acmqueue

Managing Technical Debt

Shortcuts that save money and time today can cost you down the road.

Eric Allman

In 1992, Ward Cunningham published a report at OOPSLA (Object-oriented Programming, Systems, Languages, and Applications)[2] in which he proposed the concept of technical debt. He defines it in terms of immature code: "Shipping first-time code is like going into debt." Technical debt isn't limited to first-time code, however. There are many ways and reasons (not all bad) to take on technical debt.

Technical debt often results from the tension between engineering "best practices" and other factors (ship date, cost of tools, and the skills of the engineers that are available, among others). Roughly speaking, technical debt is acquired when engineers take shortcuts that fall short of best practices. This includes sneaking around an abstraction because it is too hard (or impossible) to figure how to "do it right," skipping or scrimping on documentation (both in the code and external documentation), using an obscure or incomplete error message because it's just too hard to create something more informative, implementing code using a simple but slow algorithm even though they know that a better algorithm will be needed in production, using `void*` when you really should have created an appropriate `union*`, using build tools that don't quite work for the system at hand, skimping on good security practices, not writing unit tests, and so forth. Admit it—you've all done one or more (or maybe all) of these things at some point or another in your career. (Technical debt may also be taken on intentionally as a strategy to save time or money; more about that later.)

Not all debt (whether technical or financial) is bad. Few of us can afford to pay cash for a house, and going into debt to buy one is not financially irresponsible, provided that we know how to pay it back. In contrast, charging up luxury items on a credit card, knowing very well that your paycheck won't cover them, is usually a recipe for disaster. Using a simple but slow algorithm in a prototype can be exactly the correct path, as long as you have a plan for how you are going to update the code before it ships. That means allowing time in the schedule, making sure the issue is tracked so it doesn't get lost in the shuffle, knowing when you implement the code that a good algorithm actually does exist that will work in this instance, and trusting that management will support you.

Understanding, communicating, and managing technical debt can make a huge difference in both the short- and long-term success of a system. (Note that although this article focuses on technical debt in software engineering, many of these principles can be applied to other technical disciplines.)

COMPARISON WITH FINANCIAL DEBT

Going into financial debt usually has three important properties. First, the person making the loan wants it to be repaid eventually. Second, you usually have to pay it back with interest—that is, you pay back more money than you got in the first place. Third, if it turns out you can't pay it back, there is a very high cost, be it declaring bankruptcy, losing your house, or (if you borrowed from the wrong person) a long walk off a short pier wearing cement shoes.

Technical debt is similar in some ways, but different in others. Although you don't have to pay back the debt on any fixed schedule (and some debts may never need to be paid back), you generally do have to pay back (i.e., rewrite the code or otherwise fix the problem) the parts that affect you or your customers in a significant way. The "interest" is accrued every time you or anyone else (support-desk workers, future programmers, customers, etc.) working with your system is delayed because of bugs, performance problems, inexplicable misfeatures, time spent researching what has gone wrong when the system could have given a more explicit error message, etc. Failure to fix problems can result in the utter collapse of a system—the customer gives up and goes elsewhere, the system becomes so slow and brittle that it has to be rewritten from scratch, or in extreme cases the company is forced to close its doors.

There are some significant differences as well. Perhaps the most pernicious one is that the person who takes on technical debt isn't necessarily the one who has to pay it off—in fact, most of the time the one who takes on the debt can shuffle the costs on to other people, which encourages taking on debt. Far too many developers do not maintain their own code. Many companies have a policy that software moves from a development mode that is staffed by their best programmers to a maintenance mode staffed by second-tier engineers (who are paid less but often have far more difficult jobs than the premier team). Sometime it isn't even anyone in your organization who is paying the interest: it's the users who have to pay. Developers are rewarded more on implementation speed than long-term maintainability, and may have moved on to a different project or company before the real cost is paid. This gives the initial developer little incentive to do the job right the first time.

Unlike financial debt, technical debt almost never has to be paid off in its entirety. Most (probably all) production systems have warts that don't have significant impact on the usability or long-term maintainability of the final system. Very few systems have no `TODO` or `FIXME` or `XXX` comments somewhere in the source code. Note that the cost of paying back technical debt comes in the form of the engineering time it takes to rewrite or refactor the code or otherwise fix the problem. If the interest you ultimately accrue is less than the cost of paying back the debt, there is no point in paying it back in the first place. The problem is that it can be difficult to know in advance which debts will ultimately have the highest cost.

For example, when U.C. Berkeley's CalMail system went down in November 2011, the problem was traced to deferred maintenance—in particular, the decision to postpone updating the system even though it was known to be near capacity.[5] One disk in a RAID died, shortly followed by a second, and the cost of rebuilding the array reduced capacity sufficiently to create a crisis. Murphy's law needs to be taken into consideration when deciding how much technical debt to accept. In the CalMail case, individual hardware failures were expected in the base design, but multiple failures, happening during a historically high usage spike, created a condition that was not quickly resolvable. According to Berkeley's associate vice chancellor for information technology and chief information officer, Shelton Waggener, "I made the decision not to spend the million dollars to upgrade CalMail software for only 12 months of use given our plan to migrate to new technology. We were trying to be prudent given the budget situation, (but) in retrospective [sic] it would have been good to have invested in the storage upgrade so we would have avoided this crisis." This is a case where technical debt was taken on intentionally but turned out to be a bad gamble. Had the system survived that 12-month window, the school likely would have saved $1 million during a budget crunch.

There is a saying to the effect that there are three variables in engineering: time, functionality, and resources—pick two. In fact, there is a fourth variable: debt. Of these four variables, you can set any three of them, but you can never set all four; something just has to give, and very commonly debt is the free variable in the equation. Debt can seem "free" at first, but technical debt tends to build on itself. If the acquisition of debt involves interest in the form of increased effort to maintain and extend the system, then as you take on debt it gets harder and takes longer to do maintenance and extension. This is one form of collapse under debt: if all of your "income" (in the form of effort) is spent paying off interest and nothing is left over to move the system forward, then that system is stuck. This is especially obvious if productivity is measured in lines of code produced per day, a measure that should be relegated to the fires of hell. You don't have many choices left: add effort (hire more engineers), abandon the system and move on, or go bankrupt. In this sense, the interest on technical debt is actually compound interest, or put another way: if you don't stay on top of the debt, then the payments go up over time.

Consider these other interesting comparisons. Steve McConnell, CEO and chief software engineer at Construx Software, distinguishes between unintentional and intentional debt, which in turn is broken up as short-term (tactical) versus long-term (strategic) debt.[6] He also notes that when a system is nearing end of life, incurring debt becomes more attractive, since all debt is retired when a system is decommissioned. He also makes some interesting observations on how to communicate the concept of technical debt to nontechnical people, in part by maintaining the "debt backlog" in a tracking system and exposing it in terms of dollars rather than something more tech-oriented.

In a slightly different analysis, software designer Martin Fowler breaks down technical debt on two axes: reckless/prudent and deliberate/inadvertent.[3] He describes reckless-deliberate debt as "we don't have time for design," reckless-inadvertent as "what's layering?" and prudent-deliberate as "we must ship now and deal with consequences." This exposes a fourth class of technical debt that doesn't map easily to the financial model: prudent-inadvertent, which he describes as "now we know how we should have done it."

Another analysis of technical debt that resonates with some people is that managing technical debt is a way of managing risk in the technical realm. Software consultant Steve Freeman discusses this by comparing technical debt with an unhedged (or "naked") call option.[4] Either case (risk or unhedged calls) allows for the possibility that debt may never need to be paid back; indeed, big money can be made by taking appropriate risks. Naked calls, however, can also lose all of their value—essentially, the risk is unlimited. This doesn't often happen (most of the time when you lose, you lose only some of your money, not all of it), but it can happen, much as the wrong choice of technical debt can result in disaster.

So far we've spoken of technical debt as though it were unique to coders. This is far from true. For example, the operations department incurs its own kind of debt. Avoiding a disk-array upgrade is a tradeoff between technical debt and financial costs. Failure to consider power and cooling requirements when adding new, hotter equipment to a machine room is a debt. Failure to automate a simple-but-tedious manual process is a debt. Systems administrators who have neither (for lack of desire, inspiration, or time) documented the systems they support nor trained co-workers before going on vacation are another example. The comparison of technical debt to risk management is often starker in these non-code-related situations: you are betting that you won't run out of disk space or bandwidth, that you won't have a super-hot day, that your system won't become so

successful that the manual process becomes a bottleneck, or that nothing will go wrong while you're in Machu Picchu.

Certain staffing issues can lead to another form of technical debt: having parts of systems that are understood by only one person. Sometimes this happens because the staff is spread too thin, but it can also be caused by insecure individuals who think that if they keep everyone else in the dark, then they will be indispensable. The problem, of course, is that everyone moves on eventually.

MANAGING YOUR DEBT

Technical debt is inevitable. The issue is not eliminating debt, but rather managing it. When a project starts, the team almost never has a total grasp on the totality of the problem. This is at the root of the failure of the waterfall model of software development, which posits that all requirements can be finalized before design begins, which in turn can be completed before the system is implemented, and so forth. The argument seems good: the cost to make a change goes up exponentially as the system is developed, so the best path is to get the early stages done right before moving on. The reality is that requirements always change ("requirements churn"). It is often better to have a working prototype (even though it isn't complete or perfect) so that you and the customers can start gaining experience with the system. This is the philosophy behind Agile programming, which accepts some technical debt as inevitable but also mandates a remediation process ("plan for change").

As necessary as technical debt may be, however, it is important that the strategic parts of it be repaid promptly. As time goes on, programmers move to other companies, and the people who agreed to various compromises have moved on to other projects, replaced by others who don't see it the same way. Failure to write the documentation (both internal and external) for the initial prototype may be a good tradeoff, but the longer it goes the harder it is to write—if only because human memory is transient, and if you show most people code they wrote a year ago they will have to study it to remember why they did it that way. Code that is intended to have a limited life span may be immune to these concerns, but many short-term "prototypes" end up getting shipped to customers. Unfortunately, Fred Brooks' statement in *The Mythical Man-Month*, "Plan to throw one away; you will, anyhow,"[1] seems all too often to be corrupted to, "Make your prototype shippable; it will, anyhow." These two statements are not contradictory.

Equally as important is that some forms of technical debt are so expensive that they should be avoided entirely whenever possible. Security is an area where taking shortcuts can lead to disaster. You never want to say, "We're using passwords in the clear today, but we'll come back someday and change it to challenge-response," in anything other than very early prototypes that no one but you will ever see. This is a recipe for disaster if it ever gets accidentally deployed. You also want to avoid enshrining "bet your company" shortcuts in code. If for some reason you have no choice (for example, because during development other engineers have to write code that will interface with yours and you can't afford to keep them waiting), keep a journal of "debts that must be repaid before release." It's amazing how easy it can be to forget these things if they aren't written down.

Release cycles can make a considerable difference in the rate of acquisition and disposal of technical debt. The modern trend to "release early and often," especially in the context of Web-based services, has made it much easier to take on technical debt but has also made it easier to resolve that debt. When well-managed, this can be a blessing—taking on debt earlier allows you to release

more functionality earlier, allowing immediate feedback from customers, resulting in a product that is more responsive to user needs. If that debt is not paid off promptly, however, it also compounds more quickly, and the system can bog down at a truly frightening rate. Another side of Web-based services in particular is that a correct but inefficient solution can actually cost your company more money—for example, in the form of server-farm rental fees. Fortunately, this makes the debt easy to translate into dollar terms, which nontechnical stakeholders usually find easier to understand than assertions about maintainability.

Not all technical debt is the result of programmer laziness. Some is imposed by management or other departments, especially when they do not understand how pernicious this debt can be. Customers usually buy features, not long-term maintainability, so marketing departments often encourage engineering to move on to the next great thing rather than spending the time necessary to consolidate, clean up, and document the existing system. To them, taking these steps is an unnecessary cost—after all, the system works today, so why does engineering need to spend time gilding the lily?

There is another aspect to technical debt to consider: it occurs in many ways and is ongoing. It can come from the design or implementation phases, of course, but can also occur in the operational phase. For example, a computer system may have had a UPS (uninterruptible power supply) designed and installed, but deferred maintenance—in the form of failing to test those units and replace batteries—can render them useless. Disk arrays may be adequate when specified, but as the system grows they must be upgraded. This can be especially hard when attempting to extract dollars from a cash-strapped management to upgrade something that, from their perspective, works fine.

Management all too often aids and abets this problem. The current business mantra of "shareholder value" would be fine if shareholders were patient enough to reward long-term value creation. Instead the tendency is to think quarter to quarter rather than decade to decade, which puts immense pressure on everyone in the organization to produce as much as possible as quickly as possible, regardless of the longer-term costs (as indicated by the old lament, "there's never time to do it right, but there's always time to do it over"). Pushing costs into the future is considered a good strategy. This strongly encourages assumption of technical debt. An indicator of this is when engineering is perpetually in "crunch mode" rather than using crunches sparingly. How many companies advertise being "family friendly" on their Web sites and in their corporate values statement while encouraging their employees to work 60-hour weeks, penalizing "slackers" who work 40-hour weeks and then go home to their families? In these environments, the assumption of inappropriate technical debt is nearly impossible to avoid.

This isn't to say that management is always wrong. There are appropriate times to accrue debt. If my child needed a critical medical treatment I wouldn't refuse just because it meant taking on debt, even if it would be expensive to pay back. Likewise, management has a responsibility to customers, employees, and (yes) investors that can sometimes impose uncomfortable requirements. Debt taken on with eyes open and in a responsible way is not a bad thing. U.C. Berkeley's CIO made a bet that turned out wrong, but it could have been a winning bet. He knew he was making it, and he took responsibility for the problem when the roof did cave in. The difficulty is when management doesn't understand the debt they are taking on or takes it on too easily and too often, without a plan for paying it back. In a past job I argued that we needed more time to build a system, only to be blamed by management when we had a high defect rate that was directly attributable to the artificially short

schedule that was imposed against my better judgment. In this case, management didn't understand the debt, ignored warnings to the contrary, and then didn't take responsibility when the problems manifested.

COST OF DEBT FROM VARIOUS PERSPECTIVES

Technical debt affects everyone, but in different ways. This is part of the problem of managing the debt—even if you understand it from your perspective, there are other legitimate ways to view it.

CUSTOMERS. It may seem that the customers are the ultimate villains (and victims) in this affair. After all, if they were more patient, if they demanded less from the products and gave the company more time to do the job right the first time, none of this would happen (or maybe not). True, customers can sometimes focus more on features (and, sadly, sometimes on marketing fluff) than on long-term maintainability, security, and reliability, yet they are the ones who are most badly injured. When the mobile network goes out, when they can't get their work submitted on time, when their company loses business because they are fighting the software, they pay. Ultimately, it's all about doing what the customers need, and customers need software that works, that they can understand, that can be maintained and extended, that can be supported, and (ultimately) that they like using. This cannot happen without managing the technical debt at every level through the process, but customers seldom have any control over how that debt is managed. It is worth noting that customers who are paying for bespoke solutions generally have more control than customers who buy software "off the rack," who for the most part have to use what they are given. At the same time, when you are building software for particular customers, you may be able to negotiate "debt repayment" releases (probably not using that term).

HELP DESK. Those who work on the help desk deserve a special place in heaven—or occasionally in hell. Customers seldom call to say how happy they are; they generally have a rather different agenda. Help-desk personnel suffer from almost every aspect of technical debt: poorly designed interfaces, bad or nonexistent documentation, slow algorithms, etc. In addition, things that may not seem to affect them directly (such as obscurity in the code itself) will have an indirect effect: customers get more ornery the longer it takes to fix their problem. Though the help desk is the customers' primary input to the internal process, the desk often has no direct access to the people who can solve the problem.

OPERATIONS. In a service-oriented environment the operations people (those who carry the beepers 24/7 and who have to keep everything working) are far too often the cannon fodder on the front lines; they can spend much of their time paying for decisions that other people made without consulting them. Sometimes they get to look at the code, sometimes not. In any case they get to look at the documentation—if it exists. The (minimal) good news is that they may be able to pass the problem off to a maintainer as long as they can come up with an acceptable work-around. The rise of the DevOps movement—the concept that operations folks need to work with developers early in the cycle to make sure that the product is reliable, maintainable, and understood—is a positive development. This is a great way of reducing long-term technical debt and should be strongly encouraged.

ENGINEERS. Engineers fall into two roles: the developers who write the code and the people who have to repair, extend, or otherwise maintain that code (these may be the same engineers, but in many places they are not). At first glance, the initial developers seem to be the major creators of

technical debt, and they do have strong incentive to take on debt, but as we have seen, it can come from a number of sources. In its early days technical debt is almost invisible, because the interest payments haven't started coming due yet. Doing a quick, highly functional initial implementation makes the programmer look good at the cost of hampering engineers who join the party later. In some cases, those programmers may not even realize they are taking on the debt if they have limited experience maintaining mature code. For this reason, an average-speed, steady, experienced programmer who produces maintainable code may be a better long-term producer and ultimately higher-quality engineer than a "super-stud programmer" who can leap tall prototypes in a single bound but has never had to maintain mature code.

MARKETING. These customer-facing people often have to take the brunt of customer displeasure. They can often be the people pushing hardest for short product development times because they are pressured by sales and the customers to provide new functionality as quickly as possible. When that new functionality doesn't work properly in the field, however, they are also the ones on the wrong side of the firing line. In addition, pressure for quick delivery of new features often means that later features will take even longer to produce. Great marketing people understand this, but all too often this is not a concept that fits the marketing world model.

MANAGEMENT. There is good management and bad management. Good management understands risk management and balances out the demands of all departments in a company. Bad management often favors a single department to the detriment of others. If the favored department is marketing or sales, management will be inclined to take on technical debt without understanding the costs. Management also pays a price, however. It's not true that "there is no such thing as bad publicity," especially when your company appears to be circling the drain. Management should have no difficulty embracing the concept of managing financial debt. It is also to their advantage to manage technical debt.

SUMMARY

Technical debt can be described as all the shortcuts that save money or speed up progress today at the risk of potentially costing money or slowing down progress in the (usually unclear) future. It is inevitable, and can even be a good thing as long as it is managed properly, but this can be tricky: technical debt comes from a multitude of causes, often has difficult-to-predict effects, and usually involves a gamble about what will happen in the future. Much of managing technical debt is the same as risk management, and similar techniques can be applied. If technical debt isn't managed, then it will tend to build up over time, possibly until a crisis results.

Technical debt can be viewed in many ways and can be caused by all levels of an organization. It can be managed properly only with assistance and understanding at all levels. Of particular importance is helping nontechnical parties understand the costs that can arise from mismanaging that debt.

REFERENCES

1. Brooks, F. 1995. *The Mythical Man-Month*, Anniversary Edition. Chapter 11. Addison-Wesley.
2. Cunningham, W. 1992. The WyCash portfolio management system. OOPSLA Experience Report; http://c2.com/doc/oopsla92.html.
3. Fowler, M. 2009. Technical debt quadrant; http://martinfowler.com/bliki/TechnicalDebtQuadrant.html.

4. Freeman, S. 2010. Bad code isn't technical debt, it's an unhedged call option. Higher-order Logic; http://www.higherorderlogic.com/2010/07/bad-code-isnt-technical-debt-its-an-unhedged-call-option/.
5. Grossman, S. 2011. Calmail crashes last multiple days. *The Daily Californian* (December 1); http://www.dailycal.org/2011/12/01/calmail-crashes-last-multiple-days/.
6. McConnell, S. 2007. Technical Debt. Construx Conversations: Software Best Practices; http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ERIC ALLMAN has been a programmer, a maintainer, a help-desk survivor, both front-line and executive management, a consultant, a technology writer, and occasionally even a customer. He appreciates the difficulty and sometimes idiocy of all of those roles.