Editor: George Fairbanks
gf@georgefairbanks.com

The Rituals of Iterations and Tests

George Fairbanks

WHEN I WAS a kid, my mother would tell me to clean my room, and I would dutifully push the vacuum cleaner around. It wasn't until I was living on my own that I realized that this was the right ritual, but the goal was to have a clean room, and the ritual might or might not achieve that goal. It was a minor epiphany of youth: my elders had handed down wisdom in the form of a ritual, but that ritual wasn't strong enough to ensure the outcome.

That's the situation we find ourselves in with software development processes. Today, there is a broad consensus that the best way to develop software is to follow an iterative process: evolving a working system and using automated testing to avoid regression. In fact, this has been the standard way to do things for so long that it's easy to just follow the ritual and not think much about it.

We must not fool ourselves and think that software development is mechanical. The rituals of iterative development and testing are a good starting point, but, unless we go beyond them, a project will become a tangled mess. Projects that follow the ritual are still endangered by two slippery slopes: the accumulation of ur-technical debt and loss of intellectual control. Before we take a look at those slippery slopes, let's make sure we're using terms for software processes the same way.

It was a minor epiphany of youth: my elders had handed down wisdom in the form of a ritual, but that ritual wasn't strong enough to ensure the outcome.

Waterfall, Incremental, and Iterative

If you were building a car using a *waterfall* process, you would proceed through stages including gathering requirements, analyzing the problem, designing a solution, building the car, testing it, and delivering it. The assumption is that you can mostly finish each stage before starting the next, but things rarely work out that way.

If you were building a car using an *incremental* process, you'd follow the same stages but would deliver a series of parts that add up to a car—perhaps the wheels first, then the frame, then the engine, and so on. Notice that you must have a design before you start building the parts; otherwise the parts will not fit together, so this is really just a waterfall process with a series of deliveries.

If you were building a car using an *iterative* process, you would pick up just a few requirements and build something simple, for example, a skateboard. Then, you'd pick up a few more requirements and evolve the skateboard into a scooter. This would proceed through a bicycle and motorcycle before becoming a car. The critical insight is that you deliver something quickly so you can learn from that

Digital Object Identifier 10.1109/MS.2020.3017445
Date of current version: 22 October 2020

experience, both about your technology and what your users want.

However—and this is a big caveat—in an iterative process, the parts you build early on will not fit the final product. It would be remarkably lucky if they did, because you had seen only some of the requirements. Besides, a skateboard with car wheels would be a lousy design. When using an iterative process, there is no way to avoid going back and revising existing parts.

A well-chosen variable name relieves a mental burden, but a misnamed variable sends us quickly down a line of mistaken reasoning.

Slippery Slope: Sedimentary Process

Despite that, teams drag their feet about substantive revisions. Once the code exists, many teams find excuses to avoid that expensive rework. Managers might say, "Our developers are smart, so why would they have written bad code?" Ward Cunningham invented the debt metaphor to explain exactly this situation to his management: it's not that smart developers write bad code, it's that they had a partial understanding of the problem and built a skateboard, but now they are asked to build a motorcycle.

Because they know refactoring is necessary, developers go through the refactoring ritual, but it's expensive, so they often don't finish the job and leave behind evidence of earlier iterations. They might leave behind code referring to skateboards or motorcycles even though they have delivered a car. I'm not aware of a name for an iterative process with imperfect cleanup across iterations, so I've been calling it a *sedimentary process* because the old ideas are still there in the code but partially covered in layers of new code.[1]

You can cope with the old ideas in code by mentally translating. For example, when you see a data structure still named "motorcycle," you mentally translate that to "car." Tougher translations are not syntactic. For example, steering a motorcycle is deeply different from steering a car—not a word-for-word substitution.

Our minds are only so big. We are able to work on larger, harder problems when we allow those thoughts to spill out onto an external representation, like pencil marks on paper. The machine an engineer builds also acts as an external representation, and a clock maker can reason about the clock's operation more easily when looking at the gears and levers in the mechanism. This effect is stronger with software engineering because source code expresses our ideas far more directly than steel or concrete can. A well-chosen variable name relieves a mental burden, but a misnamed variable sends us quickly down a line of mistaken reasoning.[2]

A sedimentary process is a slippery slope. At first, it's easy to recall what you wrote yesterday and translate in your head, but, over time and with many developers, it becomes increasingly hard to look at code that says one thing while thinking about a different thing. The code fails as your external representation, even though it works fine as a machine and passes all of the tests.

This is ur-technical debt, and it is what Ward Cunningham warned us about. As ur-technical debt accumulates, more of your effort goes to the translation instead of to new features. As he put it, "Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation. . . ."[3]

Slippery Slope: Loss of Intellectual Control

A second slippery slope is the loss of intellectual control. We gain confidence that our code does the right thing from two sources: reasoning about it and running it. Reasoning gives us intellectual control, for example, confidence that it doesn't deadlock because of the locking algorithm. Running code gives us statistical control, for example, confidence that it doesn't deadlock because we've been successfully running tests on it. As Dijkstra warned us, testing can reveal the presence of bugs but not their absence. The more tests we run, the more confidence we get, in a statistical sense.[4,5]

How does a project lose intellectual control? Let's assume that the team is following the rituals of iterative development and testing. The team will have intellectual control over the first module, since there was an idea that led to its being created, and the team will have statistical control over it, since the team wrote tests covering the module.

But then the edits begin. Let's say I pick up a requirement and edit the module to handle a new Boolean, and you separately do the same for another similar requirement. The state space handled by the system has grown, but it's easy for each of us to focus only on our own requirement, not the overall state space, and so we lose some intellectual control over how the system should work. A series of edits like this continues, driving down intellectual control over the module until nobody can explain the ideas behind it: it's just a jumble of code that happens to pass the tests, which by their nature are incomplete. At some point, we may recognize the mess, but at each step of the way, it seems expedient to just add to the jumble. That's how a team can lose intellectual control over a single module.

That single module tends to infect modules around it. When I write a new module that works with that messy one, I have clear ideas about my new module but not about the other. Where the two interact, I have to adjust my code to compensate for quirks in the messy one, quirks that I don't understand. That causes parts of my new code to have no clear explanation, often with comments warning others not to change this because it's working and I'm not sure why. That's how the loss of intellectual control is a slippery slope, with trouble in one module dragging down others.

There is also a human element to this slippery slope. When you write a module that's under good intellectual control, it is easy for your coworkers to use. The other way around is trouble: you will have a hard time understanding and using your coworker's module that's not under intellectual control. You may be tempted to reexert intellectual control over that other module first, but that is extra time and effort for you, far out of proportion to just adding a quirk to your module. Perversely, you are punished for doing what's good for the team.

This is a classic prisoner's dilemma. Both of you prosper when you both create modules under intellectual control, and both of you suffer from a tangled mess when you both neglect intellectual control. However, your coworker who neglects intellectual control will prosper at your expense if you alone try to clean up the mess. In the abstract, it's easy to point blame at coworkers, but, in practice, they are not trying to cause trouble. It's easy to see if tests exist and are passing, but it takes rare (but teachable) skill to notice intellectual control slipping away.

Technical Zombies

Most projects use iterative development and automated testing. This is best practice, but it's not enough. Your team must avoid the buildup of ur-technical debt from sedimentary development and must keep a healthy balance of intellectual and statistical control.

The adage "a stitch in time saves nine" has two pieces of wisdom for us here. First, the nature of slippery slopes is that, early on, the risks seem manageable, but you may already be sliding into disaster. Second, it's easier and cheaper to keep your project on track (a stitch) than it is to recover (nine stitches).

As a result, even before the team senses it, some projects are already sliding toward disaster, and it's too expensive to reverse course. I call these projects *technical zombies* because they are walking around like regular projects, but their vitality is gone.

When I think about technical zombies, my thoughts first turn to ancient Cobol systems dragged into the 21st century, but some zombies are brand new. Just last year, a friend told me about his current project. His team had created a technical zombie using modern languages, processes, and DevOps—and had done so in just a few years. I imagine that, with key people departing the project, a technical zombie could be created in just months.

Is your project a technical zombie? If you see any of these, consider them to be warning signs:

- The codebase reveals ideas from early iterations, like a skateboard.json data structure despite the product now being a car.
- When developers talk about the problem or solution, the terms they use are different from what you see in the code, and developers mentally translate those ideas.
- You know what refactorings to do, but the effort seems too great.
- The design has been evolved into something that nobody would have created deliberately.
- Newcomers to the project must, in effect, relearn the history of the project to understand the code.
- The system lacks names and types that largely explain the problem and solution.
- It's hard to state universal truths about the system, like "all customers are stored in the customers table and only in the customers table."
- Few developers can deliver an impromptu chalk talk about how the system works.

We should expect some of these in every healthy project. The nature of iterative development means that we may tolerate bicycle wheels on the motorcycle we build—just not forever, and definitely not a car with skateboard wheels. It's not that everything must be perfect all the time; it's that we can't allow the troubles to snowball out of control.

Ritual With a Goal

As a kid, I performed the ritual of pushing the vacuum around, but I did not always achieve the goal of a clean room. To do a good job, I first had to recognize (years later, sorry mom) that the rituals alone were not enough, and I had to figure out what the goal was.

Decades ago, we rejected waterfall processes because any mistakes we made early on were magnified by the end of the project. Today, we use iterative processes because they allow us to fix our mistakes as we go. We are right to follow the rituals of iterative processes and automated tests. We just have to keep in mind that these are a means to an end, not the goal.

So, what is the goal? Ward Cunningham advised us to keep our technical debt low by expressing our consolidated understanding in the code. Before him, in the age of waterfall development, Fred Brooks said the most important characteristic of a system is its conceptual integrity. I agree, and I think they are talking about the same idea: that a team works hard to invent and evolve a coherent theory that explains the problem and solution, and they work even harder to keep the code expressing that theory.

Reaching that goal means avoiding the two slippery slopes. First, to avoid sliding into a sedimentary process, we can take advantage of what iterations offer us: the chance to fix our mistakes as we go. I particularly like how Kent Beck said it: "[M]ake the change easy (warning: this may be hard), then make the easy change."[6] I've heard teams proudly talk about their refactoring efforts, but they sometimes boil down to consolidating duplicated code. Unless your worst design mistake is duplication, the repairs must go much deeper.

Second, to avoid sliding into overreliance on statistical control via tests, we can create theories of how the problem domain works and how our solution works. We can evolve our code so that it expresses these theories, works as an external representation, and can be read by newcomers who infer our theories.

This is intellectually demanding work, and, unlike story points or passing tests, it is hard to quantify. How do we really know if we're piling up layers of sediment or building an effective theory? There is no easy answer; the team is going to wrestle with those questions throughout the project. Software development is neither easy nor mechanical.

ABOUT THE AUTHOR

GEORGE FAIRBANKS is a software engineer at Google. Contact him at gf@georgefairbanks.com.

References

1. G. Fairbanks, "Ur-technical debt," *IEEE Softw.*, vol. 37, no. 4, pp. 95–98, July/Aug. 2020. doi: 10.1109/MS.2020.2986613.
2. G. Fairbanks, "Code is your partner in thought," *IEEE Softw.*, vol. 37, no. 5, pp. 109–112, Sept./Oct. 2020.
3. W. Cunningham, "The WyCash portfolio management system," in *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 92)*, Vancouver, Canada, Oct. 5–10, 1992, pp. 29–30. doi: 10.1145/157709.157715.
4. G. Fairbanks, "Intellectual control," *IEEE Softw.*, vol. 36, no. 1, pp. 91–94, Jan./Feb. 2019. doi: 10.1109/MS.2018.2874294.
5. G. Fairbanks, "Testing numbs us to our loss of intellectual control," *IEEE Softw.*, vol. 37, no. 3, pp. 93–96, May/June 2020. doi: 10.1109/MS.2020.2974636.
6. K. Beck, "For each desired change, make the change easy (warning: this may be hard), then make the easy change," Twitter post, Sept. 25, 2012. [Online]. Available: https://twitter.com/kentbeck/status/250733358307500032