Editor: **Ciera Jaspan**
Google
ciera@google.com

Editor: **Collin Green**
Google
colling@google.com

Defining, Measuring, and Managing Technical Debt

Ciera Jaspan and Collin Green

WARD CUNNINGHAM INTRODUCED the metaphor underlying the term *technical debt* in a 1992 experience report, where he described how his company incrementally extended a piece of financial software:

> *Mature sections of the program have been revised or rewritten many times providing the consolidation that is key to understanding and continued incremental development. [...] Although immature code may work fine and be completely acceptable to the customer, excess quantities will make a program unmasterable, leading to extreme specialization of programmers and finally an inflexible product. 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 [...].*[1]

A few things stand out about this original use of the technical debt metaphor as resonant with a human-centered approach to developer productivity[2]:

- It invokes the properties of humans (for example, the ability to comprehend the product's code) as an important determinant of software engineering (SWE) process outcomes.
- It frames technical debt as arising mostly from nontechnical (business and organizational) factors.
- It focuses on the practical consequences of technical debt for engineering organizations rather than merely the consequences that exist in the code.

Cunningham's example of technical debt in that 1992 report is specific, but the metaphor is more general and centers on decision making and tradeoffs between the speed of delivery and the quality of the product. Over time, the technical debt metaphor has been used and misused in a wide variety of contexts and to describe a wide variety of behaviors, processes, and SWE scenarios. (For example, Cunningham did not intend the technical debt metaphor to excuse engineers writing bad code.[3]) As a result, the term itself—*technical debt*—can be difficult to understand and interpret.

Since 2018, our quarterly engineering satisfaction survey has asked engineers to indicate the extent to which they are "hindered by unnecessary complexity and technical debt," and the percentage of engineers who feel hindered is substantial. Early on, our engineering leads wanted to know more: What is the root cause of technical debt? How can we fix it? When engineers say technical debt is slowing them down, what do they even mean? Is "technical debt" just a catch-all term for anything an engineer dislikes? These questions motivated us to systematically investigate what technical debt means to engineers, how we might measure it, and how we might better manage technical debt. We wanted to deeply understand technical debt as *engineers chose to use the term*, rather than defining it for them, and to work on addressing technical debt from there.

Digital Object Identifier 10.1109/MS.2023.3242137
Date of current version: 18 April 2023

Defining Technical Debt

We took an empirical approach to understand what engineers mean when they refer to technical debt. We started by interviewing subject matter experts at the company, focusing our discussions to generate options for two survey questions: one asked engineers about the underlying causes of the technical debt they encountered, and the other asked engineers what mitigations would be appropriate to fix this debt. We included these questions in the next round of our quarterly engineering survey and gave engineers the option to select multiple root causes and multiple mitigations. Most engineers selected several options in response to each of the items. We then performed a factor analysis to discover patterns in the responses, and we reran the survey the next quarter with refined response options, including an "other" response option to allow engineers to write in descriptions. We did a qualitative analysis of the descriptions in the "other" bucket, included novel concepts in our list, and iterated until we hit the point where <2% of the engineers selected "other." This provided us with a collectively exhaustive and mutually exclusive list of 10 categories of technical debt:

- *Migration is needed or in progress*: This may be motivated by the need to scale, due to mandates, to reduce dependencies, or to avoid deprecated technology.
- *Documentation on project and application programming interfaces (APIs)*: Information on how your project works is hard to find, missing or incomplete, or may include documentation on APIs or inherited code.
- *Testing*: Poor test quality or coverage, such as missing tests or poor test data, results in fragility, flaky tests, or lots of rollbacks.
- *Code quality*: Product architecture or code within a project was not well designed. It may have been rushed or a prototype/demo.
- *Dead and/or abandoned code*: Code/features/projects were replaced or superseded but not removed.
- *Code degradation*: The code base has degraded or not kept up with changing standards over time. The code may be in maintenance mode, in need of refactoring or updates.
- *Team lacks necessary expertise*: This may be due to staffing gaps and turnover or inherited orphaned code/projects.
- *Dependencies*: Dependencies are unstable, rapidly changing, or trigger rollbacks.
- *Migration was poorly executed or abandoned*: This may have resulted in maintaining two versions.
- *Release process*: The rollout and monitoring of production needs to be updated, migrated, or maintained.

We've continued to ask engineers (every quarter for the last four years) about which of these categories of technical debt have hindered their productivity in the previous quarter. Defying some expectations, engineers do not select all of them! (Fewer than 0.01% of engineers select all of the options.) In fact, about three quarters of engineers select three or fewer categories. It's worth noting that our survey does not ask engineers "Which forms of technical debt did you encounter?" but only "Which forms of technical debt have hindered your productivity?" It's well understood that all code has *some* technical debt; moreover, taking on technical debt prudently and deliberately can be a correct engineering choice.[4] Engineers may run into more of these during the course of a quarter, but their productivity may not be substantially hindered in all cases.

The preceding categories of technical debt have been shown in the order of most to least frequently reported as a hindrance by Google engineers in our latest quarter. We don't expect this ordering to generalize to other companies as the ordering probably says as much about the type of company and the tools and infrastructure available to engineers as it does the state of the code base. For example, Google engineers regularly cite migrations as a hindrance, but large-scale migrations are only attempted at all because of Google's monolithic repository and dependency system;[5] other companies may find that a large-scale migration is so impossible that it is not even attempted. A fresh start-up might have few problems with dead/abandoned code or code degradation but many hindrances due to immature testing and release processes. While we do expect there to be differences across companies in how much engineers are hindered by these categories, we believe the list itself is generalizable.

Measuring Technical Debt

Our quarterly engineering survey enables us to measure the rate at which engineers encounter and are hindered by each type of technical debt, and this information has been particularly useful when we slice our data for particular product areas, code bases, or types of development. For example, we've found that engineers working on machine learning systems face different types of technical debt when compared to engineers who build and maintain back-end services. Slicing this data allows us to target technical debt interventions based on the toolchain that engineers are working in or to target specific areas of the company. Similarly, slicing the data along organizational lines allows directors to track their progress as they experiment with new initiatives to reduce technical debt.

However, we find quarterly surveys are limited in their statistical and persuasive power. Each quarter we invite only one third of engineers, and only around one third of them choose to respond. Thus, a team of 100 engineers might yield only nine or 10 survey responses, resulting in wide confidence intervals. This can lead to skepticism around generalizability and a desire to see corroborating, objective metrics.

Another problem is that survey-based measures are a lagging indicator of technical debt: it only emerges in our survey responses once it has become severe enough to hinder engineers. Accordingly, we sought to develop metrics based on engineering log data that capture the presence of technical debt of different types, too. Our goal was then to figure out if there are any metrics we can extract from the code or development process that would indicate technical debt was forming *before* it became a significant hindrance to developer productivity. We ran a small analysis to see if we could pull this off with some of the metrics we happened to have already.

We focused on three of the 10 types of technical debt: code degradation, teams lacking expertise, and migrations being needed or in progress. We selected these because they would require very different types of metrics, and we felt that we might already have data that would serve as accurate indicators of their presence. For example, we hypothesized that many "TODOs" in the code might indicate code degradation, that a large proportion of code written by someone no longer on the team might indicate that the team lacks expertise, and that many bugs referencing words like "migration" or "deprecation" should indicate a migration is needed.

It is quite possible that better technical debt indicator metrics do exist for some forms of technical debt. We only explored objective metrics for three types of technical debt, and we only sought to use existing metrics, rather than attempting to create new metrics that might better capture the underlying concepts from the survey.

An engineer's judgments about technical debt concern both the present state and the possible state.

For these three forms of technical debt, we explored 117 metrics that were proposed as indicators of one of these forms of technical debt. In our initial analysis, we used a linear regression to determine whether each metric could predict an engineer's perceptions of technical debt. We then put all of the metrics into a random forest model to see if the metrics in combination could predict developer's perceptions for each of the three types of technical debt.

The results were disappointing, to say the least. No single metric predicted reports of technical debt from engineers; our linear regression models predicted less than 1% of the variance in survey responses. The random forest models fared better, but they had high precision (>80%) and low recall (10%–25%). That is, these models could identify parts of the code base where a focused intervention could reduce technical debt, but they were also going to miss many parts of the code base where engineers would identify significant issues.

However, it's also possible that such metrics don't exist for other types of technical debt because they are not about the *present* state of a system, but a *relation* between the system's present state and some unimplemented ideal state. An engineer's judgments about technical debt concern both the present state and the possible state. The *possible* states of the world are something that mathematical models cannot incorporate without the modeler's direct intervention. For example, the fact that a project's code base consists entirely of code written in Python 2 is not technical debt in a world where there is no loss of functionality compared to another language or version or outside pressure to migrate. However, in a world where Python 3 is a preferred or required alternative, that same corpus of Python 2 constitutes a needed migration. The *present* state of the world—from the perspective of a model—is identical in these two instances, but the *possible* world has changed. Humans consider the possible world in their judgments of technical debt. If a model were to incorporate explicit rules that capture aspects of the possible world (for example, if a model were designed to count every file in Python 2 as technical debt *because* the human modeler knows Python 3 is an alternative), then the change would be detectable to the model. If we could capture this judgment as it evolves, it could form the basis for better measurements of technical debt.

As it stands, this situation points once again to the key role that human cognition and reasoning play in driving developer productivity: conceiving of the ideal state of a system and using that imagined state as a benchmark against which the current state can be judged might well be central to effective detection and comprehension of technical debt, which are prerequisite to effective management of technical debt. (Not coincidentally, Ward Cunningham was inspired to use financial debt as a metaphor in explaining software development after reading *Metaphors We Live By*, which argues that metaphor is a cognitive tool that humans use to understand and reason about complex or abstract concepts.[3,6])

Managing Technical Debt

While we haven't been able to find leading indicators of technical debt thus far, we can continue to measure technical debt with our survey and help to identify teams that struggle with managing technical debt of different types. To that end, we also added the following questions to our engineering survey:

- To what extent has your team deliberately incurred technical debt in the past three months?
- How often do you feel that incurring technical debt was the right decision?
- How much did your team invest in reducing existing technical debt and maintaining your code?
- How well does your team's process for managing technical debt work?

Combined with the survey items about the types of technical debt that are causing productivity hindrances, these questions enable the identification of teams that are struggling, reveal the type(s) of technical debt they are struggling with, and indicate whether they are incurring too much debt initially or whether they are not adequately paying down their existing debt. These are useful data, especially when teams can leverage them under guidance from experts on how to manage their technical debt. Fortunately, we have such experts at Google. Motivated in part by our early findings on technical debt, an interested community within Google formed a coalition to help engineers, managers, and leaders systematically manage and address technical debt within their teams through education, case studies, processes, artifacts, incentives, and tools. The coalition's efforts have included the following:

- *Creating a technical debt management framework* to help teams establish good practices. The framework includes ways to inventory technical debt, assess the impact of technical debt management practices, define roles for individuals to advance practices, and adopt measurement strategies and tools.
- *Creating a technical debt management maturity model and accompanying technical debt maturity assessment* that evaluates and characterizes an organization's technical debt management process and helps grow its capabilities by guiding it to a relevant set of well-established practices for leads, managers, and individual contributors. The model characterizes a team's maturity at one of four levels (listed here from least to most mature):
  - Teams with a *reactive* approach have no real processes for managing technical debt (even if they do occasionally make a focused effort to eliminate it, for example, through a "fixit").
  - Teams with a *proactive* approach deliberately identify and track technical debt and make decisions about its urgency and importance relative to other work.
  - Teams with a *strategic* approach have a proactive approach to managing technical debt (as in the preceding level) but go further: designating specific champions to improve planning and decision making around technical debt and to identify and address root causes.
  - Teams with a *structural* approach are strategic (as in the preceding level) and also take steps to optimize technical debt management locally—embedding technical debt considerations into the developer workflow—and standardize how it is handled across a larger organization.
- *Organizing classroom instruction and self-guided courses* to evangelize best practices and community forums to drive continual engagement and sharing of resources. This work also includes a

technical talk series with live (and recorded) sessions from internal and external speakers.
- *Tooling* that supports the identification and management of technical debt (for example, indicators of poor test coverage, stale documentation, and deprecated dependencies). While these metrics may not be perfect indicators, they can allow teams who already believe they have a problem to track their progress toward fixing it.

Overall, our emphasis on technical debt reduction has resulted in a substantial drop in the percentage of engineers who report that their productivity is being extremely to moderately hindered by technical debt or overly complicated code in their project. The majority of Google engineers now feel they are only "slightly hindered" or "not at all hindered" by technical debt, according to our survey. This is a substantial change and, in fact, is the largest trend shift we have seen in five years of running the survey.

In the last four years, we've made a concerted effort to better define, measure, and manage technical debt at Google, and it seems like that effort has been fruitful. That's not to say we have no technical debt at Google (cue hearty laughter from Google engineers at the very thought), but zero technical debt is not the goal anyway. We seem to have less technical debt, and—more importantly—fewer instances where engineers are hindered in their work by technical debt. Technical debt isn't unequivocally a bad thing, after all. Just like financial debt, technical debt is one component of a strategy for trading off velocity and (some form of) quality or completeness.

Just as one can thoughtfully and responsibly use financial debt to accomplish goals, one can use technical debt to do so, but it is critical to do so thoughtfully and responsibly. This isn't a new idea. This idea is central to Cunningham's original metaphor, and others have articulated the insight well. For example, Martin Fowler described technical debt as falling into four categories, based on whether it is deliberate versus inadvertent and whether it is prudent versus reckless.[4] Deliberate, prudent technical debt is nothing to fear, and its presence reflects the practicality of how one must develop systems in the real world. Deliberate, prudent technical debt results from effective processes for managing technical debt (ideally, reflecting something like the structural approach in the technical debt maturity model described previously). This view is compatible with Cunningham's original metaphor and intent and particularly with the aspects of the metaphor that connect to software development as an activity that is shaped by and influences human behavior, processes, and organizations.

ABOUT THE AUTHORS

CIERA JASPAN is the software engineering lead for the Engineering Productivity Research team at Google, Mountain View, CA 94043 USA. Contact her at https://research.google/people/CieraJaspan/ or ciera@google.com.

COLLIN GREEN is the user experience research lead for the Engineering Productivity Research team at Google, Mountain View, CA 94043 USA. Contact him at https://research.google/people/107023/ or colling@google.com.

References

1. W. Cunningham, "The WyCash portfolio management system," in *Proc. OOPSLA Exp. Rep.*, Dec. 1992. [Online]. Available: http://c2.com/doc/oopsla92.html
2. C. Jaspan and C. Green, "A human-centered approach to developer productivity," *IEEE Softw.*, vol. 40, no. 1, pp. 23–28, Jan./Feb. 2023, doi: 10.1109/MS.2022.3212165.
3. W. Cunningham, *Debt Metaphor.* (2009). [Online Video]. Available: https://www.youtube.com/watch?v=pqeJFYwnkjE
4. M. Fowler. "TechnicalDebtQuadrant." Martinfowler.com. Accessed: Jan. 31, 2023. [Online]. Available: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html
5. T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google.* Sebastopol, CA, USA: O'Reilly, 2020.
6. G. Lakoff and M. Johnson, *Metaphors We Live By.* Chicago, IL, USA: Univ. of Chicago Press, 1980.