Editor: **Tore Dybå**
SINTEF
tore.dyba@sintef.no

Managing Technical Debt

Insights from Recent Empirical Evidence

Narayan Ramasubbu, Chris F. Kemerer, and C. Jason Woodard

TECHNICAL DEBT refers to maintenance obligations that software teams accumulate as a result of their actions. For example, when programmers take a design shortcut to more quickly roll out functionality critical to business stakeholders, they start accumulating maintenance obligations related to the shortcut. Like financial debt, technical debt incurs interest in the form of additional maintenance costs until the debt is fully paid. Ward Cunningham first used the debt analogy as he reflected on his product development experience using object-oriented programming:

> *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. Objects make this transaction tolerable. The danger occurs when the debt is not repaid. … Entire engineering organizations can be brought to a stand-still under the debt load.*[1]

Cunningham's reflections portray managing technical debt as an optimization problem. This inspired us to undertake research projects to rigorously model technical debt and understand its impact on both software product performance and business strategies. Here, we synthesize the empirical findings of our projects to develop practical insights for managing technical debt.

Recognizing Trade-Offs

An important first step is to recognize the trade-offs involved in managing technical debt. Software teams need to clearly understand both the benefits and costs of technical debt in their specific business environment. For example, a firm pursuing an early-to-market business strategy might choose to incur technical debt to speed up product development and accelerate customer acquisition. But, if this debt isn't repaid in a timely way, its benefits might be eroded by the long-term costs associated with poor product reliability and with the difficulty in meeting customer demands.

Our research shows that software teams willing to accumulate technical debt could speed up functionality deployment in their products by as much as three times.[2] However, they had to contend with a threefold increase in the backlog of unresolved errors and a drop of more than 50 percent in long-term customer satisfaction scores.

Three Dimensions to Optimize Technical Debt

Once software teams recognize the specific trade-offs they face, they can move toward optimizing their debt load by tracking appropriate metrics and instituting policies to manage the level of technical debt within the range they're

 0740-7459/15/$31.00 © 2015 IEEE

willing to tolerate. Recent research has outlined how to identify and measure technical debt for a variety of programming languages.[3,4] To translate this data into actionable decisions, software teams should map their technical-debt metrics to three important dimensions (see Figure 1).

The first dimension is customer satisfaction needs, which refer to the extent of demands placed on software teams to keep end users satisfied. A variety of factors could affect this satisfaction, including the end users' intrinsic motivation, the perceived ease of use, and the perceived benefits of use. When those three metrics' values are low, software teams must deploy strong triggers for initiating customer adoption and continued product use. Such triggers could be faster rollout of the demanded functionality or enhanced usability. However, these might sometimes come at the expense of not being able to follow recommended standards focused on long-term design stability. Our research shows that, all other things being equal, advancement by a month of the rollout of an end-user-demanded functionality could increase customer satisfaction by up to 55 percent in the short term.[2]

The second dimension is the extent of software reliability demanded by the business. Our research indicates that technical debt accumulated in enterprise software systems tends to increase the chance of system failures. In examining failures of enterprise systems at 48 Fortune 500 firms, we found that technical debt arising from business logic and data schema customizations that violated vendor-prescribed design standards increased the chance of system failures by as much as 62 percent.[4] The extent to which businesses can tolerate such risk can vary; software teams must carefully consider this when deciding to accept or avoid technical debt.

The third dimension is the probability of technology disruption in a firm's environment. We observed a few cases in which technical-debt-laden teams successfully wrote off their debt by substituting newer technology platforms for older ones for product development.[5] However, adopting new technologies shouldn't be seen as a panacea for technical-debt problems. Such technologies often take time to stabilize and might not immediately be conducive to the high-reliability operations some businesses desire.

FIGURE 1. Software teams should map their technical-debt metrics to these three dimensions.

Eight Scenarios

Figure 2 summarizes our recommendations for the eight scenarios spanning the high and low levels of customer satisfaction needs, reliability needs, and the probability of technology disruption.

When both the reliability needs and the customer satisfaction needs are low (the lower-left box in Figure 2), technical debt isn't a significant concern. Software teams can continue their established product development processes without any special attention toward technical debt. These teams must invest in debt-reducing activities only when product functionality growth becomes saturated and no new technology is available to switch to. Otherwise, the teams can continue to accumulate technical debt without significant worry about long-term impacts. They can simply write off the accumulated technical debt by switching to any new technological platform as soon as it's available.

When the reliability needs are high and the customer satisfaction needs are low (the upper-left box in Figure 2), software teams should avoid technical debt. These teams must shun functionality development based on newly released features of a technological platform until the typical initial wrinkles of the new technology are ironed out. Our investigation of the 10-year evolution of an enterprise software product at 69 large client firms showed that teams that avoided technical debt by choosing to be late adopters of functionality had, on average, about 13 times fewer unresolved errors and about seven times lower bug-fixing effort expenditures than teams who incurred technical debt.[2] However, if the probability of technological disruption is high, software teams can relax the policy of being highly selective late adopters and be open to incurring technical debt, especially if it helps to solve tricky performance bottlenecks. Those teams will still need to invest in debt-reduction activities until the new technology matures and is ready for wider deployment.

Software teams in the remaining scenarios in Figure 2 can be more

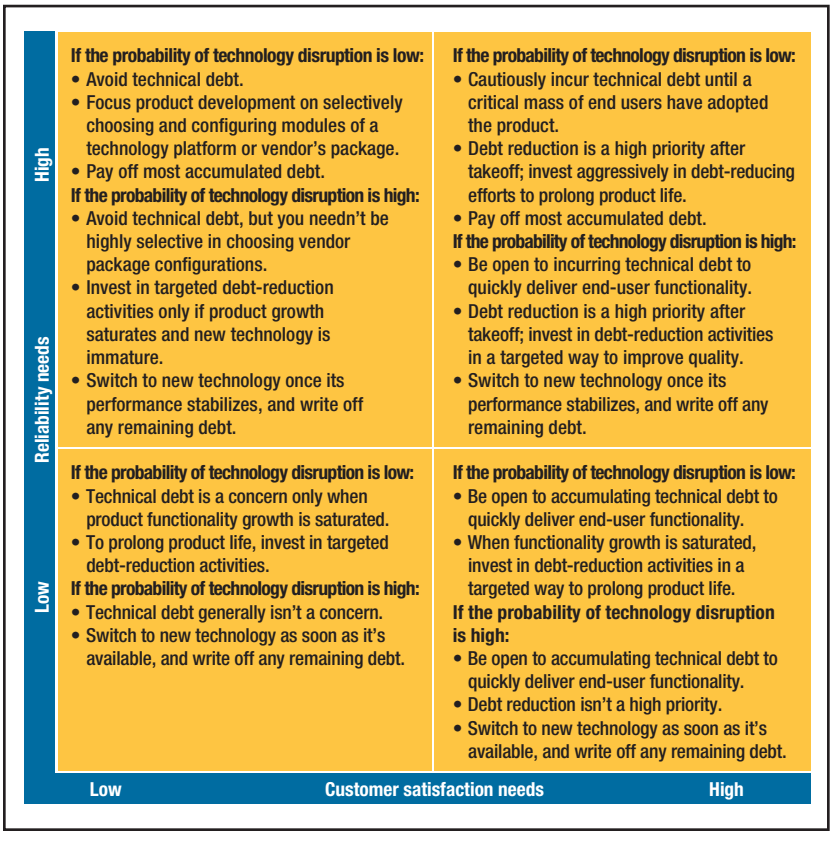

FIGURE 2. Recommendations for managing technical debt.

open to accepting technical debt. What vary between these scenarios are the timing and extent of debt-reduction investments.

Our research shows that when both the reliability needs and the customer satisfaction needs are high (the upper-right box in Figure 2), it's optimal for software teams to accumulate technical debt only until their products have "taken off." A takeoff indicates that the product has achieved a critical mass of adopters who will likely use it throughout its lifetime. Because this installed base of customers expect high reliability, software teams should aggressively pay off most of the accumulated technical debt. Such a strategy is particularly apt if the probability of technology disruption is low, because it will enhance the product's longevity. Aggressively pursuing debt reduction will help software teams maintain higher software quality and continue to add functionality to keep end users happy. On the other hand, if technological disruption is more likely, software teams can reduce debt in a more targeted way. This is because the opportunity exists to write off technical debt by switching to the new technology as soon as its performance stabilizes.

Finally, when the reliability needs are low and the customer satisfaction needs are high (the lower-right box in Figure 2), software teams can be open to incurring technical debt to quickly deliver end-user-demanded functionality. Because the reliability needs are low, teams can postpone technical-debt reduction until product functionality growth becomes saturated. Meanwhile, as in other scenarios, if a new technology platform emerges, software teams can write off all the accumulated debt by immediately switching to that platform.

We believe a technical-debt policy must be based both on the business context of a firm and on the technological environment in which the firm operates. Completely avoiding technical debt is prudent only when the probability of technological disruption is low, reliability needs are high, and prolonging an existing product's life is a high priority. We recommend that software teams optimize the timing and extent of technical-debt accumulation and reduction on the basis of the three dimensions described in this article.

Our research also highlights three main areas in which the empirical evidence is suggestive, but significant questions remain unanswered.

First, what other aspects of the business environment should influence a software team's technical-debt policy? Our qualitative case studies explored the role of resource munificence, technical capability, and the ability to transfer debt to other members of a firm's business ecosystem.[5] Unsurprisingly, we found that firms with abundant resources tended to take on less debt and pay it off faster than firms with scarce resources. But this finding is hard to interpret normatively. Perhaps firms with more resources should take on more debt because they can more easily pay it off later

(much as wealthy individuals take on debt to finance higher-yielding investments). We also found that more technically capable software teams tended to be more debt-averse. But again, this doesn't imply that good developers shouldn't take shortcuts for sound business reasons.

The ability to transfer debt raises even subtler issues, akin to the problem of moral hazard in economics. If you can shift your debt burden to someone else (for example, your customers or partners), should you? The answer might depend on the long-term consequences (for example, to your reputation or to your platform's growth).

Second, we distinguish between modular and architectural maintenance. The former results in code changes localized in software modules; the latter results in changes distributed across module boundaries. Our research on enterprise software packages indicates that these two types of maintenance undertaken by client firms to reduce technical debt have different effects on system failures, and cause unintended side effects, such as conflicts with future vendor releases.[4] Further investigation is needed to understand what debt-reduction strategies are the most effective in different client-specific environments.

Finally, how does adding or reducing technical debt affect a software product development project's evolution? We've observed that high-debt projects tend to increase rapidly in functionality, but then reach a plateau, whereas low-debt projects start off more slowly but ultimately achieve higher functionality.[2] However, a software system isn't a balance sheet; technical debt is an evocative metaphor for a more complex underlying reality. Taking on technical debt means incurring future maintenance obligations, but might also mean taking a fundamentally different path in exploring the system's design space. That path might turn out to be a shortcut, as when a technology disruption enables a team to switch to a new platform and effectively write off its debt. But it might also be a dead end, where efforts to restructure a debt-laden platform never succeed.[5]

References

1. W. Cunningham, "The Wycash Portfolio Management System," *Addendum to Proc. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 92), 1992, pp. 29–30.
2. N. Ramasubbu and C.F. Kemerer, "Managing Technical Debt in Enterprise Software Packages," *IEEE Trans. Software Eng.*, vol. 40, no. 8, 2014, pp. 758–772.
3. B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," *IEEE Software*, vol. 29, no. 6, 2012, pp. 34–42.
4. N. Ramasubbu and C.F. Kemerer, "Technical Debt and the Reliability of Enterprise Software Systems: A Competing Risks Analysis," 2014; http://ssrn.com/abstract=2523483.
5. C.J. Woodard et al., "Design Capital and Design Moves: The Logic of Digital Business Strategy," *MIS Q.*, vol. 37, no. 2, 2013, pp. 537–564.

NARAYAN RAMASUBBU is an assistant professor of business administration at the University of Pittsburgh. Contact him at narayanr@pitt.edu.

CHRIS F. KEMERER is the David M. Roderick Professor of Information Systems at the University of Pittsburgh. Contact him at ckemerer@katz.pitt.edu.

C. JASON WOODARD is an assistant professor of information systems at Singapore Management University. Contact him at jwoodard@smu.edu.sg.