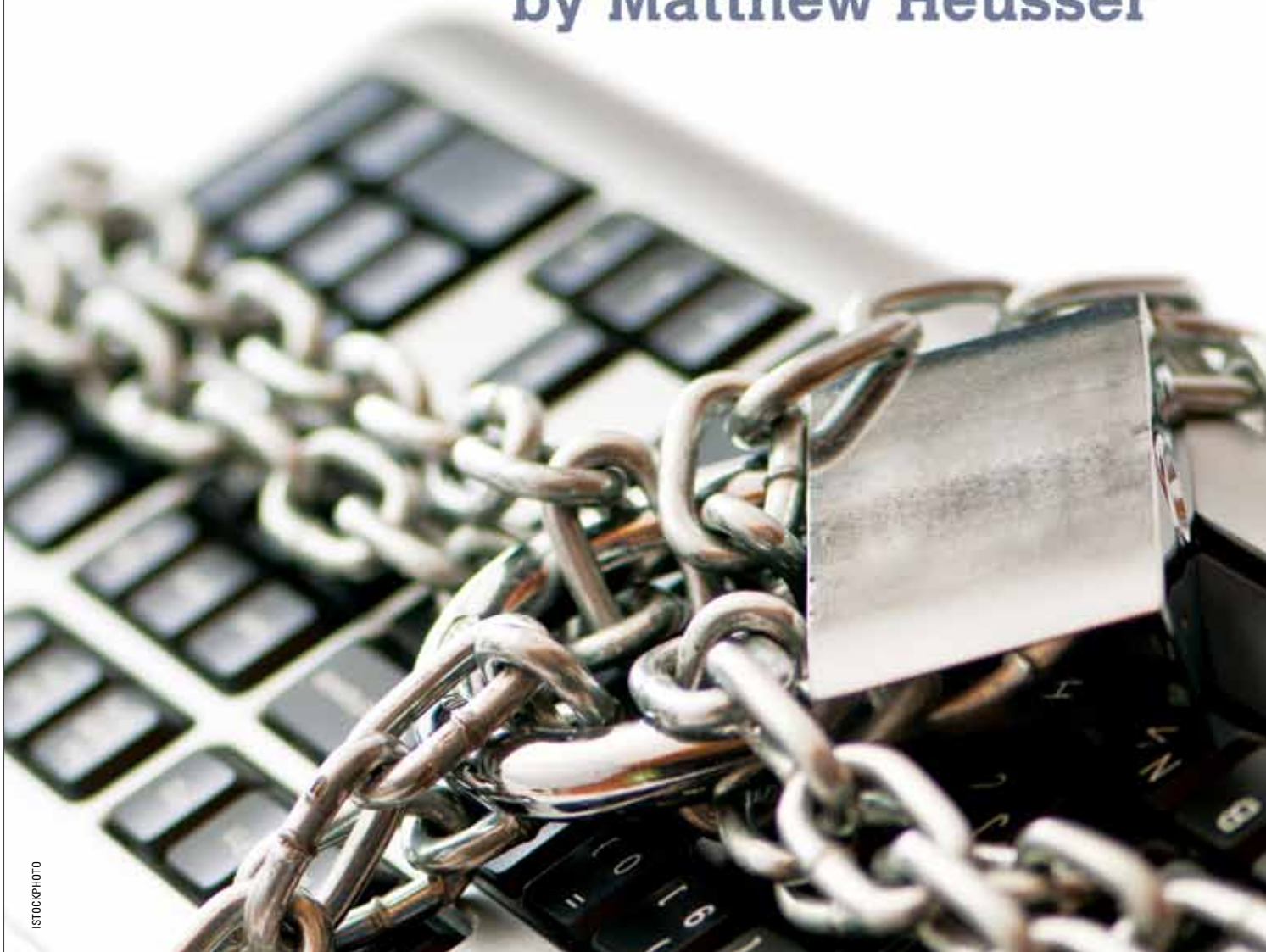


10 Thoughts on Technical Debt

by Matthew Heusser



ISTOCKPHOTO

Ward Cunningham popularized the term *technical debt* with his 1992 experience report on the development of the WyCash Portfolio Management System. Like *death march*, the term seems so intuitive as to not require explanation—yet Ward meant something specific. For example, putting off an upgrade to a compiler or a web server until after the next major release. These kinds of decisions provide some increased velocity now but at the cost of additional pain later. In his paper Ward writes:

Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. 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, object-oriented or otherwise [1].

At Calvin College in 2008, I hosted an Agile Alliance-sponsored workshop to discuss the issue of technical debt. During two days of discussions we came to very little consensus, but we did kick around some bright ideas, which I present here.

1) Technical Debt Is a Metaphor

One common objection to technical debt is that it is “broken,” an incomplete model of the world. Strictly speaking, that is true.

You see, technical debt is a metaphor. If it were not broken, if it were complete, then it would no longer be a metaphor but instead be an actual, well ... thing.

We use metaphors all the time in conversation; they can be especially helpful if the other person lacks shared experience. For example, you might explain the new hire process at a company as a “baptism by fire.” While the other person has never even walked into your building and doesn’t know the technology or the projects, he suddenly has some idea of what might go on—and you are able to explain it in three words, not three hundred.

Likewise, the technical debt metaphor can be helpful to managers and executives who are comfortable dealing with money.

Taking on debt in the real world is not always bad: A businessman might, for example, borrow money in order to purchase equipment to use to make even more money in the future. The key issues involved will be what the return is on the loan, what the interest is on the loan, and if the loan is ever paid back.

Thus, technical debt can be a tool to explain to a manager about the state of the code, without resorting to hard-to-explain terms such as “cyclomatic complexity” or “branch coverage.”

2) Technical Debt Is a Metaphor

Because technical debt is not an actual thing, attempts to measure it—trying to figure out the compound interest or the slope of the curve and assigning it a number for weight—don’t make any sense. We can find things that approximate technical

debt (like change in velocity over time or cyclomatic complexity measures), but those can be easily disconnected from the goal and manipulated.

The best measures for technical debt are likely qualitative, of the nature red/green/yellow, thumbs up or down, or the average of the team estimates from one to ten.

3) Informed Decisions Are Better than Ignored Decisions

Imagine a staff member who is under pressure from the boss and scared of being fired choosing to make technical shortcuts in order to hit a date. Now, consider the team that provides management several options for the schedule, along with the consequences of various choices.

The first person is trapped in a vicious cycle, putting the future of the project at risk in order to save today. In the second example, the costs and benefits are explicit, and management can make an informed decision whether to have the team increase the debt or not.

The first example may end in a firing, a demotion, or a system melting down; at the very least, costs will go up exponentially until the programmer transfers. In the second, it is at least possible that management (or, perhaps, the self-organizing, self-directed team) steers the outcome to success.

In the first case, the person is sadly out of luck. The second case at least has the possibility of a good outcome.

Aim for the second case.

4) Shoddy Work Is Not Technical Debt; It Is Shoddy Work

The classic examples for technical debt—all the way back to Cunningham—*defer* infrastructure work or refactoring. That is different from skipping essential design or thinking work in order to go code up a “big ball of mud.” While it is possible, at least in the short term, to go faster by increasing the technical debt, shoddy work slows you down almost immediately.

At the tech debt conference, Ron Jeffries pointed out this difference, saying that many low-skill developers cannot tell the difference between shoddy work and good work, and may use “taking on debt” as an excuse to write a lot of poor-quality code quickly.

Again, the telling sign of shoddy work is that it is a net negative: You get no additional forward momentum from it. If your team is experiencing this, you will likely want to sit down and figure a way out.

In some cases, a codebase will be in bad shape, existing as essentially a series of patches. Still, when programmers make changes, they can work to make the system a little bit better or a little bit worse. Ron was quick to point out the old Boy Scout rule for leaving a campsite in better condition than one finds it, insisting that we should strive to improve the code, at least a little bit, on every touchpoint.

There is a different kind of unprofessional work: the “clever hack.” You could argue that the clever hack is not debt because it does lead to a short-term improvement. When I use this term, however, I do not mean a good thing.

I think of it instead as an uncomfortable fix that the technical staff finds unprofessional, but management prefers—though likely does not understand.

In many cases, the technical staff is pressured to implement the hack and then, six months later when financial transactions are materially wrong in production, staff members will shrug their shoulders and blame management.

Which brings us to point number five ...

5) Professionals Take Responsibility for Their Work

Doctors, lawyers, and journalists, for example, have codes of conduct or ethics. These codes of conduct are binding and dominate the client-professional relationship.

When a client tells a lawyer how to conduct his defense and is insistent, it is common that the lawyer leave the assignment.

When a judge has a conflict of interest, he is expected to recuse himself from the case.

When a journalist is investigating a story that would incriminate a key advertiser, she is expected to run the story anyway. Journalists have been known to go to jail in order to protect sources, despite possibly extreme pressure from lawyers, police, and politicians.

Yet there are terms for professionals who “put their heads down” and “do what the boss says.” We use names like “shyster,” “hack,” “quack,” and “stoolpigeon.”

None of these terms is flattering.

If technologists want to be treated as professionals, we'll have to act like professionals. Doing work that is unethical or risky because “the boss said so,” and then blaming the boss, is not the hallmark of a professional; it is the telltale sign of a technician.

Now, I've talked to a few people about this, and the common reply is one of fear—fear of being fired. The reality is quite the opposite. The reason you are working on that module, that system, is because you are the expert. There is no one else better to do it. If a responsible job will take a week, and you can do the shoddy job in three days, there is likely no one else who could do the work at all in under a week.

It is unlikely that you will be fired. It is possible that you will get a slightly lower annual raise, but after a year or two of consistent behavior, you will be treated like a professional, and that will make a difference.

Still, I admit this conversation has to be done carefully and with skill. For more information on building a professional workplace, consider Robert Martin's *The Clean Coder: A Code of Conduct for Professional Programmers* [2].

6) Technical Debt Spirals Are a Perverse Incentive

Think about what happens when a team takes on technical debt and it actually works. By taking a shortcut, the team has met its deadline. “We took shortcuts last time and it worked” goes the thinking. What do you think is going to happen next time?

Not only is this a sort of subtle training to encourage shortcuts, but the act of skipping this time will slow down future progress. That delayed work, those cut corners, will have

to be paid later. And, like interest payments, the act of deferring the payment will increase the cost.

Technical debt slows down your velocity, which will make future project deadlines look unreasonable and may require additional shortcuts in the future ...

You get the picture. The team is going to do more and more sloppy work in order to hit *this* deadline, all the while slowing down future progress.

Eventually, if you are lucky, progress grinds to a halt. If you are unlucky, the software fails in production. In the worst cases the team may lack the expertise, time, or tools to fix what's broken.

7) Technical Debt Tends to Trend toward the “Feature Monster”

When people don't buy software, they often feel a need to have a reason, an “objection.” The real reason might be lack of budget or lack of purchasing authority, but it's much easier to come up with a missing feature. Likewise, if an internal customer wants to get more done in less time, one common complaint is the missing feature.

If the product team does not realize this, it is all too easy to come home with a never-ending list of features that must be done in order to reach Shangri-La, the land of the big sale.

Except we never reach Shangri-La. Worse, in a sort of subtle way, all of the new features that were implemented in a rush to get out the door not only slow down development but actually decrease the *value*, or quality, of the system.

This can happen in many ways: Overall system performance can decrease as features are added, the team might ignore bugs in the rush to get new features out, or the team may compromise the user interface with “one more button” or “one more field.”

Eventually, you end up with a mess.

I call this the “feature monster,” and, when you think about it, it is another subtle kind of technical debt. If you can think of a product that had every possible feature, but no one bought, then you have probably experienced the “feature monster.” It's the kind of “strategy” that can make products irrelevant and drive entire companies out of business.

The great irony with the feature monster is that, in many cases, the features don't make a difference. You implement them all and people still don't buy, because the real objection wasn't to the lack of features at all. Teams struggling with the feature monster are on a treadmill. If they finish early, they will be rewarded with a longer feature list. In this situation, taking shortcuts seems appealing, but it just means the product will become a worse ball of mud.

Like many other problems, the feature monster is born of ignorance. Explaining the system effects, along with a few classic examples, can help people to, for lack of a better term, “think different.”

Once people are aware of the problem, you can do what my colleague Ken Pier refers to as “breaking the back of the feature monster,” shifting the focus to making a good product that people actually want.

Once you realize you are carrying and feeding a feature

monster, you begin to wonder how you got tricked into it in the first place. Don't feel bad about it; the entire process is natural. You see ...

8) Technical Debt Is Often Encouraged by System Forces

The flip side of "What gets measured gets done" is "What is not measured is likely not done."

Say your project team has an aggressive schedule. Project management has the tools to measure conformance to schedule, feature completeness, even defects. What can the poor tech team do in order to comply with the schedule?

Why, take on technical debt, of course. The classic "solution" to this is to try to measure technical debt, but, well, the thing is ... you can't. The best you can do is a proxy measure, a measure-in-place-of.

9) Proxy Measures of Technical Debt will be Exploited

Proxy measures, or measures-in-place-of, are all over software development. We can't measure quality, for example, so we measure bug count as an approximation. We can't measure the test schedule, so we count the percentage of test cases completed (whatever that means). Software development managers have a problem getting a handle on productivity, so they count lines of code completed, or perhaps the number of "story points" accomplished within a two-week period.

Taken as spot observations, some of these proxies might be close to reality—lots of bugs tends to mean bad code, and a team that is producing a lot of code is likely more productive.

Yet when we observe these numbers and try to control them, human nature takes over. Suddenly the developers are spending more time trying to figure out why the bugs are invalid than they are spending actually fixing them.

This is even more the case with technical debt than with other disciplines. Sure, we could count the complexity of the code or the percentage of statement coverage of our unit tests, and the first time we look at those numbers, they might be good guides.

It's when we *systemize* and *reward* those measures that human nature takes over. The chart will continue to trend up and right, but instead of the actual things we want—better testing, less technical debt—we'll spend more time gaming the system.

My suggestion: Measure to get insight, not to control.

10) It's All Zero Sum

Notice a pattern here? With numerical measurement, things not measured will likely be stolen from in order to pay for things that are measured. Because technical debt is inherently hard to measure, teams under pressure tend to take it on without a conscious decision.

I suggest here that the solution is not to try to measure the debt. If you do that, you'll take a proxy measurement, which the team can exploit. More time spent measuring more things quickly becomes a death spiral.

The fundamental problem here isn't measurement. This is a different kind of problem, one that Fred Brooks wrote about three decades ago: Trying to fit ten pounds in a five-pound sack [3].

Technical work is zero sum. If you pay off tech debt, tomorrow's project is another day late. Take on the debt, and next month's project will be two days late. Take it on consistently, and, just like your credit card, you'll end up paying more on interest than any future progress. (If you've ever seen a maintenance project take three months to change

The Challenger Disaster

In 1986, the Space Shuttle Challenger exploded during liftoff. I was in fourth grade and watching the broadcast because the shuttle had teacher Christa McAuliffe aboard. Yes, I saw the shuttle explosion live on television, and my heart still goes out to the family and friends of the shuttle crew.

The physicist Richard Feynman is generally credited with explaining the explosion. According to Feynman, heat escaped through a rubberized ring, thus creating a fire in the fuel tank.

The issue was that these rubber O-rings had multiple redundancies built into them. If one failed, the system should have worked—even if two failed, the system should have worked.

It turned out, that was exactly what did happen; systems within the shuttle had failed, the engineering staff had decided the risk was minor, the ship went up anyway, and everything turned out fine. This happened again and again, for several years, each time with more exceptions written up. Time and again, the engineering staff "got away with it."

Right up until they didn't.

I do not mean to make light of this problem. It is less an engineering problem than a human-thinking issue, one the psychologists call "normalization of deviance."

Just like it sounds, normalization of deviance happens when violations of the rules go by without being enforced. The team might get away with it for a while, but, generally speaking, rules exist for a reason, and ideas have consequences.

The systems you work on might not mean life or death. Then again, they might. I said it earlier in this article, but, again, I think it bears repeating: Shoddy work is not technical debt; it is shoddy work. Professionals stand behind their work.

My friend Robert Martin has a story about the shuttle: It seems that many engineers did complain about the risks. They did go to the boss, and the boss's boss, and wrote letters, and made phone calls. They did everything they could; they were justified. But on that January day, when that ship exploded on the launch pad, in their heart of hearts, might have some of them been wondering: I could have called the *New York Times* or the *Washington Post* this morning and gotten an external investigation going. Why didn't I?

That said, one question remains: What kind of engineer will *you* be?

a few fields in a database built originally in a few hours, you know what I'm talking about.)

What to Do Tomorrow

In many ways your role will influence your thoughts on technical debt. Development staff tend to feel pressured by management; management, who is not actually writing the code, tends to have more of a "Who me?" attitude. Sure, managers are aware of the pressure, but they never would think of asking staff to take shortcuts that are unethical. So any debt taken on by the team should be either explicit or voluntary, right?

So we have an impasse. Each side is pointing a finger at the other side. Here are my suggestions:

Technical staff: Realize that nothing happens without you. That gives you far more power than you might realize. Also realize that there is a difference between "shoddy work" and "technical debt."

Punting the compiler to the next release—that's technical debt. Writing crappy code that you can't recognize the day after tomorrow—that's just bad work.

Sure, it is possible to save a day or two here or there with shoddy work—but the demands of professionalism won't allow it, no more than a doctor is allowed to skip washing his hands before surgery to save a few minutes.

Management: Over a lunch, take the pulse of the team about technical debt. Ask team members to compare the

actual state of the project to the ideal state and what it would take to get there. Share the actual business situation with the staff and make technical debt choices explicit. In some ways, you are a steward of the codebase, so try to have a more long-term view than the programmers, who will tend to internalize pressure to hit today's deadline. Finally, make tough choices clear to senior management. Comments like "We'll try" or "I'll see what I can do" are likely to be seen six months from now as firm commitments. Instead, make the conversation zero sum: "Sure, we can do that. What are we going to cut to get it done?"

Where to Go for More

Sadly, not enough of the notes from the technical debt workshop made it into print. Chris Sterling, one of the attendees, did recently complete a related book called *Software Debt* that covers the issue at a higher level. Likewise, Michael Feather's book *Working Effectively with Legacy Code* has some advice for programmers. Finally, although he did not attend the workshop, Ken Schwaber has a video online that covers some of these system dynamics [4]. **{end}**

matt.heusser@gmail.com

Sticky
Notes

For more on the following topic go to www.StickyMinds.com/bettersoftware.

■ References

Total Test Solutions, Unparalleled Value

Software Quality Assurance Challenge:

- deliver the best quality software product
- on time
- on budget

The Solution to Test Challenges:

- manage the test lifecycle process
- implement the best test methods
- reduce timelines, improve schedule predictability
- execute effectively
- reduce cost of test

SmarteSoft's tools and services support you at every step of the process with comprehensive automated testing solutions based on proven best practice methodologies – dramatically increasing test success.

SmarteSoft Total Test Solutions for:

- Functional Test
- Performance Test
- Regression Test
- QA Management

Whether you have never tested software before or have tested your product manually – or with a mix of manual and automated methods – SmarteSoft's easy-to-use tools and services will provide the boost you need to achieve dramatic success.



Learn more about SmarteSoft's Test Solutions and the real cost of software defects
www.smartesoftware.com/testolutions.php
+1.512.782.9409

SmarteSoft™