


# How Passion for Programming Can Make us Worse at our Jobs · Philosophical Hacker

 [medium.com/@philosohacker/how-passion-for-programming-can-make-us-worse-at-our-jobs-philosophical-hacker-24bd6310170e](https://medium.com/@philosohacker/how-passion-for-programming-can-make-us-worse-at-our-jobs-philosophical-hacker-24bd6310170e)

“Good programmers are passionate about what they do” is basically a platitude in our industry. *On the whole*, this may be true, but lately I’ve been interested in how our passion for programming might get in the way of us doing well for the companies we work for and may even lead to us being worse at programming specifically.

Here are some ways I think this passion can make us worse at what we do:

1. Neglecting the importance of the domain in building elegant solutions.
2. Poor judgments about the risks of technical debt (because of the affect heuristic)
3. Insistence on isolationism, which can enable the business to build the wrong thing.
4. Builder vs. Market quality mismatch, which leads to wasted effort.
5. Excessive specialization

These are all mistakes I’ve personally made, and while I don’t think there’s a *necessary* connection between passion for programming and these mistakes, I do think that my passion played a causal role in explaining these mistakes in my particular case. I thought this was counter-intuitive and worth sharing in case others find themselves in similar circumstances.

Let’s look at each of these mistakes in detail.

## Domain Neglect

Eric Evans opens *Domain Driven Design* with a great observation:

*The heart of software is its ability to solve domain-related problems for its user...Developers have to steep themselves in the domain to build up knowledge of the business...Yet these are not the priorities on most software projects...*

*..Most talented developers do not have much interest in learning about the specific domain in which they are working, much less making a commitment to expand their domain-modeling skills. Technical people enjoy quantifiable problems that exercise their technical skills.*

If he's right about the importance of domain modeling for good software, then our disposition to focus on technical problems can actually distract us from the design conversations we need to have to build good software.

He's got a great analogy for this in the same passage. The domain-ignorant dev is like a film editor 🎬 who chooses a shot that is better overall because someone walked in to that shot. He says:

*The film editor was focused on the precise execution of his own speciality. He was concerned that other film editors who saw the movie would judge his work based on its technical perfection. In the process, the heart of the scene had been lost.*

## Affect Heuristic Clouding Judgment on the Risks of Technical Debt

---

A big part of what we do as programmers is manage technical debt. Unfortunately, much of how we manage this technical debt is determined by our intuitive judgments about the risks and impact of technical debt in our codebase. These intuitive judgments are going to run through the affect heuristic, a mental shortcut we *subconsciously* use to judge the riskiness of X by considering our emotional reaction towards X.

As programmers, we don't like crap code, so we're likely to over-estimate the risk that code poses to the business. Widely-respected programmers have suggested that there are often other more important factors at play that are a risk to the business. Here are two examples:

*Good engineering is maybe 20 percent of a project's success. Bad engineering will certainly sink projects, but modest engineering can enable project success as long as the other 80 percent lines up right...*

*—KentBeck, TDD by Example*

*For the overwhelming majority of the bankrupt projects we studied, there was not a single technological issue to explain the failure. The cause of failure most frequently cited by our survey participants was “politics.”*

*—Tom Demarco and Timothy Lister, Peopleware*

The claim isn't that technical debt doesn't matter. It matters. The claim here is merely that a passion for good code can cloud our judgment—via the affect heuristic—about the importance of paying down technical debt relative to other goals.

### Programmer Isolationism

---

Consider two views on how involved programmers should be in non-programming activities.

On Joel Spolsky's view, developers ought to be isolated from the business via a “developer abstraction layer.” In fact, he says, “Management's primary responsibility is to create the illusion that a software company can be run by writing code, because that's what programmers do.”

Marty Cagan has a *very* different view. He says, “if your programmers are only programming, you're only getting half their value.” Support-driven development—where software developers actually do customer support—has even become a thing at companies like Zapier<sup>1</sup> and Wufoo<sup>2</sup>.

Simply asking which view is right is a bad question. Here are better ones:

1. In which circumstances is each view more appropriate?
2. Do we usually pick the right view for the right circumstance?

I think we often screw this up, and I think programmer passion, among other things, is implicated in this mistake. I think we tend to favor the Spolskian view too often. We should prefer the Spolskian view when the “how” of a project is murkier than the “what” and prefer the Caganesque view when the “what” is murkier than the “how.”

If you know exactly what you need to build but you're worried how you're going to build it, isolating developers from the business is a great play. Hard problems don't get solved with constant interruptions and a

general lack of focus on the problem. Spolsky's view FTW here.

If you don't know what you need to build and the how is fairly trivial, developers should be recruited to help the business understand what to build. Programmers understand the how, which means they can provide a great map of the space of possibilities of what to build.

The problem is that is our love for technical problems can enable the business to pick the wrong view for the wrong circumstance. We're like,

*If the business wants to isolate us so that we can program, awesome. You'll get no argument from us. We'll be busy programming. 😊*

Since serious growth as a programmer often comes on the heels of actual usage of a product by a large group of people, allowing the business to make this mistake is bad for our growth as programmers and bad for the business.

### Builder vs. Market Quality Standards Mismatch

---

Let's start with a confession: I often give myself permission to "fix" a part of the codebase that I can't stand looking at anymore. Even when this doesn't lead to an unanticipated bug, it often takes time away from shipping. "Just addressing tech debt," I say. But if I'm honest, it may be more about my sanity.

Some are fine with this. I once met an engineer at Yahoo! who said that "developer happiness" could be invoked during sprint planning to justify working on something. The authors of *Peopleware* actually claim that software quality should be set by programmers, not the market. They say:

*"We all tend to tie our self-esteem strongly to the quality of the product we produce...A market-derived quality standard seems to make good sense **only as long as you ignore the effect on the builder's attitude and effectiveness...**"*

This is a strong claim that I doubt is entirely true, but to the extent that it is true, it would mean the company wastes money satisfying our desire quality insofar as our quality standard outstrips the market demand for

quality.

This is especially problematic in a startup. An early stage startup is a war to survive, so when our passion for programming drives us to insist on higher quality than our customers demand, we wind up acting like soldiers who have gotten stuck in the trenches polishing our rifles because it makes us feel better.

I'm not saying that there's anything wrong with caring about quality or interesting problems. I just think that at a startup, it's tough if programmers are focusing on these things over survival.

### Excessive Specialization

---

For a while now, I've suspected that programmer specialization is often sub-optimal for businesses. Andy Grove's diner analogy and his concept of a "limiting step" in *High Output Management* gives me a nice way of explaining why this is probably true.

Grove says that if we're trying to run a diner well (or recruit talent, or build a compiler), we'd do well to identify the "limiting step" in the process required to create a breakfast. In a breakfast with an egg, toast, and coffee, preparing the egg is the limiting step.

This is because preparing the egg takes the most time of all the steps. Regardless of how fast we get at preparing coffee and toast, the number of breakfasts we serve will ultimately be limited by the egg preparation time, which, in turn, limits revenue the diner can generate. In many technology companies, there is a skill-set analogous to egg preparation in that it limits the overall rate at which features can be developed, which in turn limits the revenue the company can generate from those features.

An employee at the above diner who believed that improving her coffee prepping skills would actually help the business would be mistaken. The same applies to programmers who think that specializing in their non-limiting-step discipline is good for the business. Programmer passion for her "favorite stack" can blind her to this.

There are *plenty* of cases where continued specialization will in fact yield benefits to the business, even if programmers are specializing in a non-limiting-step discipline/stack.<sup>3</sup> A specialist may be able to product higher quality code in less time that contains less bugs, which could lead to increased referrals for your software product. Moreover, deeper knowledge can lead to a richer view of what's possible with a given stack, which can better inform product and project management decisions.

*However*, at some point, further investments in quality and keeping up with all the newest approaches to solving problems with our stacks will yield diminishing returns, and I suspect that this point occurs much sooner than a lot of us realize.

## Notes

---

1. <https://zapier.com/blog/support-driven-development/>
2. <https://genius.com/Kevin-hale-lecture-7-how-to-build-products-users-love-part-i-annotated>
3. I owe Andrew Dushane for this point.