Twitter Thread by GeePaw Hill



GeePaw Hill
@GeePawHill

Microtest Test-Driven Development is a strategy for *change*. To understand the case, we need to answer two questions: 1) Why have a strategy for change? 2) How does TDD provide one? Let's take up the first question today.

(Before we begin, I remind you of the relative unimportance of geekery to me just now. This is just respite.

Please work for change and support the others who are doing so.

Black lives matter.

Stay safe, stay strong, stay angry, stay kind.)

Why is having a change strategy so urgent?

The TL;DR is dead simple:

Because continuous ongoing change is at the base of all software development, both before and after our software is in the field.

When I read old-school software ideas, or for that matter what is being taught in our colleges for the most part today, I'm struck over and over again by the "timeless" aspect of the material. It is concerned almost entirely with the static and atemporal aspects of software.

In these approaches, applications spring Athena-like from the heads of our pantheon-worthy programmers. The designs emerge in an instant and once emerged they are final. The process looks like 1) Nothing. 2) Everything. 3) Move on.

I don't mean to say this has no value. On the contrary, it has produced a very great deal of both pure and applied science, and has certainly been a worthwhile endeavor. It's an impressive body of work, and I've had many years of delight from it.

But it's not enough.

Nothing, Everything, Move On, this frames software development as entirely centered around a finish-line. Implicitly (and sometimes explicitly) this diminishes the significance of everything that happens *before* the finish line, and everything that

happens *after* it.

Let's talk about after first. In old-school theory, "after" means "Move On". We have finished our app, it is being used in the video rental store -- sorry, couldn't help myself, you had to live through the '90s -- and there is nothing more to say or do.

Does this resemble the life of a professional programmer? Well. I have certainly had that experience. In the stretch of my career from 1980 to 2000 it was pretty commonplace. But somewhere around there, it became far less common, and today it's quite unusual.

Finish-lines in software are much less common than they once were. This is because of the extraordinary market-expansion we've experienced over the last forty years.

In the early days, this expansion came from the staggering decrease in the cost of physical computing. This made it possible to put computers *everywhere*. In the last two decades, the expansion has came from the staggering decrease in the cost of distribution, i.e. the internet.

You could satisfy that video rental owner in a month, giving him everything he wants. With market expansion, though, we realize we could have one app that satisfies a *million* video rental owners. All we have to do to keep expanding our market is expand our features & variants.

So our software hits the field with one customer, then ten, then ten thousand, and so on. Continually expanding our reach means we have to *start* with a reach. We ship once. Then we ship for another tier of customers, and again, and again.

So, in fact, "after" almost never means "Move On" for a modern developer. It means "next closeable customer tranche", and that translates directly into changing what we have already shipped.

Let's do the before. In oldschool theory, we start with nothing and a specification, and our job is to implement that spec. A running version of that spec is the finish-line. We take a deep breath, ask questions about the spec, and then run run run. A month or two later, voila!

Again, does that resemble the life of a working programmer? And again, I've done that many times. But most of those times were back in the day, and nowadays it hardly ever works that way. Why not? Because today's specs are vastly more complicated to implement than those old ones.

Modern apps are huge, w/orders of magnitude more moving parts than anything before 2000, h/w & s/w. The *screen* that you're reading this on has more RAM than any computer any of these old-school theorists worked with. There are *thousands* of apps & libraries letting you do it.

And how does that size and complication impact us? It makes development -- even against baseline specs, what we call MVPs -- take more time. The idea that a month or two of development is commonplace goes right out the door.

That means that the space between Nothing and Everything is very much larger. The development is going to take a lot longer, and hence become far more significant than in the old-school approach.

And what, dear reader, is another word for development?

Oh yeah. "Change".

There are just two strategies for all that change: horizontal -- in which we divide our app into layers and implement each one to its finish-line extent -- and vertical -- in which we divide our app into features & variants and implement each one to its finish-line extent.

Horizontal doesn't work very well. The reason is that the size and complication of the spec makes it difficult or impossible to determine in advance exactly what each layer does. We wind up adding things we don't need and needing things we didn't add. I'll dismiss that option.

Vertical works better. But though horizontal development depends on the accuracy about finish-line requirements for layers, vertical development depends on our ability to rapidly and safely change the code that is implementing the current version of a feature.

So before is change, and after is change, and both imply the passing of time. And thta very passing of time *itself* means change. Why? Because finish-lines only stand still for a very short time, less time than it will take us to reach them.

Does this mean there are no finish lines?

No. It's more like, there are a host of finish lines, one after the other, stretching from the second day of the project to the day no one new wants our software anymore.

We don't call these finish-lines, usually. We call them steps.

In coming muses, I will argue that TDD is an excellent change strategy. And, with certainty, some folks will argue against that idea. That's fine.

But one argument I won't take seriously: that having a change strategy doesn't matter.

Software development is continuous ongoing change. If follows, as night follows day, that to do it effectively for money, our strategy must be centered around the fundamental act of changing code.

Thanks for reading. Do me two solids, if you liked it.

1) Subscribe. Free, spam-free, full-text or audio, and it helps me make more.

https://t.co/0iffwG5jrd

2) Keep working for and supporting change, inside the trade and out.

Black lives matter.