Calm Code

Calm Code

Essays on writing software with intention

— M. Hollins

Contents

---

1. On Working in Wood

I started woodworking the year I turned thirty, and within a month it had changed how I wrote software.

The thing about working with wood is that it teaches you, immediately and without discussion, that you cannot rush. Wood has a grain. You can feel the grain through the chisel, and if you push against it you will tear the fibers and ruin the surface. You can rush in many crafts — bad cooking is still food, bad writing is still readable — but bad woodworking is, recognizably, an apology.

Code is not wood. Code has no grain. You can in fact push code against itself, sloppily, and it will run. The compiler does not care if your variable is named `data2`. The test suite does not know if your function is doing six things. The build will pass either way. This is why, I think, so much software is built with a hurry that no woodworker would tolerate. The material lets us.

But the people who use our software are not the compiler. They have grain. Their attention has grain. Their patience has grain. When we push against that grain — by shipping a confusing interface, by making them debug our weird behavior, by treating their data carelessly — we tear the surface of the relationship. They notice. The compiler doesn't, but they do.

I think the best engineers I've known are the ones who treat code as if it had grain. Who slow down at the joinery, who name things twice, who notice when something is fighting them and stop to listen to why. They produce less code, often. But the code they produce is the code I want to inherit.

The chisel and the call stack

There is a class of woodworking technique called *cutting with the grain*, and another called *cutting across the grain*. With the grain, you can take long, smooth cuts; the fibers separate cleanly. Across the grain, you must work in small, careful strokes, and even then the surface is rougher. Master craftsmen orient their work to take advantage of the grain wherever possible. They turn the board. They choose which face will be visible. They plan the cuts that matter and accept the rough cuts that don't.

This is good advice for software, too. Most codebases have a grain — a direction in which work flows naturally, where adding a feature feels like sliding a drawer closed, and another direction where adding a feature feels like prying a door off its hinges. Good engineers learn to feel that grain and work with it. Great engineers reshape the grain over time so the work flows even more cleanly.

The opposite — and we have all seen this — is the engineer who hammers through every change regardless of grain. The patches land. The features ship. But the codebase becomes a thing nobody wants to touch, and within two years it is being rewritten, and within four years the rewrite is being rewritten, and the person who reshapes the grain is somewhere else, doing the work that lasts.

What woodworking taught me about reviewing code

When you finish a piece of wood with hand tools, you spend more time looking than cutting. You bring the piece into the light, you run your finger along the joint, you set it down and walk away and come back, you pick it up again and look at it from the angle a viewer would.

I started reviewing code the same way after about a year of woodworking. I open the diff, but I do not start commenting. I read the change once, top to bottom, the way someone would read it three months from now, trying to figure out what it does. I look for the surfaces that are visible — the new function names, the new interfaces, the new error messages — and I imagine using them, the way you'd imagine running your hand along a tabletop. Only after that do I scroll back and notice the corners that need sanding.

The reviews I write are slower than they used to be. But I have not, in years, had a colleague tell me they felt rushed by my feedback. And I have started getting messages from junior engineers asking if I'd review their work specifically, which is the closest thing in our industry to someone asking you to make them a chair.

2. The Honest Backlog

I keep two backlogs. One is the team's — visible, shared, the source of weekly planning. The other is mine — a markdown file, on my laptop, called `actual.md`.

The team's backlog contains the things we have committed to. It is a record of decisions. The items in it have been discussed, prioritized, sized, and assigned. They are real work that will happen.

The actual backlog contains the things I think we *should* do but haven't decided to. Half-formed ideas. Bug suspicions I haven't verified. Refactors I'd do tomorrow if no one was watching. The note from three months ago that says "the auth flow is going to bite us by Q3."

The team's backlog is honest about commitment. The actual backlog is honest about everything else. They serve different audiences, and they make each other possible. Without the team backlog, I would drown in possibility. Without the actual backlog, I would be doing only what was assigned and waiting passively for the next surprise.

I review my actual.md every other Friday. About 1 in 10 items graduate to the team backlog. About 3 in 10 turn out to no longer be true (the suspicion was wrong, or someone else fixed it without telling me). The rest sit there, patient, accumulating context.

The first time something from `actual.md` graduated to the team backlog and then to a shipped feature — and turned out to matter — I closed the file with a sense of having paid attention to my own work in a way that no scrum ceremony had ever produced.

The grammar of a private backlog

A private backlog has a different grammar than a shared one. A shared backlog item must be readable by a stranger. It must define the work, the rationale, and ideally the acceptance criteria. It is a contract.

A private backlog item is a note to your future self. It can be a single word — "auth" — if you trust your future self to remember the context. It can be a question instead of a statement — "is the worker process leaking memory?" — because you haven't decided yet whether it's true. It can contradict another item. It can be wrong. Its only obligation is to capture the thought before you lose it.

Most of my actual.md items, on inspection, are questions. Not "fix the worker memory leak" but "is the worker memory growing?" Not "rewrite the user model" but "what would the user model look like if we started today?" Treating my private notes as questions instead of assertions has been one of the most freeing changes in how I work, because most of my "I should do X" thoughts turn out, on inspection, to be premature.

Why the team backlog needs guardrails

If your actual.md becomes the team backlog over time — if you just keep promoting items from one to the other — you have failed. The whole point is that they are different lists with different rules.

The team backlog should be small. Smaller than feels comfortable. A backlog with 400 items is not a backlog; it is a graveyard. Nobody reads it. Nobody acts on it. New items get added to the top, old items sink to the bottom, and the cycle repeats. A backlog with 30 items is something a team can hold in its head. Items get attention. The team can make collective decisions about priority. New items have to win their place by being better than something already on the list, which is healthy pressure.

The actual.md can be as large as it needs to be, because it never imposes obligations on anyone else. If you have 600 items there, that's fine — most of them will be wrong, and the ones that aren't will surface themselves over time through repeated attention.

3. A Defense of Boredom

Some of the best code I've ever written was written while I was bored.

This is not a fashionable thing to say. The dominant culture in software favors flow, focus, deep work. We treat boredom as a failure of engagement, a sign that we are working on the wrong thing or that something has gone wrong with our state of mind. We have books about how to avoid it.

But there is a kind of boredom that is the price of admission for the work that actually moves things forward. The boredom of reading the code you wrote six months ago carefully enough to understand why it is now wrong. The boredom of writing the tenth test case for an edge condition that probably won't happen. The boredom of staring at a stack trace for the third time because something is telling you that you missed it the first two times.

This boredom is not a failure mode. It is the texture of careful work. And the engineers who can sit with it are different from the engineers who can't.

The flight from boredom

I have watched a lot of engineers — myself included — flee boredom in real time. The flight usually goes like this:

1. You sit down to work on a bug. The bug is annoying.
2. You poke at it for ten minutes. The poking is not productive.
3. You feel restless.
4. You think, *I should check Slack*. You check Slack. There is something new.
5. You respond to the something new. It feels productive.
6. You think, *I should check the dashboard*. You check the dashboard. Nothing is on fire, but you check anyway.
7. You think, *I should refactor that other thing while I'm thinking about it.*
8. You start refactoring the other thing. It feels good.
9. Three hours later, the original bug is still there. The refactor is half-done. You feel busy but not productive.

I have done this. I have done it this week. The drug here is not Slack; the drug is *novelty*. When work is hard and slow, your brain offers you alternatives, and you take them because they feel good in the short term. The boredom is the cost of the work that matters. The novelty is the cost of avoiding it.

The discipline is not to never feel restless. It's to notice the restlessness and stay anyway. The work that pays off — the ten-test edge case, the careful read of the old code, the third look at the stack trace — happens in the minutes where you would much rather be doing something else.

Boredom as a tool

A subtler observation: boredom can be used as a tool, if you know it well enough.

When I am stuck on a problem and the obvious moves aren't working, I sometimes deliberately bore myself with something tangential. I'll read documentation for a library I'm not using. I'll trace through a part of the codebase I don't need to touch. I'll write notes about why the problem is hard, even though writing the notes doesn't solve it.

I do this because boredom — the controlled kind, where you're paying attention but the input isn't demanding much of you — is when my mind has the bandwidth to make connections it can't make under pressure. The answer to the hard problem often arrives ten minutes into reading something unrelated, as if it had been waiting for me to stop pushing.

This is not a substitute for focus. You cannot bore your way to a solution; you have to do the focused work too. But focus and boredom are not opposites. They are complementary modes of the same craft, and the engineers who have learned to use both are stronger than the ones who only know how to grind.

4. What Tests Are Actually For

The official story about tests is that they catch bugs. The official story is wrong, or at least, it's underspecified.

If catching bugs were the only point of tests, we could use a much weaker version of testing — just write the code, deploy it, fix the production failures, repeat. Some teams basically do this and it works for them. Their bug rates are higher but their throughput is, in some shallow sense, faster.

So if "catching bugs" is not enough to justify the cost of testing, what is? In my experience, three things, in increasing order of importance:

1. Tests catch bugs *during refactoring*. This is the thing tests are best at. When you change the shape of the code, you need a way to know whether you broke something behavior-visible. Tests give you that confidence. Without tests, refactoring becomes high-anxiety and most engineers stop doing it. The codebase calcifies.

2. Tests document the intended behavior of the system in a form that is *executable*. Comments lie because they don't run. Tests don't lie because they fail when the code drifts from them. A well-written test suite is the most reliable specification a codebase has.

3. Tests force the engineer who writes them to *think about how the code will be used* before the code is written. This is the deepest value. A function that is easy to test is usually a function that has clear inputs, clear outputs, and few hidden dependencies. Writing the test first, or even just imagining writing it, surfaces design problems before they are baked in.

The first two are obvious. The third is the one nobody talks about, and it is the one that pays for the cost of testing.

Test-first as design discipline

I used to be skeptical of test-driven development. It felt ceremonial — write the test, watch it fail, write the code, watch it pass, repeat. Compared to just writing the code, it seemed slow.

What I missed for a while was that the slowness *was* the point. Writing the test first forces you to think about the interface before you commit to the implementation. You have to decide what the function is called, what it takes, what it returns, what errors it might raise. You have to imagine using it from the outside.

Most of my worst design decisions, looking back, were made because I wrote the implementation first and then bolted a test on later. The test passed, but the interface was awkward — too many parameters, leaky abstractions, weird side effects — because I'd designed the interface from the inside-out instead of the outside-in.

I don't always write tests first now. But when I'm building something I know I'll have to live with for a while, I do. The fifteen minutes I spend writing a test before the code saves hours of work over the lifetime of that code, because the code ends up shaped in a way I can use.

What not to test

There is a counterpart belief — held by some engineers, including ones I respect — that *more* tests are always better. They aim for high coverage as a goal in itself. They write tests for every function, including trivial getters and obvious one-liners.

I think this is mostly counterproductive. Tests are code. Code has a maintenance cost. A test that asserts a getter returns the value you just set on it does not catch bugs; it duplicates the implementation in a form that breaks every time the implementation changes. You pay a maintenance cost for no real safety gain.

The tests that pay for themselves are the ones that:

- Test behavior, not implementation
- Test the boundaries of your system (inputs from the outside, outputs to the outside)
- Test the cases where multiple components interact in non-obvious ways
- Document an assumption that, if broken, would be a real bug

Everything else is, in my opinion, optional. Code coverage is a number that can be manipulated; system reliability is what actually matters, and you get there by testing the things that, if they break, will surprise you.

---

5. The Pull Request as Letter

A pull request is a letter you write to other people.

This is obvious in retrospect, but I had been writing PRs for eight years before I really internalized it. Before that, I treated the PR description as a chore — a paragraph at the top, a few bullet points, ship it. I had teammates who treated it the same way. Our PRs were perfunctory. Code review was hard. People skimmed because there was nothing in the PR to slow them down.

A good PR, I now believe, is closer in spirit to a letter than to a JIRA ticket. It has a recipient (your reviewer, your future self, anyone who looks at the git blame in two years). It has a purpose (to convince them that this change should be merged). It has a structure (context, problem, solution, trade-offs, what to look at). And it earns the reader's attention by being readable.

What a letter does that a perfunctory PR doesn't

When you write a letter, you start by establishing what the reader knows and what they don't. You make sure they have the context they need to understand the rest. You don't drop them into the middle of a complicated situation and expect them to figure it out.

Most bad PRs fail at this first step. They assume the reader has been in the contributor's head for the past three days. They reference issues by number without summarizing them. They include screenshots without explaining what to look for. They have titles like "fix bug in handler" that tell you almost nothing about what's actually changing.

A letter-quality PR opens differently. It opens with a one-paragraph summary that a stranger could read and understand. It includes the *why* — what user-visible problem this solves, or what engineering problem this unblocks. It includes any context that isn't obvious from the diff itself: prior conversations, related changes, known gotchas.

This takes more time than a perfunctory PR. It takes maybe ten extra minutes per PR. But the time you save on the review side — the questions that don't need to be asked, the back-and-forth that doesn't happen, the misunderstandings that don't arise — pays back the ten minutes many times over.

What to put in the description

The PRs I am proudest of tend to have descriptions structured like this:

1. **What this changes, in one sentence.** Plain English, no jargon.
2. **Why this change is necessary.** The user problem, the engineering problem, the strategic reason — whatever drove the work.
3. **The approach.** A paragraph about the design decisions you made and the alternatives you rejected. This is where you preempt the obvious questions ("why didn't you just X?").
4. **What to look at.** Pointers for the reviewer — files that need careful reading, edge cases worth checking, things you yourself are unsure about.
5. **What you tested.** Not just "I ran the tests" but specifically what you verified, especially for changes whose behavior is hard to test automatically.
6. **What's deferred.** Anything you noticed but consciously didn't address in this PR,