Editor in Chief: **Ipek Ozkaya**
Carnegie Mellon Software Engineering Institute,
ipek.ozkaya@computer.org

A Plea to Tool Vendors: Do Not Mislead How Technical Debt Is Managed

Ipek Ozkaya

ALL SYSTEMS HAVE technical debt, and it has to be continuously managed. Thanks to the past decade of research, software engineering teams have now built an awareness and understanding of technical debt as a concept,[1] have an improved appreciation of code analysis to drive down unintentional implementation mistakes that may lead to technical debt,[2] and even started using some practices to improve its management.[3] In fact, arguably, the managing technical debt research agenda, which was initially summarized at the 2010 National Science Foundation Future of Software Engineering Workshop,[4] has been one of the few bodies of work in software engineering where we have seen upfront and engaged collaboration among academia, industry, and tool vendors.

During this past decade, technical debt researchers and tool vendors did not always get things right, however. Blindsighted by the distorted promise of quantification of all technical debt and its financial impact with the push of a button, researchers, software engineers, and managers initially led themselves to believe that a magic tool would resolve all our technical debt, consequently cost of ownership problems. This led to a lack of separation between what causes debt, what its symptoms are, what might be the accumulating consequences, and where in the system is the debt that needs to be resolved. These confusions are often rooted in a well-intentioned desire to eliminate technical debt altogether. It is even better if a tool can do it. However, dealing with symptoms and root causes of technical debt and removing the debt in the system often necessitates different strategies, different resources, and different quantification approaches.

In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term but that set up a technical context that can make future change more costly or impossible.[5] This definition, which was the outcome of a Dagstuhl seminar attended by researchers in the area, select tool vendors, and industry, has now become accepted by the software engineering community. Rooted in this definition is the recognition that technical debt is about architecture and design tradeoffs and their consequences.[6] The reason why software developers embrace technical debt as a concept is precisely due to its power in expressing architecture and design issues, which they did not have a clear way of doing otherwise.

Tech Debt as a Distinct Issue Category

Technical debt together with defects and vulnerabilities are three high-priority categories of issues that need to be managed to deliver high-quality software successfully (Figure 1). And, in fact, managing technical debt as an issue category more systematically will not only enable concrete data-driven research, but also will allow existing issue management and defect quantification techniques to be purposed for

Digital Object Identifier 10.1109/MS.2021.3102361
Date of current version: 22 October 2021

CONTACT US

AUTHORS

For detailed information on submitting articles, visit the "Write for Us" section at www.computer.org/software

LETTERS TO THE EDITOR

Send letters to software@computer.org

ON THE WEB

www.computer.org/software

SUBSCRIBE

www.computer.org/subscribe

SUBSCRIPTION CHANGE OF ADDRESS

address.change@ieee.org
(please specify *IEEE Software*.)

MEMBERSHIP CHANGE OF ADDRESS

member.services@ieee.org

MISSING OR DAMAGED COPIES

contactcenter@ieee.org

REPRINT PERMISSION

IEEE utilizes Rightslink for permissions requests. For more information, visit www.ieee.org/publications/rights/rights-link.html

quantifying consequences of technical debt.

Defects refer to errors in coding or logic that cause a program to malfunction or to produce incorrect and unexpected results. Most, if not all, defects should be caught through routine testing and code analysis and code conformance checking practices including unit and acceptance tests. Vulnerabilities are weaknesses that can be accessed and exploited by a capable attacker. The criticality of a vulnerability is assessed by determining the risk it presents, where risk is a measure of the likelihood that a threat will exploit the vulnerability coupled with the magnitude of the resultant impact. The higher the risk, the higher the criticality. And, last, technical debt consists of design or implementation constructs that make future changes more costly, issues that neither defects nor vulnerabilities effectively address. There are subtleties in these definitions that drive the reasons why they need to be explicitly managed and overlaps which are unavoidable due to the complex nature of developing and sustaining software systems.

Software engineering and software lifecycle management practices support teams to plan, develop, deploy, and operate systems that meet the organization's business and mission goals. Developing and deploying high-quality software necessitates accepting that defects, vulnerabilities, and technical debt items all need to be actively managed to improve both the quality and the delivery tempo of a system. Consequently, the approach for managing and quantifying technical debt follows that for detecting any other issue in your system that may affect software quality and security. Uncovering technical debt, however, puts a much-needed and neglected emphasis on design and architecture choices and cost of change.

There are by all means situations where technical debt, defects, and vulnerabilities get intertwined. Technical debt as it lingers in the system increases defect proneness and vulnerability risks. Appropriate tool support can be a huge assistance in detecting security violations, implementation errors, and conformance bugs. However, these issues should not be equated and conflated with technical debt. Rather, defects lingering over multiple iterations or an increased number of security problems often represent symptoms of more critical underlying technical debt issues, which need to be examined through an architecture analysis lens and treated accordingly. Effective

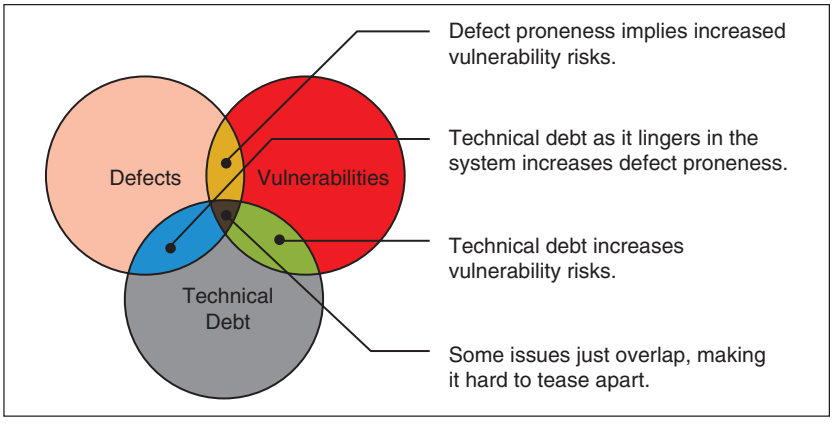

FIGURE 1. The categories of issues that need to be managed in software system development.

software analysis and automating this process is, by all means, critical in getting ahead of unintentional technical debt from creeping into the system. However, such unintentional defects, implementation errors, and conformance bugs are not the debt itself where tool vendors advocate to catch. These are symptoms of underlying lacking software engineering practices.

An early and important observation that the technical debt research community made was the fact that developers in fact talk about technical debt, both within their issue discussions embedded among other issues[7] and as part of their code comments as self-admitted technical debt.[8] Technical debt can also be discovered during other testing and evaluation activities, for example sprint retrospectives or architecture evaluation exercises.

A systematic and commonly accepted approach to document technical debt as it is identified does not exist. Some teams do record it in their issue trackers clearly as technical debt, others put it in their design records,[9] some is documented in the code as self-admitted technical debt, and some organizations assume technical debt resolves itself as an outcome of risk-tracking. Lacking a practical, common mechanism to express and track technical debt is one of the barriers in its effective management. Empowering teams to record technical debt as its special category in issue trackers along with other software development issues such as feature requests, user stories, defects, and vulnerabilities will improve software engineering teams' ability to consistently track and quantify technical debt. Today, explicitly recoding technical debt in issue trackers is a practice only some teams follow. We hopefully soon will see technical debt as part of default setups of any common issue trackers such as Jira, Team Foundation Server, Bugzilla, and the like with a consistent approach. This will be a welcomed change from some of the existing misleading guidance.

Today some divergent practices exist. For example, GitLab, consistent with industry best practices, recommends using technical debt as a label as part of its core default issues workflow.[10] On the other hand, Jira guidance suggests equating all different types of issues that are open as technical debt, which is not only incorrect but also contributes to confusion in practice.[11] When issue and bug tracking software tools embrace the distinction in Figure 1, software engineers will be able to scope and fine-tune technical debt to its relevant architecture and design tradeoff discussions. This will also help ask next step research questions as well, for example, are all self-admitted technical debt in fact technical debt or simply routine to-dos and bugs?

An Open Call to Tool Vendors

The key reason that technical debt and the promise of dealing with it in some objective way resonates with software engineers is because the concept communicates very succinctly the core challenge in software engineering: quality software is developed and sustained as a series of not so trivial tradeoffs that need to be monitored and managed, just like how we manage our money. The reason we all care about technical debt is because we all care about developing high-quality software that serves its intended needs. The much-needed shift in the practice of technical debt management, however, will neither be enabled by researchers nor software engineers. It will be the tool vendors who will enable the concrete management of technical debt,

Digital Object Identifier 10.1109/MS.2021.3102255

improved research, and better technical debt management practices.

Here is my call to tool vendors:

Dear vendors of static code analysis and software quality management tools:

- Please do not market features that detect code conformance bugs, security violations, and implementation errors as technical debt. These are indicative of much more rooted issues in software, are of course critical to avoid, and are symptoms of technical debt.
- Embrace technical debt as an architecture and design issue, and align those features in your software that can help with architecture issues and only refer to those as technical debt. Be blunt and honest about these capabilities.
- Please do not overgeneralize architecture analysis. There are only a few handfuls of architecture aspects that we can truly analyze for with tools today, and they are often limited to select quality attribute concerns around module view of the systems such as modifiability, extensibility, and maintainability. Automated architecture analysis is a hard and still unsolved research and tooling problem.
- Be clear and upfront about what aspects of the architecture issues your features can detect. Analyzing for modifiability and related technical debt differs from analyzing for security or performance.

Dear vendors of issue and bug tracking software tools:

- Please include an issue type of technical debt, maybe even shorthand it as *tech debt*. Yes, I know, there are customization features to allow teams to do that if they chose to do so. But we are together trying to change practice. Including this issue category as part of default configurations will enable software engineering teams to think differently and start improved tracking of technical debt from the start.

The availability of these shifts in tools will have cascading positive effects of giving software engineering teams more concrete ways to identify and express their technical debt. Consequently, more concrete and actionable data related to technical debt will accumulate in our software ecosystems, which will enable us to answer hard questions such as how to quantify technical debt and how to allocate better targeted resources to its management. The next decade of progress in our ability to manage technical debt will be enabled by tool vendors. I am confident they will rise up to the challenge.

Acknowledgments

The ideas I share in this article are influenced by the many conversations I have had the privilege to have within the past decade with many colleagues including Robert Nord, Philippe Kruchten, Stephany Bellomo, James Ivers, and countless others within the Software Engineering Institute and technical debt management researcher community. The good ideas are inspired by them, the mistakes are mine.

References

1. P. Kruchten, R. Nord, and I. Ozkaya, *Managing Technical Debt: Reducing Friction in Software Development*. Reading, MA: Addison-Wesley, 2019.
2. P. Avgeriou et al., "An overview and comparison of technical debt measurement tools," *IEEE Softw.*, vol. 38, no. 3, pp. 61–71, 2021. doi: 10.1109/MS.2020.3024958.
3. M. Keeling, T. J. Halloran, and G. Fairbanks, "Garbage collect your technical debt," *IEEE Softw.*, vol. 38, no. 5, pp. 113–115, 2021. doi: 10.1109/MS.2021.3086578.
4. N. Brown et al., "Managing technical debt in software-reliant systems," in *Proc. FoSER*, 2010, pp. 47–52. doi: 10.1145/1882362.1882373.
5. P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering," Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, Rep. Dagstuhl Seminar 16162, 2016.
6. M. Soliman, P. Avgeriou, and Y. Li, "Architectural design decisions that incur technical debt — An industrial case study," *Inf. Softw. Technol.*, vol. 139, p. 106,669, Nov. 2021. doi: 10.1016/j.infsof.2021.106669.
7. S. Bellomo, R. L. Nord, I. Ozkaya, and M. Popeck, "Got technical debt?: Surfacing elusive technical debt in issue trackers," in *Proc. IEEE/ACM 13th Working Conf. Mining Software Repositories*, 2016, pp. 327–338. doi: 10.1145/2901739.2901754.
8. A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Proc. Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 91–100. doi: 10.1109/ICSME.2014.31.
9. M. Keeling, "Design it! from programmer to software architect," The Pragmatic Bookshelf, 2017. http://media.pragprog.com/titles/mkdsa/architects.pdf
10. "Issues workflow—GitLab docs." https://docs.gitlab.com/ee/development/contributing/issue_workflow.html#technical-and-ux-debt (accessed Aug. 2021).
11. D. Radigan, "3 steps to taming technical debt with Jira," Atlassian, Apr. 1, 2015. https://www.atlassian.com/blog/jira-software/3-steps-taming-technical-debt (accessed Aug. 2021).