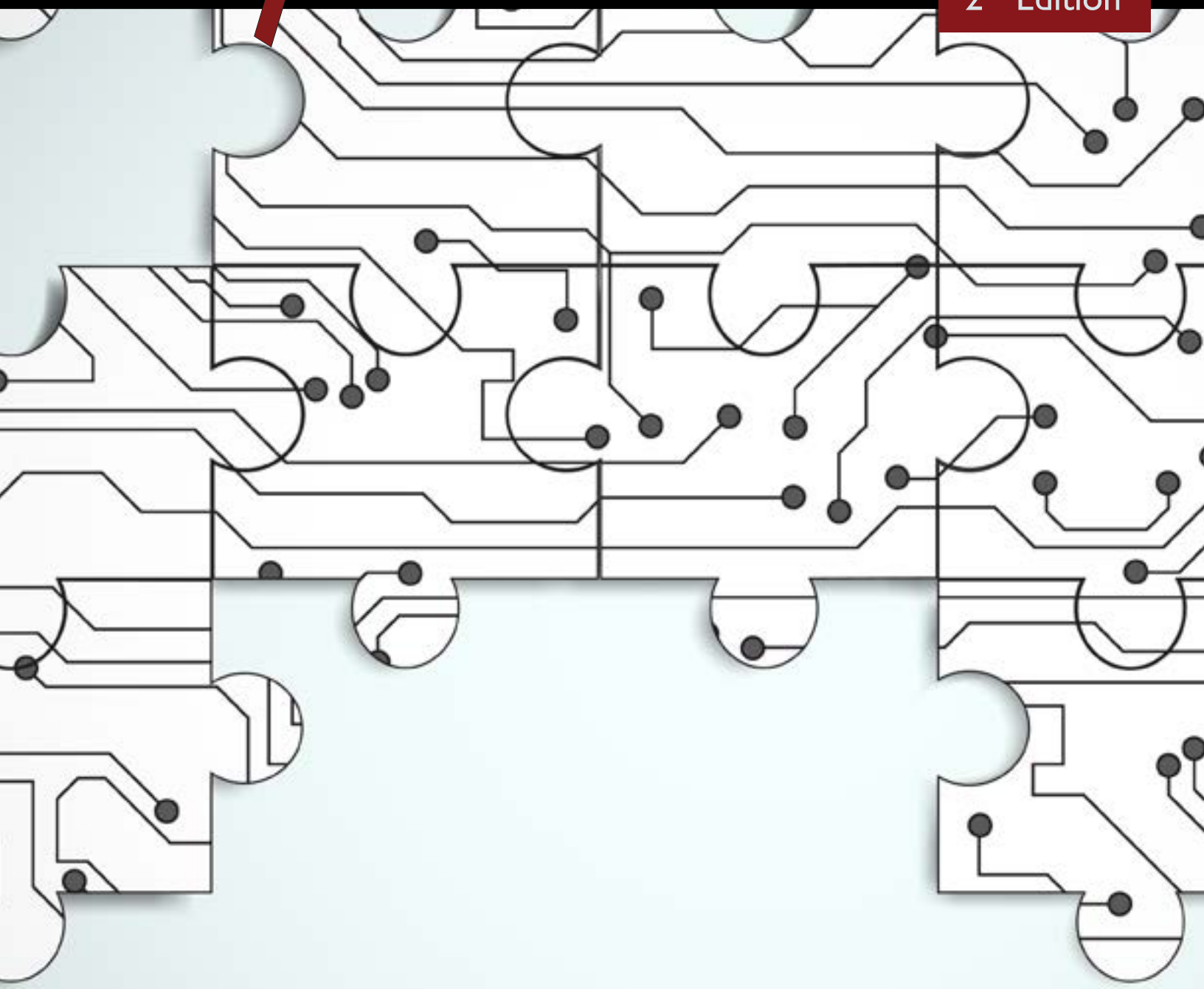
7 Mistakes That Cause *Fragile* Code

2nd Edition



JAMES KOPPEL

7 Mistakes That Cause *Fragile* Code

How do you create great code?

It's an under-studied question. Much has been written about how to program an app or build a fault-tolerant system or lead a team. Comparatively little has been written on building a codebase that programmers a year from now will enjoy working on. And yet there is nothing mysterious about writing good code. It's a skill that can be taught just as you can teach C or a web framework.

Everyone knows that code should be simple and have a clear design. But these ideas are deeper than most realize. I've watched students talk about "simplicity," and then use that to justify a change which actually makes their code more complex.

My goal in this booklet is to show you some counterintuitive ideas about what makes bad code, and to share some of the few resources that exist for helping you reach a higher level. Hopefully, some will already be familiar to you. But if I can show you one thing that's surprising and shifts the way you think about programming, then I've done my job.

Behold, the **7 Mistakes that Cause Fragile Code.**

1. Why checking errors is bad

You've probably seen functions that start with a long chain of if-statements. They make sure that names are not null, files exist, and init() has been called.

JamesKoppel Coaching

These checks are like a microwave that asks you not to input a negative number. Some are even like a microwave making sure it's not producing a nuclear reaction. In a good design, neither problem is possible.

Long chains of if-statements are as much of a smell as that microwave's instructions. When you design your APIs and data structures, you are choosing what states your program may be in. So design ones that don't contain error states.

Defensive code means there's something to defend against. So **don't prevent errors. Make them impossible.**

2. Refactoring is not boxing

A popular web framework has a class for saving E-mails to files. Its constructor has a long list of checks to make sure the file path is "valid." One of my students went to refactor it. When he came back, all of those checks were still there. He had moved them to a function called is_valid.

Similarly, I once asked a coworker to refactor some complicated UI code. He proudly demoed how he had packaged a piece of it into a new component. It required 8 parameters

Doing this is like cleaning a bedroom by grabbing random things off the floor and throwing them in a box. Easy to find stuff now, right?

I showed both how they could make their code simple. But it required more thought. It required finding the concepts of the system and expressing

them directly, instead of throwing code in a box.

Rewriting an essay is not about adding punctuation. It's thinking about what you want to say, and saying it differently. Refactoring is not about moving code into functions and modules. It's thinking about the concepts of your code, and understanding how it should be structured. To factor is to represent something as a composition of parts. To re-factor is to represent it differently.

So, when refactoring: **change the structure. Change the design.**

3. Why some decisions can't be undone

To understand why some decisions can't be undone, imagine trying to change the shape of electrical sockets in America. Perhaps you want them to be compatible with European sockets.

First, you design the new socket and plug.

Then you need to get every electronics company to update all their products.

And then you need every building owner to replace their sockets, and put a new plug on every old device.

Basically, it's not going to happen.

You can change the internals of your software however you like. But its boundary is like the socket and plug. The code you write makes assumptions about how to read and write data. Now these assumptions are embedded in the data, and your future code must deal with them. If your customers use your software for 20 years, then you'll be stuck dealing with 20 year-old decisions.

Say you work for Adobe. You want to redo the PDF format from scratch. And throw away the old code, because it's too complex.

You might have an easier time replacing all the sockets in America.

The boundaries of your program need to be designed exceptionally well, **because you're going to be living with them for a long time.**

4. Don't overthink the interior

When programmers develop taste, it's like turning on the lights in an abandoned cellar. There are rotten variable names and disgustingly complicated lines everywhere. Clean them all!

When these programmers review code by juniors, they start insisting on breaking up long lines and finding the shortest library call. They're climbing a hill towards better code, one step at a time.

But often, they're on the wrong hill. Simplifying a few lines of code might not help you add new features tomorrow. But having the right design will.
Your building may have the most disgusting and disorganized networking

closet in the world, but if no-one needs to go there for maintenance, it's not a problem.

But if the networking cables don't connect to it properly, you may have to tear down the whole building to fix it.

The amateur code reviewer focuses on the details of the submitted code. The wise code reviewer pays closer attention to **its effect on the rest of the codebase.**

5. Code quality is not (chiefly) about code

I give my students an exercise that involves understanding a new codebase. A few minutes later, I find most of them in some function, trying to read it line by line.

It's the wrong approach. They get overwhelmed.

Compare: I ask someone to tell me about the design of their kitchen.

Answer 1: It lets you do many things. If you want to chop vegetables, then you can start at the sink, reach above your head, take down a cutting board, take four steps to your left, put it down on the counter, and begin chopping. If you want to do the dishes, then start at the sink, open the dishwasher on your right, and begin doing the dishes, moving them from the sink to the dishwasher as you finish. If you want to bake something...

Answer 2: It's designed to make common tasks easy. On the left side, there is a counter with space for six cutting boards. The space between the counter area and sink area is wide enough for two people to pass at once. The dishwasher is placed next to the sink to allow high throughput between the two. The oven and stove are to the other side of the sink.

The second answer probably tells you much more about how to actually do things in this kitchen. And yet many programmers think more like the first answer, focusing on the how of software instead of the what.

If you think about good data structures first, then clean code will follow. The reverse is lipstick on a pig. David Parnas discovered this in 1972.

Or, as Linus Torvalds put it: **"Bad programmers worry about the code. Good programmers worry about data and their relationships."**

6. Getting stuck in an old design

One of my programmers came to me to report trouble. He wanted to import something from a file in a neighboring project. But this file contained many other things. Importing them would cause trouble, as often happens in C. What could he do?

It hadn't occurred to him that he could split that file in two.

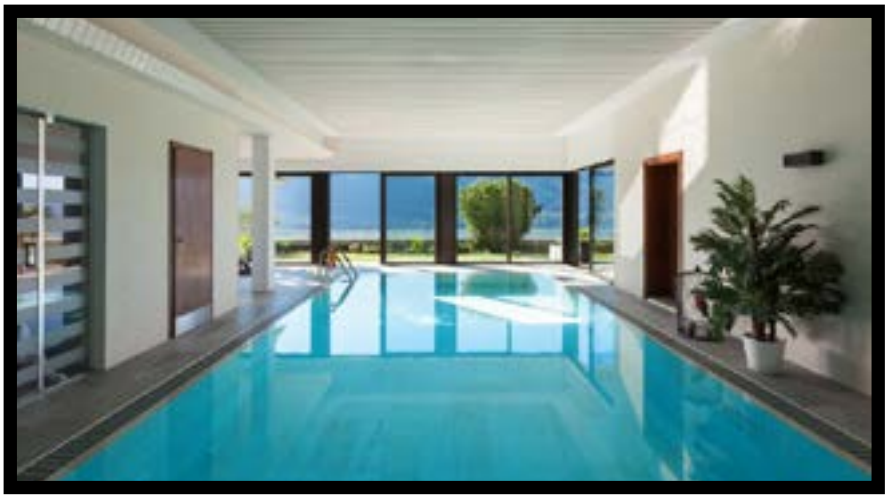
I've seen many programmers take the mindset that their work is just decoration on a stone tablet. They try to do their tasks by only writing new code, or only changing one module. Existing components don't quite do what they want. So they do more work on top to adapt them. Data conversions and conditionals abound.

Imagine building an indoor swimming pool. One way is to put up a plastic pool and run in a garden hose from outside. The other way is to, you know, actually

build an indoor swimming pool.

Like with the pool, it can take structural changes to make the codebase support what you're trying to do. But it pays off. The code becomes effortless.

And remember, in software, unlike in the real world, building a real swimming pool often isn't much harder than hacking it with a garden hose.



When you make non-local changes, suddenly the codebase is built exactly to support what you are doing. Once that work is done, the code looks effortless.

There is a design which allows the code you're writing to be beautiful and easy. If that design is not the current one, then change it.

So **don't be afraid. Make non-local changes.**

7. Are you missing half of what makes good tests?

There are two ways to write bad tests.

One is to not test enough, and fail to check that the code does what it's supposed to do. Most programmers know about this.

The other is to test too much. These tests look a bit like a literature exam that asks that students memorize Shakespeare instead of understanding the play. They succeed in checking expectations, but they're too specific. Expectations change, and they break.

Not understanding this leads to fragile tests. I've watched many programmers write tests by making inputs that cover every line of code, and check all details that are convenient. Why check that the bar method is called 3 times? Because it's called once directly and twice by the foo method; that's why.

I'm reminded of a study I once heard to try to improve student test scores by rewarding them with iPods and make-overs. The proposal stated its goal as "To distribute iPods and make-overs." And they succeeded.... at distributing iPods and make-overs.

This lesson is about more than writing tests. It's about one of the core teachings of software engineering: to think about the program's intention as something distinct from the program itself. So don't test that code does what it happens to do right now. Check that it does what it's supposed to do.

How to Improve at Software Design

If you ask someone this question today, you might get answers like:

7 Mistakes That Cause *Fragile* Code

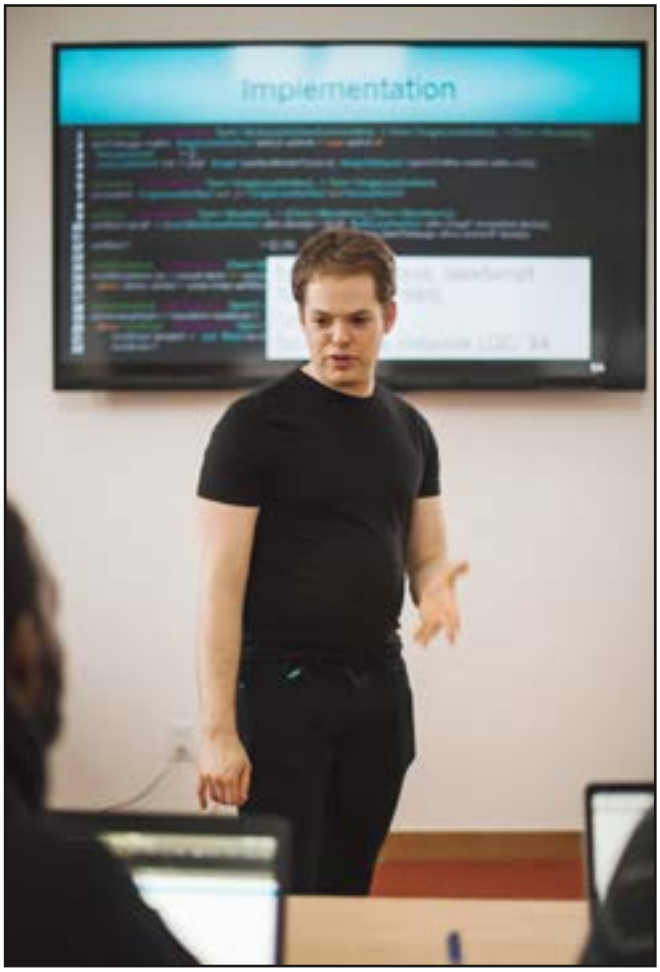
"Just work on projects."

"Find a good open-source project and read its code."

"Ask a coworker if they're willing to give you feedback."

Reading between the lines, these all boil down to "I don't know."

Contrast this with learning an instrument. Music teachers give students immediate feedback about mistakes. They teach drills for building technique. They assign pieces to challenge students in specific areas. This is deliberate practice.



No piano teacher says "The way to learn is to sit down and play." Indeed, they often must fix the bad habits of people who do exactly that.

My name is Jimmy Koppel, and my life mission is to improve the world's software quality. And for the last few years, I've been on a quest to make learning to write good code more like learning to play an instrument. Using my background in programming language theory and program analysis, I've distilled the complexities of designing a good system into a handful of clear

principles. I've helped struggling programmers become the stars of their team, and I've helped some already strong programmers become incredible.

Self-Learning

It takes a lot to become a great software engineer. How much can a short document help you?

A lot, if it changes how you learn. An athlete does not develop perfect technique by reading a long list of suggestions. They learn it by building body awareness, watching themselves on video, and learning to critique each movement.

I often find myself sitting down with an engineer, with only 15 minutes to help them. That's not enough to teach them to write great code, but it is enough to teach the programming equivalent of "watching yourself on video."

Here's the short version:

1. To practice design, you must design things without implementing them.
2. Every time a code change is hard, reflect on why.
3. Look at alternate designs for everything you build and use. Critique them.
4. For each line of code, ask what assumptions it's making.

5. Ask yourself: what are the core ideas of this software? How are they expressed in the code?

These tips should give you a nice boost at building insight. You can start practicing now.

But how badly do you want to be a great software engineer? How much are you willing to invest?

If you're like many career programmers, the answer is "a lot." So you'll want to do more than "watching yourself on video." Below, I've compiled some resources to help you learn faster.

Resources

There are lots of people who can teach you technologies like Spark, or processes like Scrum. But very little has been written on the general topic of how to write good code. So, most of these resources will be my own offerings.

From me

Newsletter: Arch-Engineer. Regular sharing of software design tips, articles, and research. If you are reading this, then you are probably already subscribed.

Blog: www.pathsensitive.com . Techniques for writing better code and learning faster, based on first principles.

Advanced Software Design Web Course and Weekend-Intensive: 6 deep principles for going from good to great. And a curriculum to make them part of you.

1-on-1 Coaching: Advanced knowledge, tailored practice, and intense feedback. For the truly ambitious.

From others

Here are a few of my favorite resources for learning software design at the advanced level. I have hundreds more in my curated list, which I frequently share on my newsletter and with my students.

I'm always interested in finding more good writing about software engineering. If you know of something, please share with me.

Blog

Joel on Software, www.joelonsoftware.com. A classic blog written by the founder of Trello and Stack Overflow. Multiple posts that contain insights on modularity, planning, and how old code affects the future.

Books: Software Engineering

The Art of Unix Programming, Eric S. Raymond. More about how to design programs than about Unix.

Understanding Software, Max Kanat-Alexander. Many high-level ideas about how to make code simpler.

The Architecture of Open-Source Applications, ed. Amy Brown and Greg Wilson. Contains lessons from many large systems.

Books: General Learning

The Art of Learning, Josh Waitzkin. Reflections on how to become the best in the world at something, by the world-famous chess player and Tai Chi champion.

Peak: Secrets from the New Science of Expertise, K. Anders Ericsson. Highlights from the psychology of skill development, written by the father of "deliberate practice."

Courses

Sandi Metz and Bob Martin both offer in-person courses on software design; both appear to be more beginner-focused than my own offerings. Occasionally, George Fairbanks teaches courses on software architecture and design. I have no first-hand experience with any of these.

Parting Words

We've talked about how to write better code. Let's take a step back and look at the bigger picture: why is this important?

Code quality is not just about aiding your company and career. It's about improving society. Software is often the slowest part to change of any system. Software maintenance is the reason it's hard for the EU to change privacy laws. 200 million lines of COBOL stand between the US government and any change to Social Security policy. By learning to write better code, you are literally making the world easier to change. Creating a world that's constantly advancing, and not stuck in the past.

And because I want your help building that world, I'm making one extra offer:

Open offer: I will have a call with any software engineer for up to 30 minutes to share resources and give advice. Completely free. To schedule, just E-mail me at jimmy@jameskoppelcoaching.com with the subject "Software engineering advice."

Best of luck in your software journey,

- **James Koppel**