Perfectionists in a World of Finite Resources

Forrest Shull

I’VE BEEN TOLD that I can be a perfectionist—but I’m working on that and expect that I can improve myself.

In the meantime, there are just so many things that seem to need doing, and they all seem to be things that a well-rounded person should be able to do. Find me at a concert, and I’ll be beating myself up for not knowing how to play a musical instrument. Find me on travel, and I’ll be regretting that I never stayed with a foreign language long enough to become proficient. These are all correctable conditions; periodically I get the bug to improve myself, and I start scanning for local classes on piano playing, or German language skills, or photography, or dance. But then I run into trouble figuring out how to fit that in when I also have to conduct my own research.

If you’ll forgive me a gross overgeneralization, knowing software engineers, I suspect that perfectionists are somewhat over-represented among our ranks. The sense of fun in developing code, along with the sense of ownership and pride in good code, often contribute to a desire to make designs and code as correct and elegant as possible.

But the unpleasant reality is that both people and software projects can’t do everything we’d like. However, as much as possible, we should at least know the alternatives and make informed decisions about what we have to pass up.

The metaphor of technical debt, originally coined by Ward Cunningham (in “The WyCash Portfolio Management System,” Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, 1992, pp. 29–30), has helped me recently get a handle on this type of issue. Almost invariably in software projects, developers can be so focused on accomplishing the needed functionality that the software itself grows less understandable, more complex, and harder to modify. Since this system deterioration usually reflects a lack of activity spent in refactoring, documentation, and other aspects of the project infrastructure, we can view it as a kind of debt that developers owe the system.

Ward Cunningham’s metaphor helps make clear an important trade-off: although a little debt can speed up software development in the short run, this benefit is achieved at the cost of extra work in the future, as if paying interest on the debt.

Technical debt gives us a framework for thinking about the fact that not doing some good things today, no matter how valuable they seem on their own merits, allows us to invest in other good things. In short, there are always trade-offs in life.

The other side of the coin is that, when deciding to not spend the time on improving our systems better today, we should be able to identify some other short-term benefit that we’re deciding to go after instead. We should be able to think about what it means that we’re not going to spend that time improving the system (for example, by refactoring). Are we deciding not to do it at all, are we expecting to put in the time at a later date, or are we expecting the team to do it quickly (perhaps by cutting cor-
nners)? If we can’t identify a plausible alternative benefit that makes “going into debt” a viable trade-off, and/or if we can’t identify a plausible strategy for paying off that debt, then these could be good indications that the decision deserves a bit of additional scrutiny.

Project-Specific Trade-Offs
I’ve recently been part of a research group (along with Carolyn Seaman, Nico Zazworka, and Yuepu Guo) that’s been working with a number of different teams on questions related to technical debt. It should not be a surprise to anyone, I think, that what constitutes technical debt can vary greatly from one project to another—as do the trade-offs that teams are willing to make regarding it.

We’ve been finding a common thread across all of this work: that it’s healthy for projects to take a bit of time for reflection on what kinds of technical debt they’re most concerned about, and think of ways to keep an eye on how much debt is accumulating.

For example, we worked with a team at a multinational company that provides document-related business solutions, products, and services. The team was working on device drivers for the company’s high-end products, and due to the size and variety of the customer base, maintainability and portability were a must. To find areas where the code had decayed, we investigated computer-assisted support for detecting “code smells,” anti-patterns formulated by Kent Beck and his colleagues (Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999) as a way to help identify areas where good design principles were breaking down. Although some tailoring of the heuristics was necessary, these “smells” turned out to be a useful way to identify areas the team agreed were accumulating technical debt.

In another example, with a mid-sized, local software development company that focuses on database-driven Web applications, we’ve found that the company highly values its projects’ use of a reference architecture. In this case, instances where developers design their own solutions and avoid reuse represent technical debt, since redesigning the system to be in compliance is expected to lead to greater understandability and maintainability over time in the future. In this same context, we’ve also had some promising results with finding potential code smells and out-of-date documentation as indicators of technical debt.

In a final example, with a team developing high-performance code for supercomputers, we’ve noticed that the team members make optimal use of the parallel processors by strongly separating calls to parallelization libraries from the code doing scientific simulation—thereby allowing both computer scientists and domain experts to focus on what they know best. We’re currently exploring whether the instances where this separation of concerns breaks down should be treated as technical debt—that is, by detecting and fixing where the planned architecture of the system isn’t followed, whether we can help the developers create a more maintainable, more flexible system.

An important thing to note is that in all these cases, we need to discuss with the whole team whether our initial ideas for technical debt are right or wrong. It doesn’t make any sense to try to claim that there is some objective, outside standard about what should constitute technical debt; what matters instead is what the developers feel is important to their context. For example, the idea of separation of concerns may not be applicable on a more homogeneous team that doesn’t deal with multiple types of specialized complexity, and I’d never expect the specific rules for how we identified “code smells” would be applicable for every development team. For systems that
aren't going to spend significant time in maintenance, the indicators about out-of-date documentation and overly-complex code might be completely beside the point. (Using the metaphor, that debt may never have to be paid off, or at least, not by the same team!)

In all cases, we seem to have hit upon a process that is bearing fruit for these teams, and which teams could do in some measure on their own: find some initial examples of what we think constitutes technical debt for the development team; show those examples and have a group discussion about whether it would be useful to take some time to “pay down that debt”; and then either find additional examples that seem to fit the same mold or update our definitions of debt in this environment. It turns out to be quite useful to take a little time for having the team members reflect a bit on whether they agree with our definitions—why or why not—and how they would trade off corrective actions against the other project demands. It can also be quite constructive to have a conversation about the team’s expectations for which types of debt will never really cause a problem, which ones have the potential to make work much harder over time, and which ones increase risks (for instance, deferring testing or hazard tracking).

Using Debt Constructively

Of course, the point of thinking about technical debt is not just to feel bad about the unfinished work that we’re letting accumulate. Having ways to identify what we care about, and estimate how much work we’ve been putting off, gives us a way to argue for taking a step back and doing some of those necessary housekeeping chores (“paying down the debt”) when the cost we are paying for them outweighs the benefits we’re getting from having deferred them (“when the interest payments become too high”).

Monitoring debt hopefully also gives us the ammunition to fight for spending time on those corrective actions against skeptical teammates or managers. In our experience, the most typical corrective actions include the following:

- Refactoring helps when the software design has degraded, the reference architecture has gone out the window, or certain parts of the system have gotten too complex or badly structured for their own good.
- Spending energy and time on updating documentation helps when newcomers are having a hard time getting up to speed on the project, when users are having a hard time making use of the feature set, or when code that was meant to be reused is being underutilized.
- Spending energy on additional verification and validation helps when the team isn't quite sure of (or is quite worried about) the risks of system failures, security lapses, and similar problems.

These aren’t the only options, of course, but we’ve found them to be the software engineering activities that most often get short shrift while teams work on new functionality.

Outside the software engineering domain, I’ll be consoling myself with the thought that passing up some of the things I’d like to be doing—such as finally learning how to play an instrument—should be properly viewed as a deliberate investment in a career I also find fun and interesting. But boy, do I have a long list of things I need to do when I retire in another couple of decades! 😍

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.