A Balancing Act: What Software Practitioners Have to Say about Technical Debt

Erin Lim, Aquatics Informatics

Nitin Taksande, Novasom

Carolyn Seaman, University of Maryland, Baltimore County

// An interview study aims to characterize technical debt at the ground level, to find out how software practitioners perceive it and to understand the context in which it occurs. //

IN 1992, WARD Cunningham first described technical debt as writing immature or "not quite right" code in order to ship a new product to market faster.[1] Since then, others have refined and broadened the definition to focus on the tradeoffs between expedient short-term decisions and the resulting, potentially crippling, long-term costs.[2–5] Some definitions are artifact-centric (that is, something—not necessarily code—must be fixed); others are more task-oriented (that is, something wasn't done). Moreover, technical debt has been classified by

- type (for example, design, testing, or documentation debt),
- intentionality,
- time horizon (short or long term), and
- degree of focus and strategy.

Most definitions, however, come down to some tradeoff among quality, time, and cost. The metaphor implies that, as long as a project properly manages technical debt, it can help that project achieve goals sooner than would have been possible otherwise.

Managing technical debt involves tracking it, making reasoned decisions about it, and preventing its worst effects. However, if we're looking for concrete ways to manage it, such definitions are very abstract. A more grounded view of technical debt is necessary to guide research about its management and to enable practitioners to reason realistically about their own debt.

So, we've taken an empirical approach to characterize technical debt at the ground level to find out what it "looks like" to the people who deal with it directly: software practitioners. Little research (with one exception[2]) has represented such a perspective. We aimed to gather examples of different ways technical debt manifests itself, to make the concept more concrete through those examples, and to find out whether the types of technical debt that are hypothesized to exist actually do. We also wanted to understand the context in which technical debt occurs, including its causes, symptoms, and effects. Finally, we wanted to know how practitioners currently deal with technical debt.

The Procedure

Our findings are based on interviews with 35 practitioners with varying

0740-7459/12/$31.00 © 2012 IEEE

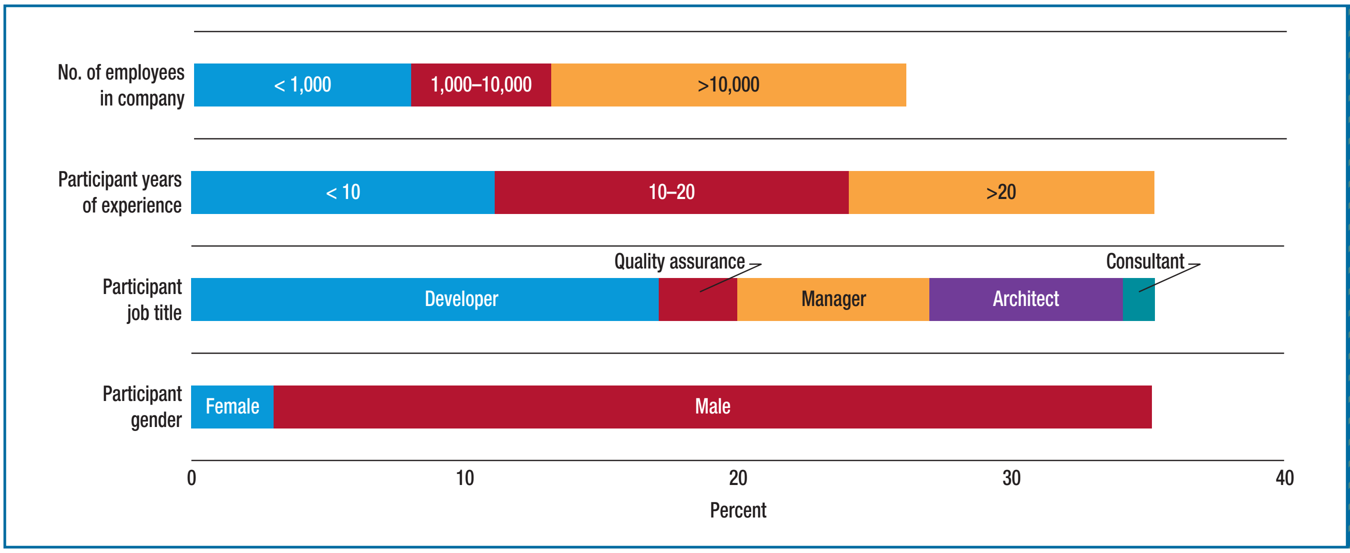

FIGURE 1. The study participants' demographics. The participants comprised 35 practitioners from 26 companies.

industry experience. Two of us (Erin and Nitin) conducted the interviews in two studies using the same interview format and questions (the interview guide we used is available at http://doi.computer.org/10.1109/MS.2012.130). Erin interviewed 19 participants in British Columbia; Nitin interviewed 16 in Maryland. We recruited them from our personal contacts and those of our mentors. The sample wasn't random, but it was broad. Figure 1 presents pertinent demographic information (all self-reported).

The interviews were semistructured, including both specific and open-ended questions.[6] Here are some examples:

- How many years of experience do you have?
- How old is the system?
- Was incurring the debt worthwhile?
- How do you communicate debt to your customers?

The questions began with background information (such as years of experience, current project, and roles and responsibilities on the project) and project-specific information (such as the system overview and market segment). Then we asked the participants to describe a particular example of technical debt, including its properties, causes, and consequences, if any.

We conducted most interviews over the phone, but some were face to face. We audio-recorded and transcribed them all for analysis. We coded the interview data, first using a small set of preformed categories (attributes, symptoms, causes, and consequences), then added subcategories and codes that emerged from the data. Erin coded data using a combination of pen, paper, and Microsoft Excel; Nitin used NVivo (www.qsrinternational.com). Both formulated findings by synthesizing the coded segments pertaining to categories that had substantial data weight. That is, the findings were grounded in the data rather than formulated ahead of time and validated through the data. We analyzed the two sets of interviews separately and then compared and merged the major findings. Here, we present the set of findings that had the most support in the data from both interview sets.

We categorized technical debt according to the types described in "Measuring and Monitoring Technical Debt": testing, defect, documentation, and design debt.[7] Our examples represented all these types (although there was only one example of documentation debt).

The Findings

Because one of the study's secondary goals was to gain insight into how practitioners define technical debt, we took pains to not impose our definitions on the participants. However, 75 percent of the participants weren't familiar with the term, so we presented to them all a brief introduction to the

See www.computer.org/software-multimedia for multimedia content related to this article.

metaphor. We explained it in terms of tradeoffs made and shortcuts taken to meet deadlines, where work had to be done later to clean up the debt.

Only two participants had difficulty understanding the concept; everyone else recognized and understood it immediately. As one participant succinctly put it, "Familiar with it? We live with it every day."

"Familiar with [technical debt]? We live with it every day."

As they elaborated on their views, a picture emerged of technical debt as a complex balancing act. The scenarios and examples they described centered on finding the right balance between different factors for the development team to achieve project goals. We discuss some of these factors here.

Short-Term Benefits versus Long-Term Pain

Most participants didn't talk about technical debt as the result of sloppy programming or poor developer discipline. (Approximately only one-quarter of the examples involved such unintentional debt.[4]) Instead, they focused on it as a result of intentional decisions to trade off competing concerns during development (as other research has also found[2]). Some participants acknowledged that these decisions resulted from short-term thinking in reaction to pressures of the moment.

The technical debt's negative effects tended to be longer term. One participant defined technical debt as "what shortcuts could you take that maybe get you out to the markets faster versus, you know, that might cause you long-term pain." This potential pain included increased complexity, poor performance, low maintainability, and fragile code. Some participants described situations in which nobody wanted to change certain areas of code because "you feared that any time you made a change, you were going to cause something else to go wrong." Others described how technical debt affected their product's performance, reliability, and stability: this led to poor customer perception and often made customers angry or unhappy. Furthermore, some participants reflected that customer support subsequently required increased effort and cost due to having to address the technical debt at the customer site.

In some of the scenarios participants described, the short-term benefits clearly outweighed the future cost. In these cases, system changes were often infrequent, or the software in question was known to be short-lived and noncritical. In other cases, it wasn't so clear whether the right balance had been achieved.

Software Quality versus Business Reality

The participants commonly acknowledged that technical debt is essentially a balance between software quality and business reality. One participant described it this way:

> *Some shortcut has been taken or some less-than-desirable implementation has been done or we've descoped something for one reason or another, and so we've acquired some amount of the work that can or should be done at some point in the future. The reason why I believe it comes into being is simple business situations.... The business reality forces us to make choices at points in time to be able to get the broader outcomes of delivering a solution.*

One impact on software quality that several participants described was system instability—unexpected behavior, or what might be called "fragility." As one participant said, "The debt would be that most things would work, but some customer would try some oddball thing in a new release and it wouldn't work." Some participants stated that the system's unpredictability made it more difficult to estimate the time and effort required to change the software, thus increasing maintenance costs.

The participants described technical debt as compromises on process, scope, testing, and design. Some participants reported that their team decided to cut back on enforcing code and design reviews and writing unit tests because of time constraints. Sometimes, when dropping requirements wasn't an option, they provided solutions that fulfilled the requirements only enough for customer acceptance, knowing that those solutions' implementation still required work.

Technical debt's effects weren't all negative. The tradeoff between delivering a high-quality product or quickly delivering a product depended on the product's value. The participants provided several examples of the business realities that influenced their team's decision to take on technical debt. These situations included

- being contractually obligated to deliver the system under a tight deadline,
- meeting deadlines to integrate with a partner product before release,
- delivering in time for an upcoming

trade show that presented good marketing opportunities,
- getting to market to catch a narrow "shopping window" for their target market—that is, a period of time when customers are evaluating products on the market to determine whether they want to renew their existing software licenses or switch to a different product, and
- developing a working prototype to secure investor funding.

One participant related how incurring technical debt allowed his project to not only capture market share but also to collect early customer feedback. This helped him avoid building something customers didn't want. Another participant stated that if his project didn't take on technical debt, it ran the risk of not being able to deliver at all.

So, although the participants recognized that taking on technical debt always affects software quality in some way, numerous scenarios existed in which the technical debt's market benefits clearly outweighed the hit on quality.

Customers' Expectations versus Their Needs and Wants

Several participants attributed incurring technical debt, indirectly or directly, to customers. One cited reason was that customers often don't know what they want or need. Consequently, some project teams incurred technical debt because they made assumptions without clarifying the requirements. In contrast, others used technical debt to deliver their software quickly to allow customers the opportunity to experiment with it and provide useful feedback.

Another common reason for incurring technical debt was uncertainty over the market's direction. As one participant observed, "It's really hard to get perfect information about what people think they want today … but it's impossible to know where you're going to want to be in five years." Several participants reported that, although they had made the best decisions they could with the knowledge they had at the time, these decisions quickly, and unintentionally, became technical debt when their development teams received new market information.

Several participants described technical debt incurred when customers came up with new requirements during acceptance testing or when the customers' feature wish list was too long to address within the release's time window. Consequently, those participants had to compromise on requirements or quality to meet deadlines, including incomplete implementation of new requirements (requiring the gaps to be filled in later) and inadequate testing. One participant described this as affecting planning: "[The customer] may not be willing to wait for weeks of planning time to get a proper design."

Nonetheless, commitments to the customer always took precedence. These commitments usually led the participants to balance a release's number of features and quality against deadlines.

Measuring technical debt isn't easy because its impact isn't uniform.

Measuring Debt and Communicating Its Consequences

Many participants wanted a way to measure their projects' technical debt. However, they recognized that measuring technical debt isn't easy because its impact isn't uniform. One participant asked,

> *At some point, if it works and it's meeting the needs and there's relatively little or no impact to customers or productivity or maintainability or any of the –ilities or anything that's important to us as a business, then, is there value in doing it [paying back the debt]?*

Another participant stated,

> *[Technical debt] just gets gradually worse and worse; it's only gradually worse than it was the previous day. So, it's always easy to say, well, I'm going to put this off until tomorrow or until the next release or whatever because you can always do that. But, at some point, you get overwhelmed by it.*

The key to characterizing technical debt, then, isn't so much to measure it piece by piece as it's incurred but to measure the cumulative effect over time. The participants struggled to find a way to do this.

Technical debt isn't as visible to external stakeholders, such as customers and management, as it is to developers who work with the code daily. One of the reasons, a participant who worked as a software architect pointed out, is that a software project will more likely acquire technical debt in its infrastructure than in its user interface:

> *Because [the user interface] is so in-your-face, it never suffered because it always had to work, because that's what the customer sees…. All of that infrastructural stuff we would put in*

the business layer ... that's probably where [the technical debt is likely to be], because you could bury it there.

Many participants also found that management didn't recognize the value in addressing technical debt unless doing so provided management with a tangible reward or the customer was paying for it. Similarly, customers weren't easily convinced to allow the development team time to repay its technical debt unless they could derive business value from doing so.

Several participants brought up how to communicate technical debt to customers and other nontechnical stakeholders. Most participants agreed that trying to directly explain the technical debt in the system (or potential technical debt) to customers was a good idea, but they didn't always know how to do that. Some suggested documenting the debt in some way (but they weren't specific).

Participants did mention several instances of successful communication with customers about technical debt. In one example, the development team decided to incur technical debt by providing a bare-bones temporary solution to address new requirements the customer had provided during acceptance testing. When the team explained the tradeoff and told the customer that more work would have to be done later to fully implement those requirements, the customer accepted it. Other participants also said that, in their experience, if stakeholders understood the problem correctly, they were cooperative in extending deadlines to avoid technical debt or, if the debt was already incurred, providing extra time to pay it back.

A Coherent Code Base versus Just Getting Stuff Done

Developers are more likely to view technical debt as bugs, defects, issues, and deficiencies in code, whereas managers usually see it as risks to their projects. So, managers in our study were more willing to incur technical debt, whereas developers generally preferred having no technical debt in the code. An engineering manager we interviewed pointed out that the differing goals between engineering and management contributed to the tension around incurring technical debt:

Engineers don't like [technical debt] because they want to create perfect software. As a technical leader, I'm probably about 60 percent technical and 40 percent business; I also recognize you have to get things to market. You have to hit your windows, right? You've got to look [at] "who is my target customer?" You've got to factor [in] all these things.

However, a developer participant clarified that developers want to create "perfect software" because they must work with the code base on a day-to-day basis. So, they're more aware of issues that could arise from not addressing the technical debt.

Our data also revealed that technical debt often resulted from decisions made by high-level technical and nontechnical managers. The attitudes of the technical leadership and management strongly influenced decisions to incur technical debt. Teams with technical leadership who had "just get stuff done" attitudes were more likely to incur technical debt than those with team leaders who focused on what the software needed to do.

For instance, a participant described a situation in which the team leader was so focused on making his product the best in its market space that he drove the team to develop features quickly. However, he just as quickly abandoned the features to move on to something else when customer response was poor. Unfortunately, this resulted in half-finished, abandoned features that became a maintenance issue for the development team.

Another participant described how a project team incurred significant technical debt by allowing its project priorities to be driven by its largest customer, who favored developing new features over bug fixes. A different participant told how the technical leadership aimed to retain top talent by carving out side projects that developers could build while experimenting with new technologies and languages. This approach ultimately resulted in a "Frankenstein" architecture—that is, architecture that is haphazardly stitched together rather than well designed and coherent—for the company's flagship product.

What We Learned

We found that most project teams now recognize that technical debt is unavoidable and necessary within business realities. So, managing technical debt involves finding the best compromise for the project team. It involves a willingness to accept some technical risks to achieve business goals and an understanding of the need to temper customer expectations to enforce software quality. As one participant described it, "underpromise and overdeliver."

If you can't avoid technical debt, you must manage it. This means recognizing it, tracking it, making reasoned decisions about it, and preventing its worst consequences. Although most participants expressed a desire for better ways of doing all these things, they did share some of their strategies:

- Do nothing—"if it ain't broke, don't fix it"—because the debt might not ever become visible to the customer.
- Use a risk management approach to evaluating and prioritizing technical debt's cost and value. One participant described a policy of allocating

5 to 10 percent of each release cycle to addressing technical debt.
- Manage the expectations of customers and nontechnical stakeholders by making them equal partners and facilitating open dialogue about the debt's implications.
- Conduct audits with the entire development team to make technical debt visible and explicit; track it using a Wiki, backlog, or task board.

We recommend starting with the last strategy from this list. You can't deal with technical debt unless you identify it and communicate with those who can make decisions about it. Various types of analysis tools can identify some hidden debt in the code and design. Soliciting known debt from the developers who are closest to the product will reveal other types of debt.

It seems telling that most of the technical debt our participants described was intentional and didn't include the types of structural decay that are often the object of technical-debt research (code smells,[8] complexity and other quality measures,[9] modularity violations,[10] and so on). One interpretation could be that such code-based types of debt aren't so relevant in industry or that the interview's structure favored reporting of intentional debt. Another possibility is that these types of debt are just hidden better. The latter explanation would strongly motivate the use of static and dynamic analysis to uncover such debt.

The degree to which a project or organization needs to manage its technical debt clearly depends on its context. Often, past experiences of development pain can help guide the search for the most relevant types of debt. The choice of debt management strategy is also context-specific. So, the search for an out-of-the-box technical debt management approach is likely futile. Our findings indicate that to deal with the balancing act related to technical debt, practitioners must make that debt explicit, communicate its costs and benefits among all stakeholders, and manage the inevitable tradeoffs.

ABOUT THE AUTHORS

ERIN LIM is a software developer at Aquatic Informatics. Her research interests include software design, software development processes, and project management. Lim received an MASc in computer engineering from the University of British Columbia. Contact her at erin.lim@aquaticinformatics.com.

NITIN TAKSANDE is a business analyst in the healthcare services domain at Novasom. His research interests include empirical software engineering, healthcare IT, IT project management, and business analysis. Taksande received an MS in information systems from the University of Maryland Baltimore County. Contact him at nitint1@umbc.edu.

CAROLYN SEAMAN is an associate professor in information systems at the University of Maryland, Baltimore County and a research fellow at the Fraunhofer USA Center for Experimental Software Engineering. Her research interests include software maintenance, software metrics, empirical software engineering, and qualitative research methods. Seaman received a PhD in computer science from the University of Maryland. Contact her at cseaman@umbc.edu.

References

1. W. Cunningham, "The WyCash Portfolio Management System," *Addendum to Proc. Object-Oriented Programming Systems, Languages, and Applications*, ACM, 1992, pp. 29–30.
2. T. Klinger et al., "An Enterprise Perspective on Technical Debt," *Proc. Workshop Managing Technical Debt*, ACM, 2011, pp. 35–38.
3. M. Fowler, "Technical Debt," blog, 26 Feb. 2003; www.martinfowler.com/bliki/TechnicalDebt.html.
4. S. McConnell, "10x Software Development," blog, 1 Nov. 2007; http://forums.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx.
5. N. Brown et al., "Managing Technical Debt in Software-Reliant Systems," *Proc. Workshop Managing Technical Debt*, ACM, 2010, pp. 47–52.
6. C. Seaman, "Qualitative Methods," *Guide to Advanced Empirical Software Engineering*, F. Shull et al., eds., Springer, 2008, pp. 35–62.
7. C. Seaman and Y. Guo, "Measuring and Monitoring Technical Debt," *Advances in Computers*, vol. 82, 2011, pp. 25–46.
8. N. Zazworka et al., "Investigating the Impact of Design Debt on Software Quality," *Proc. Workshop Managing Technical Debt*, ACM, 2011, pp. 17–23.
9. J.-L. Letouzey, "The SQALE Method for Evaluating Technical Debt," *Proc. 3rd Int'l Workshop Managing Technical Debt*, IEEE CS, 2012, pp. 31–36.
10. S. Wong et al., "Detecting Software Modularity Violations," *Proc. 33rd Int'l Conf. Software Eng.* (ICSE 11), ACM, 2011, pp. 411–420.