Twitter Thread by GeePaw Hill

GeePaw Hill
@GeePawHill

Because microtest TDD is more a "way of geeking" than a technique or a received body of knowledge, building one's faculties is a long and sometimes perilous effort. Let's talk about learning.

(I, too, feel a little relief just now, tho not as much as some, because recent events aren't an ending, they're the beginning of a lot of work.

Black Lives Matter.

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

Let's keep changing this.)

I want to approach the question(s) of learning TDD in some ways that are a little different from others I've seen. I've written two TDD curricula myself, and am involved currently in helping with two more. I see all of them, including the current ones, as "interesting failures".

At first, learning/teaching TDD seems like a road, then over time it seems to become a tree, and ultimately it feels much more like a spreading activation network, including lots of external triggers, plateau periods, and transformative experiences.

(I'm open to the possibility that this is what *any* judgment-centric skill looks like, but for now I just want to talk about TDD.)

We work in a trade with a very fragile culture, dominated by badging, compartmentalization, rulesets, and the giving of orders. Developing one's judgment, nuance, experimentalism, and vision is quite difficult in such a culture.

The system's not malevolent, it's just reacting, as all systems do, when exposed to powerful environmental force. In this case it's the reaction to mind-blowing growth in demand. I worry, tho, that it's anaphylactic, an immune system running amok.

During the early stages of learning TDD, there are some pretty definitive lessons to be mastered, and they tend to go in a fairly simple sequence for learners.

You gotta learn the tool. How to run one, some, or all. How to find the tests for the thing you're working on. What failed tests look like. How to decide whether the test is right or the code is right.

You gotta learn how to write a test for code that's a) simple, and b) already there. Sometimes we skip that step, and go straight to how to writea test for code that's a) simple and b) not yet written. I, personally, think that's a bad skipping, but others disagree.

You gotta learn to write a whole class or function, by working red-green-refactor in a steady rhythm. We usually do this by offering small exercises, what we (mistakenly in my pedantic view) call katas.

Right around here, if you're learning from strong sources, you'll encounter your first "awkward collaborator" problems, situations where the language or the library or your own code presents a barrier to testability, and you start having to be clever.

This is the beginning of the tree-shaped part of the learning. It is also, sadly, the end of most formal TDD education.

"Here's your badge, go over that wall and face the Boche. Remember, kid, it's easy: red green refactor, RED GREEN REFACTOR, *RED* *GREEN* *REFACTOR*!!!"

Why the tree? Because there are lots of ways for a collaboration to be awkward, and correspondingly lots of ways to de-awkward things.

Why the sadly? If you haven't learned some of these ways, you're going to try to TDD while ignoring that awkwardness. This is just about exactly the wrong thing to do, and it will nearly always result in you eventually giving up in despair and declaring TDD a fraud.

You don't have to learn them all, you have to learn some of them, and you have to learn that there are more of them for different situations, because it will sustain your confidence, when you don't already see the way home, that there is one, you just haven't found it yet.

Crossing into the branches successfully has been, for me and I think for many others a matter of some luck, in two factors, 1) the order in which the variants are presented, 2) the quality of your four mentors. (More about this second one in a minute.)

Getting experience in these branches is how your sense of feel starts really going. To quote Rita Mae Brown, "Good judgment comes from experience, and experience comes from bad judgment."

In fact, I'd go so far as to say that learning TDD is impossible without using your real keyboard to solve modest but real problems on a regular basis using TDD. The challenge: most of us work in codebases that only have very simple problems or insanely complex ones. :)

Sample lessons from the branching part: "Can't test it here, move it there where you can test it, and just *use* it here."

"If I passed the supplies instead of the supplier, I could test this."

"If I had a host & two peers instead of a boss and a worker, that'd be testable."

Finally, the learning goes from feeling tree-shaped to feeling like a spreading activation network. You've got enough time in, seen enough problems & answers, that you work "cross-branch".

You take what you learned when you faced a database, and you see that it will work on a promise flow. You take the trick you learned in Java and you try it out in javascript.

And more and more, you're gaining "pre-emptive" awareness. You worry less and less about testing a design and more and more about designing testable things.

Flows of fad -- some good, some bad -- come from outside to trigger the spread. New tech, a new framework, some crazy gal who claims that the Frobisher system eliminated 99.999% of her team's defects in twenty minutes on a Thursday morning.

You'll try lots of tools and techniques, discard most of them, keep some in the belt for limited cases, and bring others into your daily practice. They, in turn, will lead you to still others to try.

It's a lot! Road to tree to network. And it doesn't happen overnight. In the rooms, we say "time takes time". It's a way of saying what Euclid said: "There is no royal road to geometry."

So how do you begin? I intend to talk soon about a lot of different things to learn about and try with TDD, but before we even go there, I'm going to make a very odd recommendation: Start looking around for your four mentors: a polestar, a rabbi, a teacher, and a swim buddy.

A polestar is someone up near the top of all this. You'll likely think of them as famous, maybe even use the dreaded term "thought leader". (They're mostly just big fish in a small pond, a fact most of them know well.)

You can use your polestar as a source of energy & ideas. These people, I fill the role for some, are passionate & noisy. We're not all right all the time, we throw out ideas and puzzle through them, we share what it's liking being good at something but not good enough.

Some candidates: (I won't use tags, as I hate big tag blocks myself.) Ron Jeffries, Kent Beck, Joe Rainsberger, Liz Keogh, Ted Young, Emily Bache. There are others, too. Find one with a genial style for you, some chemistry, and follow what they say and do.

A rabbi -- the term comes from, of all the damned things, the world of law enforcement -- is someone who can help your learning by giving you the time and space to make it possible. It's usually someone in your org that has more weight than you, and is willing to back you.

The truth is that you're going to have to learn TDD while you have a day job. A rabbi can help you do this. They're not experts or teachers -- at least not of TDD. Rather, they're supporters and enablers. Rabbis don't care about technique, they care about your growth.

A teacher is someone who's up for showing you the ropes. Teachers love making and sharing experiences with their students. They *do* care about technique, and they have some of it, and they want you to have it.

The best teachers are the ones who love questions, and the ones who know a different ways to get at the same idea. The very best ones are often only a chapter or two ahead of you in the book. Teachers might be paid, or they might not, or it might be indirect.

Some polestars are teachers, it varies. But it can be tricky: learning means changing one's vision and vocabulary. The seeing and saying can be quite different between a master practitioner and a noob.

A swim buddy is someone you know personally who's learning the same stuff you are at the same time -- or very close. Swim buddies help you pace yourself, they help you learn the questions you're not asking, they help you make the learning sticky. They're also a source of fun.

An invaluable service of swim buddies: combining and softening the ebb & flow of energy. Swim buddies give energy when you need it, and gather it from you when they need it.

So look around. Find your polestar, your rabbi, your teacher, and your swim buddy. That's how you really begin to learn TDD, and possibly any other complex skill. You don't have to decide today, and no such decision is binding anyway, but get started on it, yeah?

Attend carefully, by the way, to chemistry, and be prepared a) to shop around until it fits, and b) to grieve a little when it's time, as it inevitably is, to move on.

I have a whole lot more to say about learning TDD, and a lot of lessons in my head that I haven't seen presented elsewhere. We'll get to all that, but I think we're done for today.

A road, a tree, a spreading activation network. Microtest TDD is a way of geeking, it's not a technique or an add-on or an algorithm, it's an approach. And it's highly dependent on the development of your judgment & experience. Start your learning by looking for you four mentors.

Thanks so much for reading along. Two requests:

1) Subscribe.

https://t.co/0iffwG5jrd

2) By all means celebrate, but please don't stop working for change, inside the trade and outside.

Black Lives Matter.