

10 Years of Technical Debt Research and Practice: Past, Present, and Future

Marcus Ciolkowski, QAware

Valentina Lenarduzzi, LUT University

Antonio Martini, University of Oslo

DELIVERING INCREASINGLY COMPLEX software-reliant systems demands better ways to manage the long-term effects of short-term expedients. The technical debt (TD) metaphor has gained significant traction as a way to understand and communicate such issues. Almost 25 years after the term was coined in 1992 by Ward Cunningham, and more than 10 years after the first edition of the TechDebt workshop/conference series, we take a brief look at the past, present, and future of TD.

TD's Past: Origins and First Research

TD is a popular metaphor in software engineering. Cunningham introduced it¹ to explain the need for continuous refactoring to his managers: working in an iterative instead of a waterfall model increased project speed, much like borrowing money. Rather than spending time to first understand a problem, a project starts programming immediately with partial comprehension. This way, working code can be delivered faster, and users can provide feedback to better meet their needs. However, as with any debt, if you continually take on more, it is essential to regularly pay back some of the principal. Otherwise, a project can be crippled by interest. In software engineering, this point can be reached, for example, when the cost of new features and maintenance exceed the budget: a project reaches a state of bankruptcy.²

TD interest can take many forms. The most well known is lower maintainability: the upkeep cost is higher than it would be otherwise. However, TD interest can affect other internal and external qualities, such as performance, operability (e.g., increasing costs), and usability (e.g.,

leading users to avoid a product or spend more time completing tasks). All these result in expenses that may strain a project budget and are relevant to consider for TD management. Paying back TD, according to Cunningham, means software needs to be refactored to reflect knowledge gained during the course of a project: refactoring should strive to continuously rewrite software “to look as if we had known what we were doing all along ... and as if it had been easy to do.”¹ In this interpretation, TD includes deficiencies in internal and external software qualities: refactoring may result in redesigning a use case (e.g., because we learn what users really need) and rewriting lines to fix design and code-level issues (e.g., because we learn how a new framework actually should be used).

This original understanding has been both narrowed and broadened. It has been restricted to refer to deficiencies in internal software qualities rather than Cunningham’s more comprehensive view. Fowler’s and McConnell’s definitions are probably the most well known, and they interpret TD as deficits in internal quality, which is similar to the definition derived at a Dagstuhl seminar on the topic.³ At the same time, they broaden the definition to include additional causes and forms of TD: in their view, it can be committed deliberately or inadvertently and prudently or recklessly. This expands Cunningham’s interpretation, as deliberate TD, e.g., taking shortcuts by “writing bad code” to speed up development, is an idea he explicitly opposed. Today, the meaning has been further expanded to “any code that a developer dislikes ... hacky code, code written by novices, code written without consideration of software architecture (so-called big

balls of mud), and code with anti-patterns flagged by static analysis tools.”⁴ All in all, it is well accepted that projects will always have some TD, and taking on TD can be useful, sometimes even required, to achieve success (e.g., for start-ups to attain a critical time to market).

The TD metaphor was quickly taken up by the industry. Together with concepts from the broken windows theory, it paved the way for business and technical people to discuss software quality. Additionally, the emergence of agile concepts and frameworks helped practitioners to embrace managing TD since they all emphasize software quality and continuous refactoring to a certain degree.

Researchers began investigating TD in the early 2000s. Until 2010, only a few articles were published, while from 2010 to 2015, the first larger studies emerged, which contributed to conceptualizing TD.^{5,6} TD has been categorized.⁷ However, only some types of TD have been thoroughly investigated, leaving others in need of more in-depth exploration. TD in code (also named *code debt*) is by far the most studied aspect. A large number of works about mining software repositories, in addition to surveys and case studies, have investigated the impact of postponing refactoring specific issues (e.g., code smells and antipatterns). Financial aspects of TD have also been examined.² Many, however, have not received the same attention, at least from the TD point of view. Research led to commercial tools for identifying TD that were taken up by the industry and thus helped spread the concept. Popular tools, such as SonarQube, developed add-ons to estimate a TD principal based on a code smell and rule violations.

The downside was that they manifested the impression that TD consists of low-level code deficiencies and nothing else.

TD's Present: Better Understanding, the First Evidence, and New Practices

After the first TD workshop, in 2010, publications about the topic began to multiply, especially in the past five years. In 2016, researchers and expert practitioners (e.g., from Siemens and Google) from all around the world participated in the Dagstuhl seminar,³ with the purpose of understanding the state of the art, clarifying the TD concept, and

Researchers and practitioners have focused on several topics, ranging from measuring TD in code bases (e.g., studying several TD indexes) to improving awareness during software organization. Studies have shown that creating TD awareness in organizations is critical to establishing effective identification and management tools and processes. A recent trend has been to investigate debt items that are self-admitted or tagged in code by developers. Ongoing research also focuses on how to enhance software processes, studying how teams work with TD and introduce tracking and management processes.^{9–11} Software companies are immature at managing TD, mostly

Given the hard (if not impossible) task of measuring TD, it is necessary to estimate principal and interest to prioritize TD in practice.¹³ In fact, TD can be beneficial in the short term, but its avoidance or removal is often down-prioritized in favor of feature development, which causes it to be sticky and toxic in some contexts (i.e., it spreads and grows in a contagious manner). Understanding how to estimate current and future TD principal and interest, also in relationship with an organization's road map, is key to contextualizing and making informed decisions. In addition, TD needs a connection to a company's business, as it should be prioritized based on its impact on product value. Different authors have proposed approaches to prioritize TD removal and the development of new features.¹³

An estimated 30% of development resources are wasted because of TD, with peaks of 80%, leading to project crises.

TD's Future: New Perspectives and Well-Known Open Issues

TD research has slowly evolved during the past 10 years. What are the big challenges for the next decade? The articles in this special issue provide a hint: novel practical solutions for well-known issues have been proposed, and gaps have been identified that need to be filled. One study analyzes how TD is especially challenging in the presence of uncertainty, requiring a collaborative approach to managing it. Two surveys of practitioners summarize pitfalls and solutions in specific domains (agile projects and game development). One of them provides insights into how impediments, decision factors, enabling practices, and action diagrams can help implement preventive measures, monitoring techniques, and TD repayment processes. The second analyzes how the

compiling a road map for future research. In addition, a few systematic literature reviews were compiled, and several surveys of practitioners were conducted, shedding light on practices in various countries. A key result is insight into the (costly) impact that TD has on software development: on average, an estimated 30% of development resources are wasted because of TD, with peaks of 80%, leading to project crises and hindering efforts to reach new customers.⁸ In addition, TD seems to have an impact on the development community's social structure and the morale of developers, who often refer to high levels of it as "wading through mud."

using unsystematic approaches and a few tools to identify low-level code debt.⁹ One key issue is that the bulk of the tools focus on detecting TD in code and low-level designs and architectures,¹² and there is little to no support for detecting other kinds of debt. Although there are tools for detecting architectural issues, they are complex, and most do not provide a clear indication of "debt" and, especially, interest related to the issues they are able to find. In addition, current tools lack accuracy (they produce many false positives), completeness (they do not cover many important issues), and refinement (they are not usable in practice).

gaming industry accumulates TD and compares this to other sectors. One article looks into fostering TD prioritization and communication by conceptualizing causes, effects, payment practices, and payment avoidance reasons, with a prioritization schema for technical and nontechnical roles.

Challenges in a data science context are described in another study that summarizes experience with data-driven TD management gained in several industry research projects. Using data to identify (architectural) TD is possible by measuring architectural smells in code: one of the studies investigates how practitioners perceive architectural smells, what maintenance and evolution issues they associate with them, how they introduce them, and how they deal with them in terms of practices and tools. Finally, the special issue contains a study that reports on dealing with TD in procedural languages, drawing from analysis and experience with GO and advocating for improved techniques to identify debt in that and other languages.

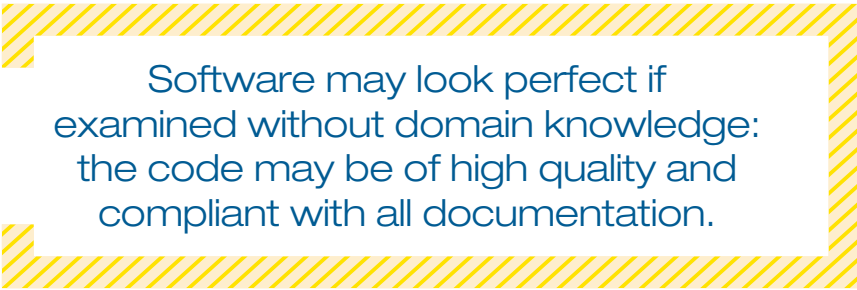
Based on the publications of the past few years, we can expect to see more works investigating the impact TD has on internal qualities, such as faultiness, reliability, and code maintainability, maybe with increased support for nonobject-oriented languages. However, from a practical perspective, the current focus on internal qualities is too restrictive: when arguing for repaying TD, interest should include not only maintainability but other forms, such as operating expenses, opportunity costs, security, user experience problems, and product value. As an example, very recent research highlights how TD is detrimental to the morale (and productivity) of

developers. Another key quality that was prominent at the 2021 TechDebt conference is the impact TD has on security. Such a relationship needs to be further investigated.

Automatically identifying TD items should not be the only focus: practitioners also benefit from support in detecting TD indirectly via indicators, and they need to be able to trace such items to the actual debt in the code. Root cause analysis will most likely remain a manual process in many cases. A TD indicator may be a batch job that slowly but continuously takes longer or consumes

how to avoid and manage TD. In addition, we need to extend the scope of research to higher-level debt, such as architectural-level TD, which is often overlooked even though it is considered one of the more expensive types.^{8,14} Researchers have explored architectural-level TD in several articles to better understand what it is and how to manage it. However, solutions are still preliminary.

Other types of less understood and understudied higher-level TD are requirements and domain-level debt.¹⁵ Requirements debt is typically defined as a gap between known



Software may look perfect if
examined without domain knowledge:
the code may be of high quality and
compliant with all documentation.

more memory (perhaps because of a memory leak or growing data volume). This may go unnoticed for a long time until the system crashes, causing incidents that require an emergency hotfix (usually a quick and dirty solution), which again increases TD. Detecting such indicators early is important.

Researchers should consider other aspects of TD, such as economic and long-term impacts and reasons why debt removal may be postponed. To this purpose, indicators and “smells” should not only consider code bases but include new sources of data, for example, project information, documentation, and versioning tools. Similarly, aspects such as organizational structures and social contexts may provide valuable insight into

requirements and an application (e.g., requirements are deliberately implemented incompletely to meet a deadline, with the intent to do the rest of the work later). Domain debt denotes a disparity between an application and a domain, which may be unknown to the development team. This is often similar to requirements debt but could be seen as incomplete, incorrect, and unidentified requirements and domain models instead of a misalignment of documented stipulations and an implementation.

For example, users may work differently than expected, or an assumption about a domain could be incorrect (e.g., that calendar appointments are always single events). As a consequence, users may lack support for an important workflow

step, or a data model might be difficult to adapt and not match domain rules (e.g., times are expressed as Coordinated Universal Time offsets, which do not facilitate appointment series). Software may look perfect if examined without domain knowledge: the code may be of high quality and compliant with all documentation. However, users might suffer, and the system might be difficult or impossible to extend to new requirements (e.g., enabling appointment series requires redesigning the data model according to time zone information). TD items are not constrained to maintainability but spread, for example, to operability, usability, and business agility. These types of higher-level debt are hard to manage because domain knowledge is often required to spot them, and they are based on the problem space rather than the solution space. That typically makes them cross-cutting and expensive, both in principal and interest, as they involve many stakeholders for analysis and repair.

One of the most important topics to address is how to estimate principal and interest and prioritize TD. After all, the whole purpose of the research in this area is to understand when TD should be taken up, when it should be kept, and when it should be avoided or repaid. This process probably will not be completely automated since context knowledge is often required. However, teams will benefit from methods enabling them to systematically assess TD and make informed decisions. Although a few approaches have been proposed, steps toward more holistic support are needed. For example, other types of debt, such as social and process debt, have been shown to generate a large amount of TD. It is important to thoroughly study the (economic)

impact TD has. Only with clear evidence and a broad collection of practical experiences can we ultimately convince stakeholders that TD needs to be taken into consideration when evolving a system. Researchers and the industry need to work together to collect more evidence.

New technologies are constantly introduced, and the continuously evolving software industry keeps adopting them, often without considering their pros and cons and accumulating more TD than expected. Examples include cloud-native technologies, such as microservices and microfront ends. When adopting them, companies should consider the TD they will incur as they rush to redevelop systems. Solutions to partially mitigate this issue exist, but they need to be thoroughly investigated from the TD viewpoint. As an example, continuous integration/continuous deployment and infrastructure as code might enable companies to simplify the configuration and management of their systems. This potentially helps reduce the TD due to quick configuration patches—or it may increase it if naively done. Another example is the continued hype surrounding machine learning and artificial intelligence. Companies are still not aware of methods to keep the quality and TD of such applications under control.

Another area of future work centers around process support. Since agile frameworks and DevOps are on the rise, it will be important to provide guidance for how to integrate TD management and mitigation strategies into development approaches. For example, agile processes offer an opportunity to adopt good TD management (e.g., through Scrum tools, such as retrospectives, the definition of done, and

continuous refactoring). This is emphasized even more in DevOps since business, development, and operations become integrated and systematic user feedback is heavily stressed. However, there is a risk that developers may find themselves in a feature factory where there is a strong focus on developing new features and almost no awareness and management of TD.

In summary, we have analyzed the past, present, and future of TD management. There have been 10 years of research and practice since the first TechDebt workshop, and TD has been discovered, debated, and analyzed from several points of view. Evidence has emerged about its good and bad effects, and initial approaches have been proposed to manage it, but more research and insights are needed. We believe it will take several more years to bring TD management from adolescence to adulthood. 🍷

References

1. W. Cunningham, "The WyCash portfolio management system," in *Proc. Addendum Object-Oriented Program. Syst., Lang., Appl. (Addendum) - OOPSLA '92*, Vancouver, BC, 1992, pp. 29–30. doi: 10.1145/157709.157715.
2. A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Inf. Softw. Technol.*, vol. 64, pp. 52–73, Aug. 2015. doi: 10.1016/j.infsof.2015.04.001.
3. P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, Eds., "Managing technical debt in Software engineering," Schloss Dagstuhl Leibniz-Zentrum, Dagstuhl, Germany, Dagstuhl Rep., Dagstuhl Seminar 16162, 2016.

4. G. Fairbanks, "Ur-technical debt," *IEEE Softw.*, vol. 37, no. 4, pp. 95–98, July/Aug. 2020. doi: 10.1109/MS.2020.2986613.
5. P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, Nov./Dec. 2012. doi: 10.1109/MS.2012.167.
6. E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *J. Syst. Softw.*, vol. 86, no. 6, pp. 1498–1516, 2013. doi: 10.1016/j.jss.2012.12.052.
7. Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, no. C, pp. 193–220, Mar. 2015. doi: 10.1016/j.jss.2014.12.027.
8. T. Besker, A. Martini, and J. Bosch, "Software developer productivity loss due to technical debt—A replication and extension study examining developers' development work," *J. Syst. Softw.*, vol. 156, pp. 41–61, Oct. 2019. doi: 10.1016/j.jss.2019.06.004.
9. A. Martini, T. Besker, and J. Bosch, "Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations," *Sci. Comput. Program.*, vol. 163, pp. 42–61, Mar. 2018. doi: 10.1016/j.scico.2018.03.007.
10. J. Yli-Huuma, A. Maglyas, and K. Smolander, "How do software development teams manage technical debt? – An empirical study," *J. Syst. Softw.*, vol. 120, pp. 195–218, Oct. 2016. doi: 10.1016/j.jss.2016.05.018.
11. N. Rios, R. O. Spínola, M. Mendonça, and C. Seaman, "The practitioners' point of view on

ABOUT THE AUTHORS



MARCUS CIOLKOWSKI is the lead IT consultant at QAware, Munich, 81549, Germany. Contact him at marcus.ciolkowski@qaware.de.



VALENTINA LENARDUZZI is a researcher at LUT University, Lappeenranta, 15210, Finland. Contact her at valentina.lenarduzzi@lut.fi.



ANTONIO MARTINI is an associate professor at the University of Oslo, Oslo, 0373, Norway. Contact him at antonima@ifi.uio.no.

- the concept of technical debt and its causes and consequences: A design for a global family of industrial surveys and its first results from Brazil," *Empirical Softw. Eng.*, vol. 25, no. 5, pp. 3216–3287, 2020. doi: 10.1007/s10664-020-09832-9.
12. P. C. Avgeriou et al., "An overview and comparison of technical debt measurement tools," *IEEE Softw.*, vol. 38, no. 3, pp. 61–71, May/June 2021. doi: 10.1109/MS.2020.3024958.
13. V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. Arcelli Fontana, "A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools," *J. Syst. Softw.*, vol. 171, p. 110,827, Jan. 2021. doi: 10.1016/j.jss.2020.110827.
14. N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? Manage it? Ignore it? Software practitioners and technical debt," in *Proc. 2015 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE 2015)*, New York: Association for Computing Machinery, pp. 50–60. doi: 10.1145/2786805.2786848.
15. H. Störrle and M. Ciolkowski, "Stepping away from the lamppost: Domain-level technical debt," in *Proc. Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, 2019, p. 8. doi: 10.1109/SEAA.2019.00056.