duplication, remove it.

I was recently trying to debug a device driver that was structured with two main processing loops. Upon inspection, these loops were almost entirely identical, with some minor differences for the type of data they were processing. This fact was not immediately obvious because each loop was 300 lines (of very dense C code) long! It was tortuous and hard to follow. Each loop had seen a different set of bugfixes, and consequently the code was flaky and unpredictable. A little effort to factor the two loops into a single version halved the problem space immediately; I could then concentrate on one place to find and fix faults.

Dead Code

If you don't maintain it, your code can rot. And it can also die. *Dead code* is code that is never run, that can never be reached. That has no life. Tell your code to get a life, or get lost.

These examples both contain dead code sections that aren't immediately obvious if you quickly glance over them:

```
    if (size == 0)
    {
        // ... 20 lines of malarkey ...
        for (int n = 0; n < size; ++n)
        {
            // this code will never run
        }
        // ... 20 more lines of shenanigans ...
    }
```

and

```
void loop(char *str)
{
    size_t length = strlen(str);
    if (length == 0) return;
    for (size_t n = 0; n < length; n++)
    {
        if (str[n] == '\0')
        {
            // this code will never run
        }
    }
    if (length) return;
    // neither will this code
}
```

Other manifestations of dead code include:

- Functions that are never called
- Variables that are written but never read
- Parameters passed to an internal method that are never used
- Enums, structs, classes, or interfaces that are never used

Comments

Sadly, the world is riddled with awful code comments. You can't turn around in an editor without tripping over a few of them. It doesn't help that many corporate coding standards are a pile of rot, mandating the inclusion of millions of brain-dead comments.

Good code does *not* need reams of comments to prop it up, or to explain how it works. Careful choice of variable, function, and class names, and good structure should make your code entirely clear. Duplicating all of that information in a set of comments is unnecessary redundancy. And like any other form of duplication, it is also dangerous; it's far too easy to change one without changing the other.

Stupid, redundant comments range from the classic example of byte wastage:

```
++i; // increment i
```

to more subtle examples, where an algorithm is described just above it in the code:

```
// loop over all items, and add them up
int total = 0;
for (int n = 0; n < MAX; n++)
{
    total += items[n];
}
```

Very few algorithms when expressed in code are complex enough to justify that level of exposition. (But some *are*—learn the difference!) If an algorithm does need commentary, it may be better supplied by factoring the logic into a new, well-named function.

KEY ➤ Make sure that every comment adds value to the code. The code itself says *what* and *how*. A comment should explain *why*—but only if it's not already clear.

It's also common to enter a crufty codebase and see "old" code that has been surgically removed by commenting it out. Don't do this; it's the sign of someone who wasn't brave enough to perform the surgical extraction completely, or who didn't really understand what they were doing and thought that they might have to graft the code back in later. Remove code completely. You can always get it back afterwards from your source control system.

KEY ➤ Do not remove code by commenting it out. It confuses the reader and gets in the way.

Don't write comments describing what the code *used* to do; it doesn't matter anymore. Don't put comments at the end of code blocks or scopes; the code structure makes that clear. And don't write gratuitous ASCII art.

Verbosity

A lot of code is needlessly chatty. At the simplest end of the verbosity spectrum (which ranges from infra-redundant to ultra-voluble) is code like this:

```
bool is_valid(const char *str)
{
    if (str)
        return strcmp(str, "VALID") == 0;
    else
        return false;
}
```

It is quite wordy, and so it's relatively hard to see what the intent is. It can easily be rewritten:

```
bool is_valid(const char *str)
{
    return str && strcmp(str, "VALID") == 0;
}
```

Don't be afraid of the ternary operator if your language provides one; it really helps to reduce code clutter. Replace this kind of monstrosity:

```
public String getPath(URL url) {
    if (url == null) {
        return null;
    }
    else {
        return url.getPath();
    }
}
```

with:

```
public String getPath(URL url) {
    return url == null ? null : url.getPath();
}
```

C-style declarations (where all variables are declared at the top of a block, and used much, much later on) are now officially passé (unless you're still forced to use officially defunct compiler technology). The world has moved on, and so should your code. Avoid writing this:

```
int a;
// ... 20 lines of C code ...
a = foo();
// what type was an "a" again?
```

Move variable declarations and definitions together, to reduce the effort required to understand the code, and reduce potential errors from uninitialised variables. In fact, sometimes these variables are pointless anyway. For example:

```
bool a;
int b;
a = fn1();
b = fn2();
if (a)
    foo(10, b);
else
    foo(5, b);
```

can easily become the less verbose (and, arguably clearer):

```
foo(fn1() ? 10 : 5, fn2());
```

Bad Design

Of course, unnecessary code is not just the product of low-level code mistakes or bad maintenance. It can be caused by higher-level design flaws.

Bad design may introduce many unnecessary communication paths between components—lots of extra data marshalling code for no apparent reason. The further data flows, the more likely it is to get corrupted en route.

Over time, code components become redundant, or can mutate from their original use to something quite different, leaving large sections of unused code. When this happens, don't be afraid to clear away all of the deadwood. Replace the old component with a simpler one that does all that is required.

Your design should consider whether off-the-shelf libraries already exist that solve your programming problems. Using these libraries will remove the need to write a whole load of unnecessary code. As a bonus, popular libraries will likely be robust, extensible, and well used.

Whitespace

Don't panic! I'm not going to attack whitespace (that is, spaces, tabs, and newlines). Whitespace is a good thing—do not be afraid to use it. Like a well-placed pause when reciting a poem, sensible use of whitespace helps to frame our code.

Use of whitespace is not usually misleading or unnecessary. But you can have too much of a good thing, and 20 newlines between functions probably is too much.

Consider, too, the use of parentheses to group logic constructs. Sometimes brackets help to clarify the logic even when they are not necessary to defeat operator precedence. Sometimes they are unnecessary and get in the way.

So What Do We Do?

To be fair, often such a buildup of code cruft isn't intentional. Few people set out to write deliberately laborious, duplicated, pointless code. (But there are some lazy programmers who continually take the low road rather than invest extra time to write great code.) Most frequently, we end up with these code problems as the legacy of code that has been maintained, extended, worked with, and debugged by many people over a large period of time.

So what do we do about it? We must take responsibility. Don't write unnecessary code, and when you work on "legacy" code, watch out for the warning signs. It's time to get militant. Reclaim our whitespace. Reduce the clutter. Spring clean. Redress the balance.

Pigs live in their own filth. Programmers needn't. Clean up after yourself. As you work on a piece of code, remove all of the unnecessary code that you encounter.

This is an example of how to follow Robert Martin's advice and honour "the Boy Scout Rule" in the coding world: Always leave the campground cleaner than you found it.[2]

KEY ➤ Every day, leave your code a little better than it was. Remove redundancy and duplication as you find it.

But take heed of this simple rule: make "tidying up" changes separately from other functional changes. This will ensure that it's clear in your source control system what's happened. Gratuitous structural changes mixed in with functional modifications are hard to follow. And if there is a bug then it's harder to work out whether it was due to your new functionality, or because of the structural improvement.

Conclusion

Software functionality does not correlate with the number of lines of code, or to the number of components in a system. More lines of code do not necessarily mean more software.

So if you don't need it, don't write it. Write less code, and find something more fun to do instead.

Questions

1. Do you naturally write succinct logical expressions? Are your succinct expressions so terse as to be incomprehensible?
2. Does the C-language-family's *ternary operator* (e.g., `condition ? true_value : false_value`) make expressions more or less readable? Why?
3. We should avoid *cut-and-paste* coding. How different does a section of code have to be before it is justifiable to not factor into a common function?
4. How can you spot and remove dead code?
5. Some coding standards mandate that every function is documented with specially formatted code comments. Is this useful? Or is it an unnecessary burden introducing a load of worthless extra comments?

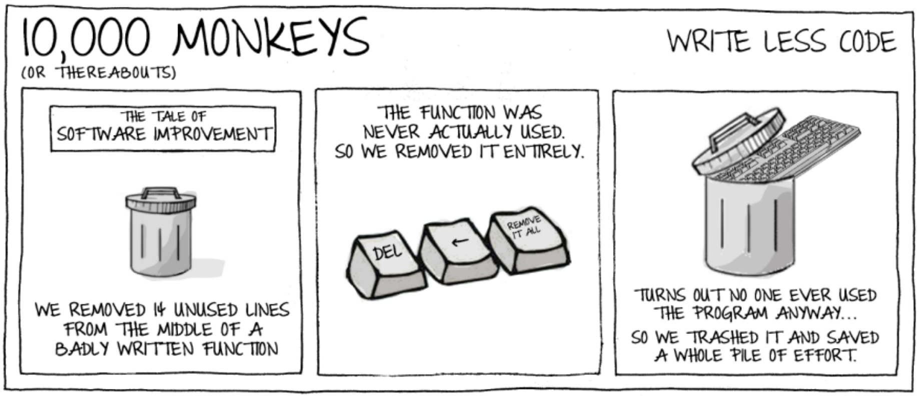
See also

- *Improve Code by Removing It* Describes techniques for identifying larger sections of redundant, dead code and removing it.

2. Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship* (Upper Saddle River, NJ: Prentice Hall, 2008).

Try this….

For the next few days, look at your code critically to identify any redundant, duplicated, or long-winded sections. Work on removing this unnecessary code.


10,000 MONKEYS
(OR THEREABOUTS)
WRITE LESS CODE
THE TALE OF SOFTWARE IMPROVEMENT
WE REMOVED 14 UNUSED LINES FROM THE MIDDLE OF A BADLY WRITTEN FUNCTION
THE FUNCTION WAS NEVER ACTUALLY USED. SO WE REMOVED IT ENTIRELY.
DEL
←
REMOVE IT ALL
TURNS OUT NO ONE EVER USED THE PROGRAM ANYWAY...
SO WE TRASHED IT AND SAVED A WHOLE PILE OF EFFORT.

CHAPTER 4

Improve Code by Removing It

We ascribe beauty to that which is simple;
which has no superfluous parts; which exactly answers its end...

— Ralph Waldo Emerson

Less is more. It's a trite maxim, but sometimes it really is true.

Some of the most exciting improvements I remember making to code involved *removing* vast chunks of it. Let me tell you, it's a good feeling.

War Story: No Need for the Code

As an Agile software development team, we'd been following the hallowed eXtreme Programming tenets, including *YAGNI*. That is, *You Aren't Gonna Need It*: a caution to not write unnecessary code—even code you *think* is going to be needed in future versions. Don't write it now if you don't need it now. Wait until you do have a genuine need.

This sounds like eminently sensible advice. And we'd all bought in to it.

But human nature being what it is, we fell short in a few places. At one point, I observed that the product was taking too long to execute certain tasks—simple tasks that should have been near instantaneous. This was because they had been over-implemented, festooned with extra bells and whistles that were not required, and littered with hooks for later extension. None of these things were being used, but at the time they each had seemed sensible additions.

So I simplified the code, improved the product performance, and reduced the level of global code entropy by simply removing all of the offending "features" from the codebase. Helpfully, my unit tests told me that I hadn't broken anything else during the operation. A simple and thoroughly satisfying experience.

KEY ➤ You can improve a system by adding new code. You can also improve a system by removing code.

Code Indulgence

So why did all that unnecessary code get written? Why did one programmer feel the need to write extra code, and how did it get past review or the pairing process?

It was almost certainly the programmers' indulging their own personal vices. Something like:

- It was a fun bit of extra code, and the programmer wanted to write it. *(Hint: Write code because it adds value, not because it amuses you, or you'd enjoy trying to write it.)*
- Someone thought it was a feature that would be needed in the future, so decided to code it now, whilst they thought about it. *(Hint: That isn't YAGNI. If you don't need it right now, don't write it right now.)*
- But it was only a small thing; not a massive "extra" feature. It was easier to just implement it now, rather than go back to the customer to see whether it was really required. *(Hint: It always takes longer to write and to maintain extra code. And the customer is actually quite approachable. A small extra bit of code snowballs over time to a large piece of work that needs maintenance.)*
- The programmer invented extra requirements that were not documented in the story that justified the extra feature. The requirement was actually bogus. *(Hint: Programmers do not set system requirements; the customer does.)*

Now, we had a well-understood lean development process, very good developers, and procedural checks in place to avoid this kind of thing. And unnecessary extra code still snuck in.

That's quite a surprise, isn't it?

It's Not Bad, It's Inevitable

Even if you can avoid adding unnecessary *new* features, dead pieces of code will still spring up naturally during your software development. Don't be embarrassed about it! They come from a number of unavoidable accidental sources, including:

- Features are removed from an application's user interface, but the backend support code is left in. It's never called again. Instant code necrosis. Often it's not removed "because we might need it in the future, and leaving it there isn't going to hurt anyone."

- Data types or classes that are no longer being used tend to stay put in the project. It's not easy to tell that you're removing the last reference to a class when working in a separate part of the project. You can also render *parts* of a class obsolete: for example, reworking methods so a member variable is no longer needed.
- Legacy product features are *rarely* removed. Even if your users no longer want them and will never use them again, removing product features never looks good. It would put a dent in the awesome list of tick-box features. So we incur perpetual product testing overhead for features that will never be used again.
- The maintenance of code over its lifetime causes sections of a function to not be executable. Loops may never iterate because code added above them negates an invariant, or conditional code blocks are never entered. The older a codebase gets, the more of this we see. C helpfully provides the preprocessor as a rich mechanism for writing non-executable spaghetti.
- Wizard-generated UI code inserts hooks that are frequently never used. If a developer accidentally double-clicks on a control, the wizard adds backend code, but the programmer never goes anywhere near the implementation. It's more work to remove these kinds of autogenerated code blocks than to simply ignore them and pretend that they don't exist.
- Many function return values are never used. We all know that it's morally reprehensible to ignore a function's error code, and we would *never* do that, would we? But many functions are written to *do something* and return a result that someone *might* find useful. Or might not. It's not an error code, just a small factoid. Why go through extra effort to calculate the return value, and write tests for it, if no one ever uses it?
- Much "debug" code is necrotic. A lot of support code is not needed once the initial implementation has been completed. It is unsightly scaffolding that hides the beautiful architecture underneath. It's not unusual to see reams of inactive diagnostic printouts and invariant checks, testing hook points, and the like, that will never be used again. They clutter up the code and make maintenance harder.

So What?

Does this really matter? Surely we should just accept that dead code is inevitable, and not worry about it too much if the project still works. What's the cost of unnecessary code?

- It is undeniable that unnecessary code, like any other code, requires maintenance over time. It costs time and money.
- Extra code also makes it harder to learn the project, and requires extra understanding and navigating.

- Classes with one million methods that may, or may not, be used are impenetrable and only encourage sloppy use rather than careful programming.
- Even if you buy the fastest machine money can buy, and the best compiler toolchain, dead code will slow down your builds, making you less productive.
- It is harder to refactor, simplify, or optimise your program when it is bogged down by zombie code.

Dead code won't kill you, but it will make your life harder than it needs to be.

KEY ➤ Remove dead code wherever possible. It gets in the way and slows you down.

Waking the Dead

How can you find dead code?

The best approach is to pay attention whilst working in the codebase. Be responsible for your actions, and ensure that you always clean up after your work. Regular code reviews do help to highlight dead code.

If you're serious about rooting out unused code sections, there are some great code coverage tools that will show you exactly where the problems are.[1] Good IDEs, especially when used with statically typed languages, can automatically highlight unused code. For public APIs, many IDEs have a "find references" feature that can show whether a function is ever called.

To identify dead features, you can instrument your product and gather metrics on what customers actually use. This is useful for making all sorts of business decisions, rather than just identifying unused code.

Surgical Extraction

There is no harm in removing dead code. Amputate it. It's not like you're throwing it away. Whenever you realise that you need an old feature again, it can easily be fetched from your version control system.

KEY ➤ It is safe to remove code that you *might* need in the future. You can always get it back from version control.

There is a counter argument to that simple (and true) view, though: how will a new recruit know that the removed code is available in version control if they don't know

1. You might even already have them—look at the warning options provided by your compiler.

that it existed in the first place? What's going to stop them writing their own (buggy or incomplete) version instead? This is a valid concern. But similarly, what would stop them rewriting their own version if they simply didn't notice the code fragment was already located elsewhere?

As in previous chapters, remember to remove dead code as a single step; do not conflate it in a version control check-in that also adds functionality. Always separate your "spring cleaning" work from other development tasks. This makes the version history clearer, and also makes revivifying removed code a breeze.

KEY ➤ Code cleanup should always be made in separate commits to functional changes.

Conclusion

Dead code happens in even the best codebases. The larger the project, the more dead code you'll have. It's not a sign of failure. But not doing something about it when you find dead code *is* a sign of failure. When you discover code that is not being used, or find a code path that cannot be executed, remove that unnecessary code.

When writing a new piece of code, don't creep the specification. Don't add "minor" features that you think are interesting, but no one has asked for. They'll be easy enough to add later, if they are required. Even if it *seems* like a good idea. Don't do it.

Questions

1. How can you identify "dead code" that is not run in your program?
2. If you temporarily remove code that is not currently required (but may be needed in the future) should you leave it commented out (so it is still visible) in the source tree, or just delete it completely (as it will be stored in the revision history)? Why?
3. Is the removal of legacy (unused) features always the right thing to do? Is there any inherent risk in removing sections of code? How can you determine the right time to remove unused features?
4. What percentage of your current project's codebase do you think is unnecessary? Does your team have a culture of adding things they *like* or that they *think* will be useful?

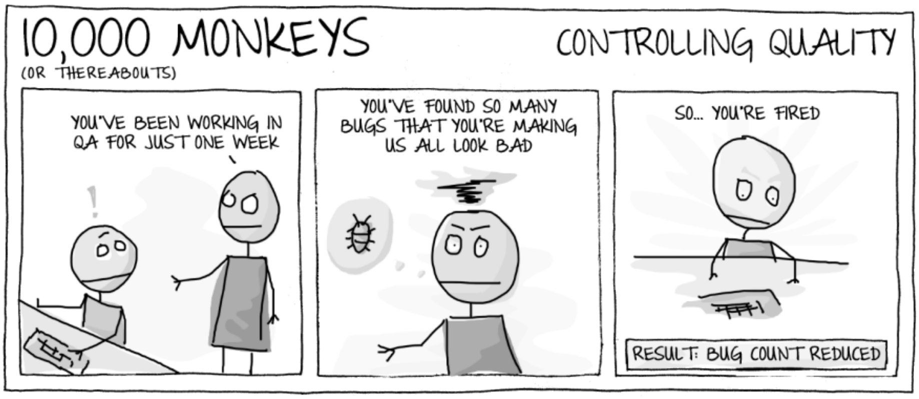
See also

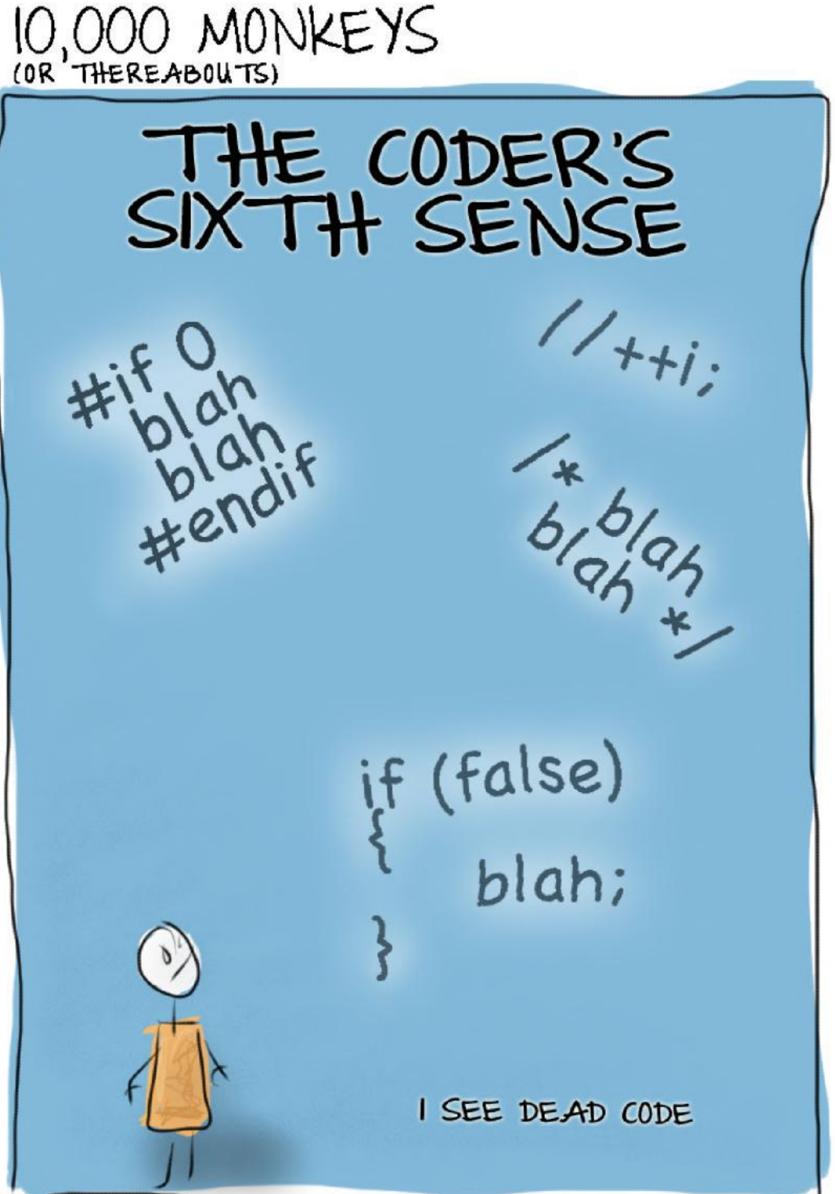
- *Write Less Code!* Talks about removing duplication at the micro level: whittling away unncessary lines of code.
- *Wallowing in Filth* How to navigate a route into problematic code so you can spot what needs to be removed.
- *Coping with Complexity* Removing dead code reduces complexity in your software.

- *Effective Version Control* Removing dead code does not mean it's lost forever. You can retrieve it from version control if you make a mistake.

Try this….

Look for dead and unnecessary code in the files you are working in. Remove it!


10,000 MONKEYS
(OR THEREABOUTS)
CONTROLLING QUALITY
YOU'VE BEEN WORKING IN QA FOR JUST ONE WEEK
YOU'VE FOUND SO MANY BUGS THAT YOU'RE MAKING US ALL LOOK BAD
SO... YOU'RE FIRED
RESULT: BUG COUNT REDUCED


10,000 MONKEYS
(OR THEREABOUTS)
THE CODER'S SIXTH SENSE
#if 0 blah blah #endif
//++i;
/* blah blah */
if (false) { blah; }
I SEE DEAD CODE

CHAPTER 5

The Ghost of a Codebase Past

I will live in the Past, the Present, and the Future.
The Spirits of all Three shall strive within me.
I will not shut out the lessons that they teach!

— Charles Dickens
A Christmas Carol

Nostalgia isn't what it used to be. And neither is your old code. Who knows what functional gremlins and typographical demons lurk in your ancient handiwork? You thought it was perfect when you wrote it—but cast a critical eye over your old code and you'll inevitably bring to light all manner of code gotchas.

Programmers, as a breed, strive to move onwards. We love to learn new and exciting techniques, to face fresh challenges, and to solve more interesting problems. It's natural. Considering the rapid turnover in the job market, and the average duration of programming contracts, it's hardly surprising that very few software developers stick with the same codebase for a prolonged period of time.

But what does this do to the code we produce? What kind of attitude does it foster in our work? I maintain that exceptional programmers are determined more by their attitude to the code they write and the way they write it, than by the actual code itself.

The average programmer tends not to maintain *their own* code for too long. Rather than roll around in our own filth, we move on to new pastures and roll around in *someone else's* filth. Nice. We even tend to let our own "pet projects" fall by the wayside as our interests evolve.

Of course, it's fun to complain about other people's poor code, but we easily forget how bad our own work was. And you'd never *intentionally* write bad code, would you?

Revisiting your old code can be an enlightening experience. It's like visiting an ageing, distant relative you don't see very often. You soon discover that you don't know them as well as you think. You've forgotten things about them, about their funny quirks and irritating ways. And you're surprised at how they've changed since you last saw them (perhaps, for the worst).

KEY ➤ Looking back at your older code will inform you about the improvement (or otherwise) in your coding skills.

Looking back at old code you've produced, you might shudder for a number of reasons.

Presentation

Many languages permit artistic interpretation in the indentation layout of code. Even though some languages have a de facto presentation style, there is still a large gamut of layout issues which you may find yourself exploring over time. Which ones stick depends on the conventions of your current project, or on your experiences after years of experimentation.

Different tribes of C++ programmers, for example, follow different presentation schemes. Some developers follow the standard library scheme:

```
struct standard_style_cpp
{
    int variable_name;
    bool method_name();
};
```

Some have more Java-esque leanings:

```
struct JavaStyleCpp
{
    int variableName;
    bool methodName();
};
```

And some follow a C# model:

```
struct CSharpStyleCpp
{
    int variableName;
    bool MethodName();
};
```

A simple difference, but it profoundly affects your code in several ways.

Another C++ example is the layout of member initialiser lists. One of my teams moved from this traditional scheme:

```
Foo::Foo(int param)
: member_one(1),
  member_two(param),
  member_three(42)
{
}
```

to a style that places the comma separators at the beginning of the following line, thus:

```
Foo::Foo(int param)
: member_one(1)
, member_two(param)
, member_three(42)
{
}
```

We found a number of advantages with the latter style (it's easier to "knock out" parts in the middle via preprocessor macros or comments, for example). This prefix-comma scheme can be employed in a number of layout situations (e.g., many kinds of lists: members, enumerations, base classes, and more), providing a nice consistent shape. There are also disadvantages, one of the major cited issues being that it's not as "common" as the former layout style. IDEs' default auto-layout also tends to fight with it.

I know over the years that my own presentation style has changed wildly, depending on the company I'm working for at the time.

As long as a style is employed consistently in your codebase, this is really a trivial concern and nothing to be embarrassed about. Individual coding styles rarely make much of a difference once you get used to them, but inconsistent coding styles in one project make everyone slower.

The State of the Art

Most languages have rapidly developed their in-built libraries. Over the years, the Java libraries have grown from a few hundred helpful classes to a veritable plethora of classes, with different skews of the library depending on the Java deployment target. Over C#'s revisions, its standard library has also burgeoned. As languages grow, their libraries accrete more features.

And as those libraries grow, some of the older parts become deprecated.

Such evolution (which is especially rapid early in a language's life) can unfortunately render your code anachronistic. Anyone reading your code for the first time might presume that you didn't understand the newer language or library features, when those features simply did not exist when the code was written.

For example, when C# added generics, the code you would have written like this:

```
ArrayList list = new ArrayList(); // untyped
list.Add("Foo");
list.Add(3); // oops!
```

with its inherent potential for bugs, would have become:

```
List<string> list = new List<string>();
list.Add("Foo");
list.Add(3); // compiler error - nice
```

There is a very similar Java example with surprisingly similar class names!

The state of the art moves much faster than your code. Especially your old, untended code.

Even the (relatively conservative) C++ library has grown considerably with each new revision. C++11 language features and library support have made much old C++ code look old-fashioned. The introduction of a language-supported threading model renders third-party thread libraries (often implemented with rather questionable APIs) redundant. The introduction of lambdas removes the need for a lot of verbose handwritten "trampoline" code. The range-based `for` helps remove a lot of syntactical trees so you can see the code-design wood. Once you start using these facilities, returning to older code without them feels like a retrograde step.

Idioms

Each language, with its unique set of language constructs and library facilities, has a particular "best practice" method of use. These are the *idioms* that experienced users adopt, the modes of use that have become honed and preferred over time.

These idioms are important. They are what experienced programmers expect to read; they are familiar shapes that enable you to focus on the overall code design rather than get bogged down in macro-level code concerns. They usually formalise patterns that avoid common mistakes or bugs.

It's perhaps most embarrassing to look back at old code, and see how un-idiomatic it is. If you now know more of the accepted idioms for the language you're working with, your old non-idiomatic code can look quite, quite wrong.

Many years ago, I worked with a team of C programmers moving (well, shuffling slowly) towards the (then) brave new world of C++. One of their initial additions to a new codebase was a `max` helper macro:

```
#define max(a,b) ((a)>(b)) ? (a) : (b))
// do you know why we have all those brackets?

void example()
{
    int a = 3, b = 10;
    int c = max(a, b);
}
```

In time, someone revisited that early code and, knowing more about C++, realised how bad it was. They rewrote it in the more idiomatic C++ shown here, which fixed some very subtle lurking bugs:

```
template <typename T>
inline T max(const T &a, const T &b)
```

```
{
    // Look mum! No brackets needed!
    return a > b ? a : b;
}

void better_example()
{
    int a = 3, b = 10;

    // this would have failed using the macro
    // because ++a would be evaluated twice
    int c = max(++a, b);
}
```

The original version also had another problem: wheel reinvention. The best solution is to just use the built-in `std::max` function that always existed. It's obvious in hindsight:

```
// don't declare any max function

void even_better_example()
{
    int a = 3, b = 10;
    int c = std::max(a,b);
}
```

This is the kind of thing you'd cringe about now, if you came back to it. But you had no idea about the right idiom back in the day.

That's a simple example, but as languages gain new features (e.g., lambdas) the kind of idiomatic code you'd write today may look very different from previous generations of the code.

Design Decisions

Did I *really* write that in Perl; what was I thinking?! Did I *really* use such a simplistic sorting algorithm? Did I *really* write all that code by hand, rather than just using a built-in library function? Did I *really* couple those classes together so unnecessarily? Could I *really* not have invented a cleaner API? Did I *really* leave resource management up to the client code? I can see many potential bugs and leaks lurking there!

As you learn more, you realise that there are better ways of formulating your design in code. This is the voice of experience. You make a few mistakes, read some different code, work with talented coders, and pretty soon find you have improved design skills.

Bugs

Perhaps this is the reason that drags you back to an old codebase. Sometimes coming back with fresh eyes uncovers obvious problems that you missed at the time. After you've been bitten by certain kinds of bugs (often those that the common idioms steer you away from) you naturally begin to see potential bugs in old code. It's the programmer's *sixth sense*.

Conclusion

No space of regret can make amends for one life's opportunity misused.

— Charles Dickens
A Christmas Carol

Looking back over your old code is like a code review for yourself. It's a valuable exercise; perhaps you should take a quick tour through some of your old work. Do you like the way you used to program? How much have you learnt since then?

Does this kind of thing actually *matter*? If your old code isn't perfect, but it works, should you do anything about it? Should you go back and "adjust" the code? Probably not—*if it ain't broke don't fix it*. Code does not rot, unless the world changes around it. Your bits and bytes don't degrade, so the meaning will likely stay constant. Occasionally a compiler or language upgrade or a third-party library update might "break" your old code, or perhaps a code change elsewhere will invalidate a presumption you made. But normally, the code will soldier on faithfully, even if it's not perfect.

It's important to appreciate how times have changed, how the programming world has moved on, and how your personal skills have improved over time. Finding old code that no longer feels "right" is a good thing: it shows that you have learnt and improved. Perhaps you don't have the opportunity to revise it now, but knowing where you've come from helps to shape where you're going in your coding career.

Like the Ghost of Christmas Past, there are interesting cautionary lessons to be learnt from our old code if you take the time to look at it.

Questions

1. How does your old code shape up in the modern world? If it doesn't look too bad, does that mean that you haven't learnt anything new recently?
2. How long have you been working in your primary language? How many revisions of the language standard or built-in library have been introduced in that time? What language features have been introduced that have shaped the style of the code you write?
3. Consider some of the common idioms you now naturally employ. How do they help you avoid errors?

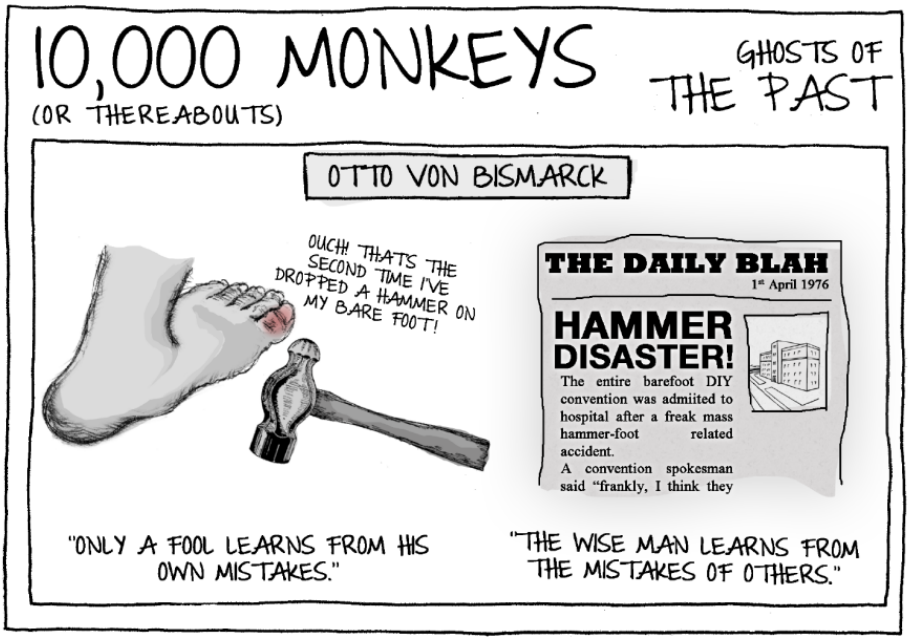
See also

- *Keeping Up Appearances* Contains more discussion of code layout.
- *Nothing Is Set in Stone* Code never stands still, nor does your understanding of it.

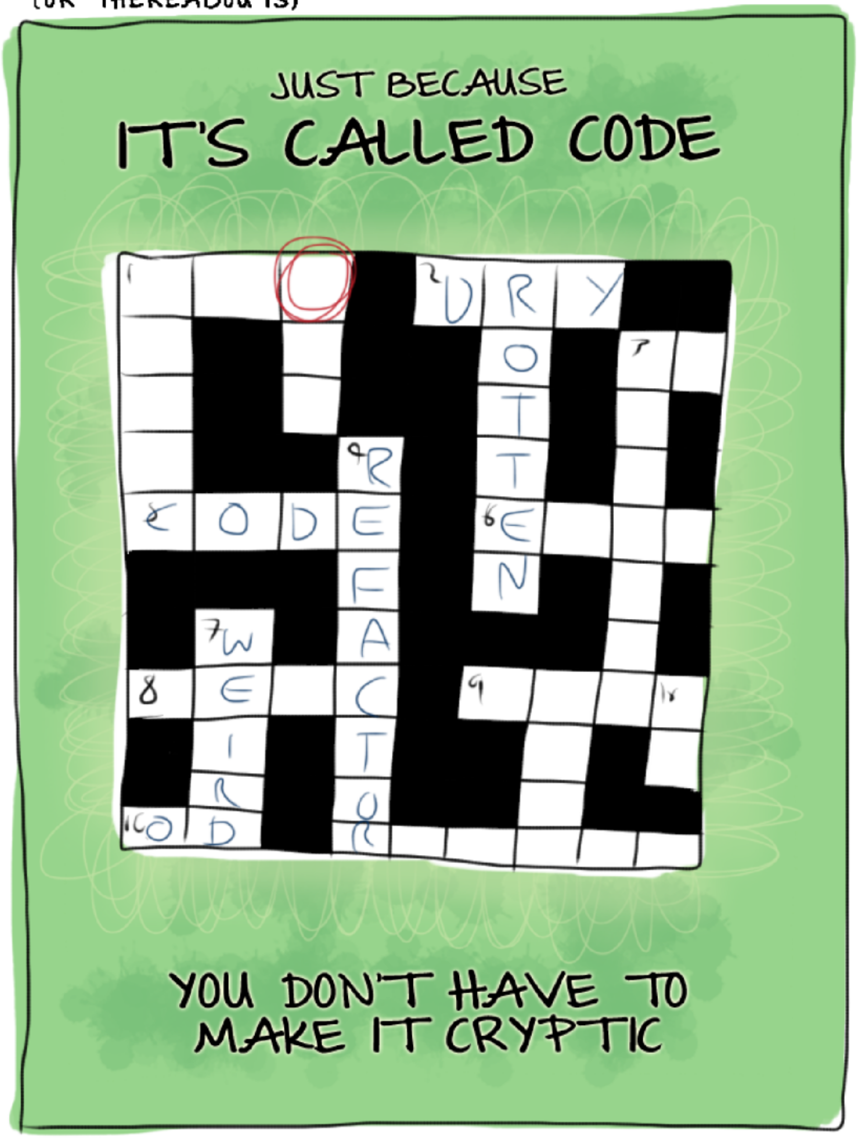
- *A Tale of Two Systems* An example of revisiting old code; learning from mistakes and appreciating successes.

Try this....

Take a quick tour through some of your old work. Do you like the way you used to program? How much have you learnt since then?


10,000 MONKEYS
(OR THEREABOUTS)
GHOSTS OF THE PAST
OTTO VON BISMARCK
OUCH! THATS THE SECOND TIME I'VE DROPPED A HAMMER ON MY BARE FOOT!
THE DAILY BLAH
1st April 1976
HAMMER DISASTER!
The entire barefoot DIY convention was admiited to hospital after a freak mass hammer-foot related accident.
A convention spokesman said "frankly, I think they
"ONLY A FOOL LEARNS FROM HIS OWN MISTAKES."
"THE WISE MAN LEARNS FROM THE MISTAKES OF OTHERS."

10,000 MONKEYS
(OR THEREABOUTS)



CHAPTER 6

Navigating a Route

...the Investigation of difficult Things by the Method of Analysis, ought ever to precede the Method of Composition.

— Sir Isaac Newton

A new recruit joined my development team. Our project, whilst not vast, was relatively large and contained a number of different areas. There was a lot to learn before he could become effective. How could he plot a route into the code? From a standing start, how could he rapidly become productive?

It's a common situation; one which we all face from time to time. If you don't, then you need to see more code and move on to new projects more often. (It's important not to get stale from working on one codebase with one team forever.)

Coming into any large existing codebase is hard. You have to rapidly:

- Discover where to start looking at the code
- Work out what each section of the code does, and how it achieves it
- Gauge the quality of the code
- Work out how to navigate around the system
- Understand the coding idioms, so your changes will fit in sympathetically
- Find the likely location of any functionality (and the consequent bugs caused by it)
- Understand the relationship of the code to its important satellite parts (e.g., its tests and documentation)

You need to learn this quickly, as you don't want your first changes to be too embarrassing, accidentally duplicate existing work, or break something elsewhere.

A Little Help from My Friends

My new colleague had a wonderful head start in this learning process. He joined an office with people who already knew the code, who could answer innumerable small questions about it, and point out where existing functionality could be found. This kind of help is simply invaluable.

If you are able to work alongside someone already versed in the code, then exploit this. Don't be afraid to ask questions. If you can, take opportunities to pair program and/or to get your changes reviewed.

> **KEY ➤** Your best route into code is to be led by someone who already knows the terrain. Don't be afraid to ask for help!

If you can't pester people nearby, don't fear; there may still be helpful people further afield. Look for online forums or mailing lists that contain helpful information and helpful people. There is often a healthy community that grows around popular open source projects.

The trick when asking for help is to always be polite, and to be grateful. Ask sensible, appropriate questions. "Can you do my homework for me?" is never going to get a good response. Always be prepared to help others out with information in return.

Employ common sense: make sure that you've Googled for an answer to your question first. It's simple politeness to not ask foolish questions that you could easily research yourself. You won't endear yourself to anyone if you continually ask basic questions and waste people's precious time. Like the boy who cried wolf and failed to get help when he really needed it, a series of mind-numbingly dumb questions will make you less likely to receive more complex help when you need it.

Look for Clues

If you are rooting in the murky depths of a software system without a personal guide, then you need to look for the clues that will orient you around the code.

These are good indicators:

Ease of getting the source

How easy is it to obtain the source?

Is it a single, simple checkout from version control that can be placed in any directory on your development machine? Or must you check out multiple separate parts, and install them in specific locations on your computer?

Hardcoded file paths are evil. They prohibit you from easily building different versions of the code.

KEY ➤ Healthy projects require a single checkout to obtain the whole codebase, and the code can be placed in *any* directory on your build machine. Do not rely on multiple checkout steps, or code in hardcoded locations.

As well as availabilty of the source code itself, consider availability of *information about* the code's health. Is there a CI (*continuous integration*) build server that continually ensures that all parts of the code build successfully? Are there published results of any automated tests?

Ease of building the code

This can be very telling. If it's hard to build the code, it's often hard to work with it.

Does the build depend on unusual tools that you'll have to install? (How up-to-date are those tools?)

How easy is it to build the code from scratch? Is there adequate and simple documentation in the code itself? Does the code build straight out of source control, or do you first have to manually perform many small configuration tweaks before it will build?

Does one simple, single step build the entire system, or does it require many individual build steps? Does the build process require manual intervention?[1] Can you work on a small part of the code, and only build that section, or must you rebuild the whole project repeatedly to work on a small component?

KEY ➤ A healthy build runs in one step, with no manual intervention during the build process.

How is a release build made? Is it the same process as the development builds, or do you have to follow a very different set of steps?

When the build runs, is it quiet? Or are there many, many warnings that may obscure more insidious problems?

Tests

Look for tests: unit tests, integration tests, end-to-end tests, and the like. Are there any? How much of the codebase is under test? Do the tests run automatically, or do they require an additional build step? How often are the tests run? How much coverage do they provide? Do they appear appropriate and well constructed, or are there just a few simple stubs to make it look like the code has test coverage?

There is an almost universal link here: code with a good suite of tests is usually also well factored, well thought out, and well designed. These tests act as a great route

1. A single, automatic build step means your build can be placed into a CI harness and run automatically.

into the code under test, helping you understand the code's interface and usage patterns. It's also a great place from which to start working on a bugfix (you can start by adding a simple, failing unit test—then fix that test, without breaking the others).

File structure
Look at the directory structure. Does it match the code shape? Does it clearly reveal the areas, subsystems, or layers of the code? Is it neat? Are third-party libraries neatly separated from the project code, or is it all messily intermingled?

Documentation
Look for the project documentation. Is there any? Is it well written? Is it up-to-date? Perhaps the documentation is written in the code itself using *NDoc*, *Javadoc*, *Doxygen*, or a similar system. How comprehensive and up-to-date does this documentation appear?

Static analysis
Run tools over the code to determine the health and to plot out the associations. There are some great source navigation tools available, and Doxygen can also produce very usable class diagrams and control flow diagrams.

Requirements
Are there any original project requirements documents or functional specifications? (In my experience, these often tend to bear little relation to the final product, but they are interesting historical documents nonetheless.) Is there a project wiki where common concepts are collected?

Project dependencies
Does the code use specific frameworks and third-party libraries? How much information do you need to know about them? You can't learn every aspect of all of them initially, especially because some libraries are huge (Boost, I'm looking at you). But it pays to get a feel for what facilities are provided for you, and where you can look for them.

Does the code make good use of the language's standard library? Or do many wheels get reinvented? Be wary of code with its own set of custom collection classes or homegrown thread primitives. System-supplied core code is more likely to be robust, well tested, and bug-free.

Code quality
Browse through the code to get a feel for the quality. Observe the amount and the quality of code comments. Is there much dead code—redundant code commented out but left to rot? Is the coding style consistent throughout?

It's hard to draw a conclusive opinion from a brief investigation like this, but you can quickly get a reasonable feel for a codebase from some basic reading.

Architecture

By now you should be able to get a reasonable feel for the shape and the modularisation of the system. Can you identify the main layers? Are the layers cleanly separated, or are they all rather interwoven? Is there a database layer? How sensible does it look? Can you see the schema? Is it sane? How does the app talk to the outside world? What is the GUI technology? The file I/O tech? The networking tech?

Ideally, the architecture of a system is a top-level concept that you learn before digging in too deeply. However, this is often not the case, and you *discover* the real architecture as you delve into the code.

KEY ➤ Often the *real* architecture of a system differs from the *ideal* design. Always trust the code, not the documentation.

Perform *software archaeology* on any code that looks questionable. Drill back through version control logs and "svn blame" (or the equivalent) to see the origin and evolution of some of the messes. Try to get a feel for the number of people who worked on the code in the past. How many of them are still on the team?

Learn by Doing

A woman needs a man like a fish needs a bicycle.

— Irina Dunn

You can read as many books as you like about the theory of riding a bicycle. You can study bicycles, take them apart, reassemble them, investigate the physics and engineering behind them. But you may as well be learning to ride a fish. Until you get on a bicycle, put your feet on the pedals and try to ride it for real, you'll never advance. You'll learn more by falling off a few times than from days of reading about how to balance.

It's the same with code.

Reading code will only get you so far. You can only really learn a codebase by getting on it, by trying to ride it, by making mistakes and falling off. Don't let inactivity prevent you from moving on. Don't erect an intellectual barrier to prevent you from working on the code.

I've seen plenty of great programmers initially paralysed through their own lack of confidence in their understanding.

Stuff that. Jump in. Boldly. Modify the code.

KEY ➤ The best way to learn code is to modify it. Then learn from your mistakes.

So what should you modify?

As you are learning the code, look for places where you can immediately make a benefit, but that will minimise the chances you'll break something (or write embarrassing code).

Aim for anything that will take you around the system.

Low-Hanging Fruit

Try some simple, small things, like tracking down a minor bug that has a very direct correlation to an event you can start hunting from (e.g., a GUI activity). Start with a small, repeatable, low-risk fault report, rather than a meaty intermittent nightmare.

Inspect the Code

Run the codebase through some code validators (like *Lint*, *Fortify*, *Cppcheck*, *FxCop*, *ReSharper*, or the like). Look to see if compiler warnings have been disabled; re-enable them, and fix the messages. This will teach you the code structure and give you a clue about the code quality.

Fixing this kind of thing is often not tricky, but very worthwhile; a great introduction. It often gets you around most of the code quickly. This kind of nonfunctional code change teaches you how things fit together and about what lives where. It gives you a great feel for the diligence of the existing developers, and highlights which parts of the code are the most worrisome and will require extra care.

Study, Then Act

Study a small piece of code. Critique it. Determine if there are weak spots. Refactor it. Mercilessly. Name variables correctly. Turn sprawling code sections into smaller well-named functions.

A few such exercises will give you a good feel for how malleable the code is and how yielding to fixes and modifications. (I've seen codebases that really fought back against refactoring).

Be wary: writing code is easier than reading it. Many programmers, rather than putting in the effort to read and understand existing code, prefer to say "it's ugly" and rewrite it. This certainly helps them get a deep understanding of the code, but at the expense of lots of unnecessary code churn, wasted time, and in all likelihood, new bugs.

Test First

Look at the tests. Work out how to add a new unit test, and how to add a new test file to the suite. How do the tests get run?

A great trick is to try adding a single, one-line, failing test. Does the test suite immediately fail? This *smoke test* proves that the tests are not actively being ignored.

Do the tests serve to illustrate how each component works? Do they illustrate the interface points well?

Housekeeping

Do some spit-and-polish on the user interface. Make some simple UI improvements that don't affect core functionality, but do make the app more pleasant to use.

Tidy the source files: correct the directory hierarchy. Make it match the organisation in the IDE or project files.

Document What You Find

Does the code have any kind of top-level *README* documentation file explaining how to start working on it? If not, create one and include the things that you have learned so far.

Ask one of the more experienced programmers to review it. This will show how correct your knowledge is, and also help future newbies.

As you gain understanding of the system, maintain a layer diagram of the main sections of code. Keep it up-to-date as you learn. Do you discover that the system is well layered, with clear interfaces between each layer and no unnecessary coupling? Or do you find the sections of code are needlessly interconnected? Look for ways of introducing interfaces to bring about separation without changing the existing functionality.

If there are no architectural descriptions so far, yours can serve as the documentation that will lead the new recruit into the system.

Conclusion

Scientific investigations are a sort of warfare carried on in the closet or on the couch against all one's contemporaries and predecessors.

— Thomas Young

The more you exercise, the less pain you feel and the greater the benefit you receive. Coding is no different. The more you work on new codebases, the more you are able to pick up new code effectively.

Questions

1. Do you often enter new codebases? Do you find it easy to work your way around unfamiliar code? What are the common tools you use to investigate a project? What tools can you add to this arsenal?
2. Describe some strategies for adding new code to a system you don't understand fully yet. How can you put a firewall around the existing code to protect it (and you)?

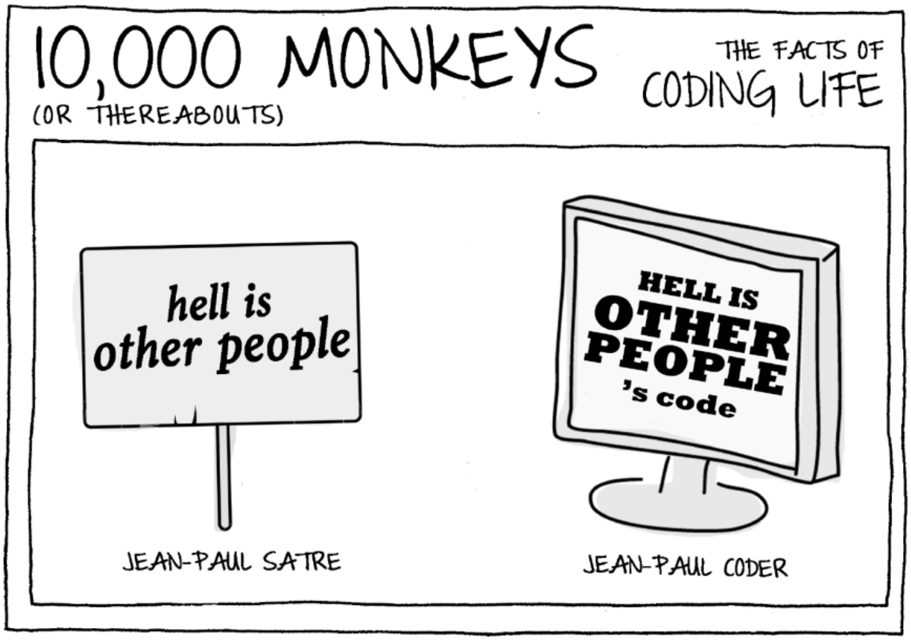
3. How can you make code easier for a new recruit to understand? What should you do now to improve the state of your current project?
4. Does the likely time you will spend working on the code in the future affect the effort and manner in which you learn existing code? Are you more likely to make a "quick and dirty" fix to code that you will no longer have to maintain, even though others will have to later on? Is this appropriate?

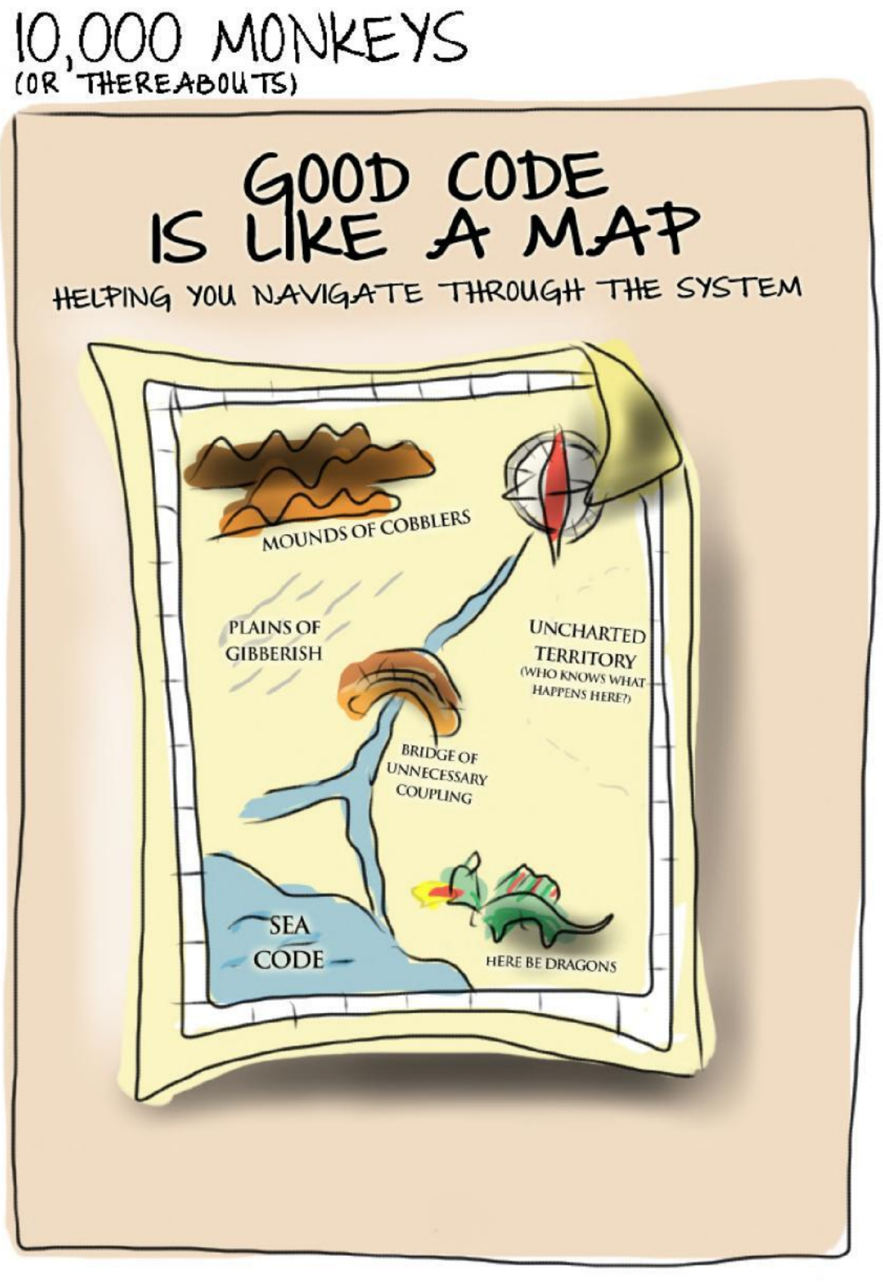
See also

- *Wallowing in Filth* How to gauge the quality of code, and make safe adjustments.
- *Live to Love to Learn* Learning a new codebase is like learning any new subject. These techniques will help.
- *Nothing is Set in Stone* Learn by doing: make changes to the code to understand it better.

Try this....

The next time you approach new code, plan a mindful route into it. Use these techniques to build a good understanding.




10,000 MONKEYS
(OR THEREABOUTS)
GOOD CODE IS LIKE A MAP
HELPING YOU NAVIGATE THROUGH THE SYSTEM
MOUNDS OF COBBLERS
PLAINS OF GIBBERISH
UNCHARTED TERRITORY (WHO KNOWS WHAT HAPPENS HERE?)
BRIDGE OF UNNECESSARY COUPLING
SEA CODE
HERE BE DRAGONS

CHAPTER 7

Wallowing in Filth

As a dog returns to its vomit, so fools repeat their folly.

— Psalms 26:11

We've all encountered it: *quicksand code*. You wade into it unawares, and pretty soon you get that sinking feeling. The code is dense, not malleable, and resists any effort made to move it. The more effort you put in, the deeper you get sucked in. It's the man-trap of the digital age.

How does the effective programmer approach code that is, to be polite, *not so great*? What are our strategies for *coping with crap*?

Don't panic, don your sand-proof trousers, and we'll wade in…

Smell the Signs

Some code is great, like fine art, or well-crafted poetry. It has discernible structure, recognisable cadences, well-paced meter, and a coherence and beauty that make it enjoyable to read and a pleasure to work with.

But, sadly, that is not always the case.

Some code is messy and unstructured: a slalom of gotos that hide any semblance of algorithm. Some is hard to read: with poor layout and shabby naming. Some code is cursed with an unnecessarily rigid structure: nasty coupling and poor cohesion. Some code has poor factoring: entwining UI code with low-level logic. Some code is riddled with duplication: making the project larger and more complex than it need be, whilst harbouring the exact same bug many times over. Some code commits "OO abuse": inheriting, for all the wrong reasons, tightly associating parts of code that have no real need to be bound. Some code sits like a pernicious cuckoo in the nest: C# written in the style of JavaScript.

Some code has even more insidious badness: brittle behaviour where a change in one place causes a seemingly unconnected module to fail—the very definition of *code chaos theory*. Some code suffers from poor threading behaviour: employing inappropriate thread primitives or exercising a total lack of understanding of the safe concurrent use of resources. This problem can be very hard to spot, reproduce, and diagnose, as it manifests so intermittently.

(I know I shouldn't moan, but sometimes I swear that programmers shouldn't be allowed to type the word *thread* without first obtaining a license to wield such a dangerous weapon.)

KEY ➤ Be prepared to encounter bad code. Fill your toolbox with sharp tools to deal with it.

To work effectively with alien code, you need to able to quickly spot these kinds of problems, and understand how to respond.

Wading into the Cesspit

The first step is to take a realistic survey of the coding crime scene. You arrive at the shores of new code. What *are* you wading into?

The code may have been given to you with a pre-attached stigma. No one wants to touch it because they know it's foul. Some quicksand code you discover yourself when you feel yourself sinking.

It's all too easy to pick up new code and dismiss it because it's not written in the style you'd prefer. Is it really dire work? Is it truly *quicksand code*, or is it merely unfamiliar? Don't make snap judgments about the code, or the authors who produced it, until you've spent some time investigating.

Take care not to make this personal.

Understand that few people set out to write shoddy code. Some filthy code was simply written by a less capable programmer. Or by a capable programmer on a bad day. Once you learn a new technique or pick up a team's preferred idiom, code that seemed perfectly fine a month ago is an embarrassing mess now and requires refactoring.

You can't expect any code, even your own, to be perfect.

KEY ➤ Silence the feeling of revulsion when you encounter "bad" code. Instead, look for ways to practically improve it.

The Survey Says…

We've already looked at techniques for navigating a new codebase in Chapter 6.

As you build a mental model of a new piece of code, you can begin to gauge its quality using benchmarks like:

- Are the external APIs clean and sensible?
- Are the types used well chosen, and well named?
- Is the code layout neat and consistent? (Whilst this is certainly not a guarantee of underlying code quality, I do find that inconsistent, messy code tends also to be poorly structured and hard to work with. Programmers who aim for high-quality, malleable code also tend to care about clean, clear presentation. But don't base your judgment on presentation alone.)
- Is the structure of cooperating objects simple and clear to see? Or does control flow unpredictably around the codebase?
- Can you easily determine where to find the code that produces a certain effect?

It can be hard to perform this initial survey. Maybe you don't know the technology involved, or the problem domain. You may not be familiar with coding style.

Consider employing *software archaeology* in your survey: mine your revision control system logs for hints about the quality. Determine: how old is this code? How old is it in relation to the entire project? How many people have worked on it over time? When was it last changed? Are any recent contributors still working on the project? Can you ask them for information about the code? How many bugs have been found and fixed in this area? Many bugfixes centered here indicates that the code is poor.

Working in the Sandpit

You've identified quicksand code, and you are now on the alert. You need a sound strategy to work with it.

What is the appropriate plan of attack?

- Should you repair the bad code?
- Should you perform the minimal adjustment necessary to solve your current problem, and then run away?
- Should you cut out the necrotic code and replace it with new, better work?

Gaze into your crystal ball. Often the right answer will be informed by your future plans. How long will you be working with this section of code? Knowing that you will be pitching camp and working here for a while influences the amount of investment you'll put in. Don't attempt a sweeping rewrite if you haven't the time.

Also, consider how frequently this code has been modified up to now. Financial advisors will tell you that *past performance is not an indicator of future results*. But often it is.

Invest your time wisely. This code might be unpleasant, but if it has been working adequately for years without tinkering, it is probably inappropriate to "tidy it up" now, especially if you're unlikely to need to make many more changes in the future.

KEY ➤ Pick your battles. Consider carefully whether you should invest time and effort in "tidying up" bad code. It may be pragmatic to leave it alone right now.

If you determine that it is not appropriate to embark on a massive code rework right now, that doesn't mean you are necessarily left to drift in a sea of sewage. You can wrestle back some control of the code by cleaning progressively.

Cleaning Up Messes

Whether you're digging in for the long haul, or just making a simple fix-and-run, heed Robert Martin's advice and follow the "the Boy Scout Rule": *Always leave the campground cleaner than you found it.* It might not be appropriate to make a sweeping improvement today, but that doesn't mean you can't make the world a slightly less awful place.

KEY ➤ Follow the Boy Scout Rule. Whenever you touch some code leave it *better* than you found it.

This can be a simple change: address inconstant layout, correct a misleading variable name, simplify a complex conditional clause, or split a long method into smaller, well-named sub-functions.

If you regularly visit a section of code, and each time leave it slightly better than it was, then before long you'll wind up with something that might be classified as *good*.

Making Adjustments

The single most important advice when working with messy code is this:

KEY ➤ Make code changes slowly, and carefully. Make one change at a time.

This is so important that I'd like you to stop, go back, and read it again.

There are many practical ways to follow this advice. Specifically:

- Do not change code layout whilst adjusting functionality. Tidy up the layout, if you must. Then commit your code. Only *then* make functional changes. (However, it's preferable to preserve the existing layout unless it's so bad that it gets in the way.)

- Do everything you can to ensure that your "tidying" preserves existing behaviour. Use trusted automated tools, or (if they are not available) review and inspect your changes carefully; get extra sets of eyeballs on it. This is the prime directive of *refactoring*: the well-known set of techniques for improving code structure.

  This goal can only be reached effectively if the code is wrapped in a sound set of unit tests. It is likely that messy code will not have any tests in place, so consider whether you should first write some tests to capture important code behaviour.

- Adjust the APIs that wrap the code without directly modifying the internal logic. Correct naming, parameter types, and ordering; generally introduce consistency. Perhaps introduce a new outer interface—the interface you wish that code had. Implement it in terms of the existing API. Then at a later date you can rework the code behind that interface.

Have courage in your ability to change the code. You have a safety net: source control. If you make a mistake, you can always go back in time and try again. It's probably not wasted effort, as you will have learnt about the code and its adaptability in doing so.

Sometimes it is worth boldly ripping out code in order to replace it. Badly maintained code that has seen no tidying or refactoring can be too painful and hard to correct piecemeal. There is an inherent danger in replacing code wholesale, though: the unreadable mess of special cases *might* be like that for a reason. Each bodge and code hack encodes an important piece of functionality that has been uncovered through bitter experience. Ignore these subtle behaviours at your peril.

An excellent book that deals with making appropriate changes in quicksand code is Micheal Feathers' *Working Effectively with Legacy Code*.[1] In it, he describes sound techniques to introduce seams into the code—places where you can introduce test points and most safely introduce sanity.

War Story: The Curious Case of the Container Code

There was a container class. It was central to our project. Internally, it was foul. The API stank, too. The original coder had worked hard to wreak code mischief. The bugs in it were hidden by the already confusing behaviour. Indeed, the confusing behaviour *was* a bug itself.

One of our programmers, a highly skilled developer, tried to refactor and repair this container. He kept the external interface intact, and improved many internal qualities: the correctness of the methods, the buggy object lifetime behaviour, performance, and code elegance.

1. Michael Feathers, *Working Effectively with Legacy Code* (Upper Saddle River, NJ: Prentice Hall, 2004).

He took out nasty, ugly, simplistic, stupid code and replaced it with the polar opposite. But in his effort to maintain the old API, this new version was internally far too contrived, more like a science project than useful code. It was hard to work with. Although it succinctly expressed the old (bizarre) behaviour, there was no room for extension.

We struggled to work with this new version, too. It had been a wasted effort.

Later on, another developer simplified the way we used the container: removing the weirder requirements, therefore simplifying the API. This was a relatively simple adjustment to the project. Inside the container, we removed swaths of code. The class was simpler, smaller, and easier to verify.

Sometimes you have to think laterally to see the right improvement.

Bad Code? Bad Programmers?

Yes, it's frustrating to be slowed down by bad code. The effective programmer does not only deal well with the bad code, but also with the people that wrote it. It is not helpful to apportion blame for code problems. People don't tend to purposefully write drivel.

KEY ➤ There is no need to apportion blame for "bad" code.

Perhaps the original author didn't understand the utility of code refactoring, or see a way to express the logic neatly. It's just as likely there are other similar things you do not yet understand. Perhaps they felt under pressure to work fast and had to cut corners (believing the lie that it helps you get there faster; it rarely does).

But of course, you know better.

If you can, *enjoy* the chance to tidy. It can be very rewarding to bring structure and sanity to a mess. Rather than see it as a tedious exercise, look at it as a chance to introduce higher quality.

Treat it as a lesson. Learn. How will you avoid repeating these same coding mistakes yourself?

Check your attitude as you make improvements. You might think that you know better than the original author. But do you always know better?

I've seen this story play out many times: a junior programmer "fixed" a more experienced programmer's work, with the commit message "refactored the code to look neater." The code indeed looked neater. But he had removed important functionality. The original author later reverted the change with the commit message: "refactored code back to working."

Questions

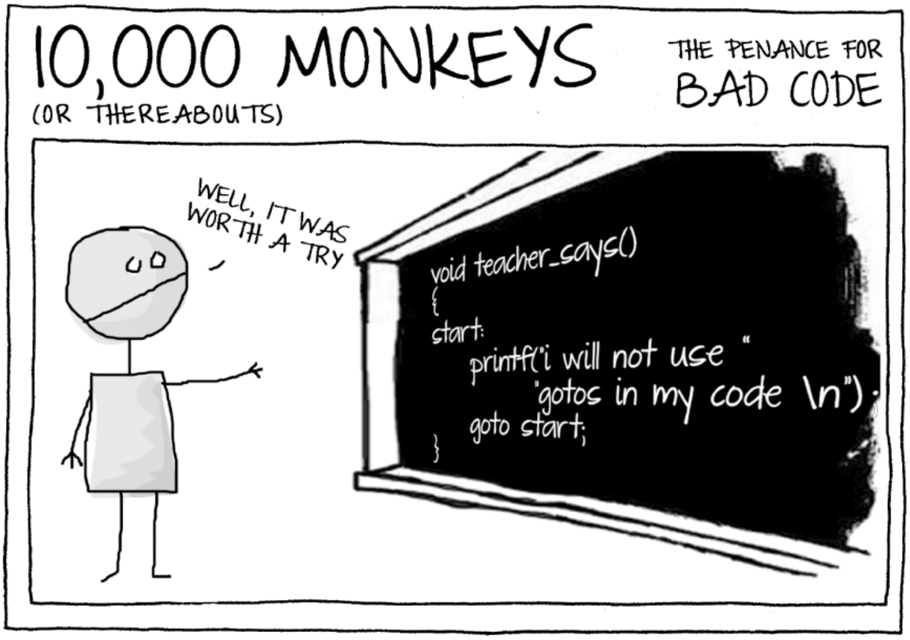
1. Why does code frequently get so messy?
2. How can we avoid this from happening in first place? Can we?
3. What are the advantages of making layout changes separately from code changes?
4. How many times have you been confronted with distasteful code? How often was this code *really* dire, rather than "not to your taste"?

See also

- *Navigating a Route* Techniques to familiarise yourself with a new codebase.
- *Improve Code by Removing It* Improve "filthy" programs by exorcising dead code.
- *An Ode to Code* An unnecessarily extreme reaction to bad code.

Try this....

Employ the Boy Scout Rule. Leave every piece of code you touch better, if even only fractionally.



```
void teacher_says()
{
start:
    printf("i will not use "
        "gotos in my code \n");
    goto start;
}
```