

Reducing Friction in Software Development

Paris Avgeriou, University of Groningen

Philippe Kruchten, University of British Columbia

Robert L. Nord and Ipek Ozkaya, Software Engineering Institute

Carolyn Seaman, University of Maryland, Baltimore County

// As the global inventory of software grows, technical debt does too. Its management is becoming the dominant driver of software engineering progress. Getting ahead of the software quality and innovation curve will involve establishing technical-debt management as a core software engineering practice. //



MANY LARGE SOFTWARE systems are, like most of the world's state economies, in deep debt. However, this debt is technical, not financial. Major software failures—for example, the recent United Airlines failure and New York Stock Exchange glitch—are being recognized in the popular media as the result

of accumulating technical debt.¹ In 2012, researchers conservatively estimated that for every 100 KLOC, an average software application had approximately US\$361,000 of technical debt, the cost to eliminate the structural-quality problems that seriously threatened the application's business viability.² Although you

can assign some numerical value to technical debt in many ways (still under debate and highly dependent on the context), the undeniable message is that technical debt is real and significant.

Steve McConnell defined technical debt as “a design or construction approach that is expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now.”³ Technical debt's effect on software development is roughly analogous to friction in mechanical devices. The more friction due to wear and tear, lack of lubrication, or bad design, the harder the device is to move, and the more energy you have to apply to get the same effect. Grady Booch said, “There is still much friction in the process of crafting complex software; the goal of creating quality software in a repeatable and sustainable manner remains elusive to many organizations, especially those who are driven to develop in Internet time.”⁴

Technical debt is pervasive; it affects all software engineering aspects, from how we handle requirements to how we deploy to the user base, in how we write code, in what tools we use to analyze code and modify it, and to a greater extent in what design decisions we make at the system and software architecture level. Technical debt even manifests in how we run software development organizations, such as how teams are formed and members interact socially. Technical debt is the mirror image of software technical sustainability, which is “the longevity of information, systems, and infrastructure and their adequate evolution with changing surrounding conditions. It includes maintenance, innovation, obsolescence, data integrity, etc.”⁵



We envision the future of software engineering as revolving around this friction called technical debt: how to avoid it by design, how to identify it, how to cope with it, and how to wisely and purposely incur it to gain commercial advantage. The software development industry has no choice but to treat technical-debt management as a first-class-citizen software engineering practice.

So, dark clouds are on the software industry's horizon, maybe not visible from the academic labs but very menacing from where the CIOs and CTOs sit. Technical debt must be managed through dedicated process and tooling, must become intrinsic in software economics, must be dealt with at the architecture level and through empiricism and data science, and must even be taught in school. It's the next big thing, and it's messy.

A Watershed Moment

The term “technical debt” isn't new—Ward Cunningham introduced it in 1992—and neither are the concepts it covers. For 35 years, software engineers have been examining it under other names: software maintenance, evolution, aging, decay, reengineering, sustainability, and so on. But progress has been piecemeal; the topic wasn't considered “sexy” and was rarely taught in school. Who wants to make a career of maintaining massive amounts of software written by others? New code in a new programming language on the latest platform or “stack” is way more fun and trendier.

Slowly over the last 10 years, many large companies whose success depends on software have realized that technical debt, under any name, is real and is hurting them badly. It has started to translate into financial

terms: not just abstract debt but real costs in the present and the near future, which will impact the financial bottom line. Government organizations, large buyers of software, are recognizing that the cost of software is more than the purchase price. Now, they need justifications of the total cost of ownership, not just initial development costs, from the software industry.

Five years ago, the community found a forum to vent its growing unrest and discuss potential solutions at the First International

Workshop on Managing Technical Debt (www.sei.cmu.edu/community/td2015/series). We're now experiencing a watershed moment, facilitated not only by growing interest in the topic but also by a long, productive history in several software engineering subdisciplines. These research streams are all at a unique point at which they've matured to be part of the answer to the technical-debt question.⁶ (For an in-depth discussion on how these streams contribute to technical-debt management, see “A Systematic Mapping Study on Technical Debt and Its Management”⁷ and “The Financial Aspect of Managing Technical Debt: A Systematic Literature Review.”⁸)

For example, program analysis techniques, although not new, have only recently become sophisticated enough to be useful in industrial contexts and to be incorporated into development environments.⁶ So, they're

positioned to play a role in identifying technical debt, in a way that they weren't a few years ago. Similarly, software quality metrics, qualitative research methods, and software risk management approaches have progressed to the point at which they can contribute to both research in, and practical approaches to, managing technical debt.^{7,8} Building on these streams, scientists are publishing an overwhelming amount of research,⁷ and a lively discourse is taking place in industry through blogs, white papers, and conferences.⁹

Technical debt's effect on software development is roughly analogous to friction in mechanical devices.

The number of research papers published on the topic from both academia and industry has soared since 2010.⁷ Despite the initial focus on source-code-level issues, technical-debt research now encompasses the life cycle from requirements to testing and building, as well as horizontal processes such as versioning and documentation. Several glossaries and ontologies have been proposed to explain and exploit the technical-debt metaphor. The most common terms (with a certain consensus) are principal, interest, and risk (see the sidebar).

To support the aforementioned approaches, people have proposed tooling, both research prototypes and commercial tools. However, only a handful of these tools are dedicated to technical debt, and quantification remains a challenge. Because technical debt originated as a metaphor borrowed from economics and

AN ESSENTIAL GLOSSARY OF SOFTWARE TECHNICAL DEBT



Accruing interest	Additional costs incurred by building new software and depending on some element of technical debt—a nonoptimal solution. These costs accrue over time onto the initial principal, leading to the current principal.
Cause	The process, decision, action, lack of action, or external event that creates a technical-debt item.
Consequence	Technical-debt items' effect on the value, quality, or cost of the current or future state of the system.
Cost	The financial burden of developing or maintaining the product, which is mostly paying the people working on it.
Current principal	The cost of developing a different or “better” solution now.
Initial principal	The cost savings gained by taking some initial approach or “shortcut” in development.
Recurring interest	Additional costs incurred by the project in the presence of technical debt, owing to reduced productivity (or velocity) or induced defects or loss of quality (thus reducing maintainability and evolvability). These are sunk costs, which aren't recoverable.
Risk	The probability or threat that a technical-debt item accumulates such that it hinders system viability.
Quality	The degree to which a system, component, or process meets customer or user needs or expectations (from IEEE Std. 610).
Symptom	An observable qualitative or measurable consequence of technical-debt items.
Technical debt	The complete set of technical-debt items associated with a software system or product.
Technical-debt item	An atomic element of technical debt connecting a set of development artifacts, with consequences for the system's quality, value, and cost. It's triggered by causes related to process, management, context, or business goals.
Value	The business value derived from the product's ultimate consumers—its users or acquirers (the people who are going to pay good money to use it)—and the product's perceived utility.

has predominantly financial consequences, many approaches in industry and academia leverage economic terms as well as theories such as cost-benefit analysis, portfolio management, and real options.^{8,10}

The technical-debt concept resonates well with software developers. Recent results from broad-based industry studies show that developers have a deep understanding of what technical debt is and can articulate its challenges. More important, developers in the trenches are looking for well-defined approaches to help communicate, identify, and resolve

technical debt throughout the development life cycle.¹¹ Research and industry have rarely come this close around a common problem, also supported by tool vendors' increasing focus and interest. If this watershed moment is managed well, it can only accelerate progress.

Technical Debt's Role

Here, we envision where technical debt is headed, from five viewpoints: technical-debt management and tooling; software economics and sustainability; design, architecture, and code; an empirical and data science

basis, and evolving the software engineering curriculum.

Technical-Debt Management and Tooling

Technical-debt management has five core activities.⁷ First, identify the technical debt—for example, through static code analysis or stakeholder workshops on design decisions.

Second, measure the technical debt in terms of benefit and cost. Benefit is usually approximated subjectively, but for cost, many metrics exist that translate into effort.

Even though such metrics are debatable, they can stir discussion among stakeholders and provide a reference point for assessing progress.

Third, prioritize the technical debt—that is, identify the items that have the highest payoff and should be repaid first. This is essentially an investment process that optimally allocates the available limited resources to the most pressing technical-debt items. Economic investment theories such as real options have been used for prioritization.⁸

Fourth, repay the technical debt through refactoring.

Finally, monitor items that aren't repaid, because their cost or value might change over time. This is crucial because certain technical-debt items might escalate to the point of becoming unmanageable.

Four orthogonal management activities support the core activities. The *documentation* of technical-debt items can take different forms, such as design documents, backlogs, or code comments. *Communication* of the documented technical debt should occur among stakeholders, among engineers, and between technical and management stakeholders. Investing in technical-debt repayment over new features or other customer needs requires a delicate discussion with hard evidence. Furthermore, *traceability* between technical-debt items and other software engineering artifacts is crucial to support repayment. For example, repayment might require either knowledge of the affected design decisions and architecture components or renegotiation of system requirements. Finally, technical-debt *prevention* can be prudent when potential debt could accumulate quickly and ominously or when incurring technical debt carries no strategic short-term benefit.

Because of the number of core and supporting activities, it's reasonable to ask how much management is necessary and feasible. Recent experience in implementing technical-debt management has shown that exhaustively following such a process is excessively resource intensive. Thus, realistically, only a portion of the technical debt will be explicitly managed. Rigorously managing selected debt items, especially large, potentially high-impact ones, is worthwhile; the rest can be listed with no further analysis. An alternative is to streamline debt management: use tools or cut corners on things such as estimation and documentation. So, developers need to prioritize technical-debt-management activities, which is somewhat analogous to prioritizing technical-debt items.

We envision the following future:

- Tools will go beyond source code analysis to help identify and measure technical debt at the architecture level, with user input but little required effort. Tools will seamlessly trace technical-debt items to components, design decisions, and requirements and will propose refactorings that take all these levels into account. Tools will also apply economic theories to help stakeholders prioritize technical-debt items and make investments and business decisions.
- Software repositories will be mined for code and architecture smells and refactoring opportunities, and technical-debt items will be documented automatically to facilitate review and discussion among stakeholders. Captured communication within a development community will

help monitor or even prevent technical-debt items.

The core and supporting activities are to some extent part of the daily practice of software development. We've also seen tool support that's effectively integrated in practitioners' daily work.⁷ In addition, researchers have proposed data-mining techniques that provide smarter ways to manage technical debt bottom-up rather than with an overarching top-down model. More important, industrial studies are providing strong evidence of effective processes being elaborated and gradually becoming part of industrial practice.^{10–12}

Software Economics and Sustainability

Software development is a business-driven investment activity. Usually, a divide exists between how executives and managers define and foresee value and how developers' design decisions accelerate or hinder those value propositions. Bridging this divide is possible only through a better understanding of software economics and sustainability.

The world of financial markets employs historical and often reliably collected data. It also has working machinery, the stock market, that helps create models on which analysis can be based. In addition, the variables and data to be collected are often proven by experience.

That framework doesn't translate easily to software development project management and system design and development. Our current software economic models are limited to either treatment of software production as a small percentage of product development costs or oversimplified application of basic financial theories.⁸ However, with the advances in

machine learning and software data analytics, we'll be able to fine-tune development decisions' impact and move to a model in which collecting and analyzing software quality data become seamless. This will bring the challenge and opportunity of building software economic models that help anticipate and plan for how to

data such as change requests, commit histories, capability planning, and velocity tracking will enable better fine-tuning of economic models.

Earlier software economics research had similar aspirations; however, it failed to be relevant to both

avoid debt accumulation is a leading topic in software architecture research. Industry studies showed that practitioners' main concerns also tend to stem from architectural design.¹¹ Research in this area has tried to devise architectural measures, identify architectural dependencies, and examine pattern drift and decay, as well as provide an uncertainty-based approach to prioritizing architectural refactoring opportunities. A key difference between architectural-level and code-based technical debt is that the former is hard to detect just with tools and usually requires interaction with the architects.

Although large intentional architectural debt wasn't what Cunningham had in mind when he proposed the metaphor, we know that such debt can considerably speed up time-to-market and let organizations put their code in users' hands earlier, get feedback, and evolve it. For startup ventures, preserving capital in the early stages is key. The major issue is to clearly identify the corresponding debt and plan for its repayment.

Furthermore, technical debt can't be seen as one big problematic blob in the system. It must be broken down into items connecting development artifacts, with consequences for the system's quality, cost, and value. Each item has a unique location, such as a code or design smell or an architecture decision violation. So, each item has a specific type (for example, a design smell indicates design debt). Such debt must be part of the release planning strategy, at the same level as defects or new features. Failure to do so is what leads to the crippling of some software development efforts.

Solid architectural approaches that take into account short- and long-term quality goals will push

Technical debt occurs constantly, right from the start of a software project.

take on and pay back technical debt.

As a result, the future will hold these advances:

- We'll see the concrete application of technical debt as an investment activity based on the prioritization of technical-debt items. This will be supported by known software product development timeline strategies for assigning business value to intrinsic system qualities such as maintainability and evolvability.
- There will be widespread application of software economic theories and models to software development activities. Instead of shying away from the divide between technical and business stakeholders, we'll develop and employ data-driven approaches that incorporate a deep understanding of the software business's complexities.
- Software development data will be widely available. We're already seeing more and better data as a natural byproduct of improved tools and ecosystems. Easier access and availability of

technical and managerial stakeholders. Evidence from the field demonstrates that both types of stakeholders relate to technical debt and the underlying technical and managerial issues, providing an avenue for enhanced communication and an opportunity for success.¹¹ In addition, recent case studies demonstrated how to incorporate such thinking into software development—for example, to manage modifiability decisions.¹²

Design, Architecture, and Code

Software development is an engineering activity. Technical debt's original definitions led us to think of it as simply bad code quality, and low internal code quality is possibly the prevalent kind of debt. Tools, including static code analyzers, help identify these types of problems and related documentation and testing issues.

Recent technical-debt studies have identified a relationship between architectural shortcuts and potentially higher maintenance and evolution costs.¹² Understanding how to objectively manage architectural concerns and make architectural decisions to

technical-debt management forward in the life cycle:

- Researchers and tool vendors will bridge the gap between implementation environments and architectural models. This will improve communication of architectural decisions and bring architecture closer to implementation, thus linking architecture-level technical debt with source code.
- Looking at multiple views of the architecture, especially views mined from source code and development and deployment infrastructure, will result in earlier recognition of technical debt.
- Using architectural approaches to exploit technical debt as a design strategy will be a conscious, mainstream approach. Architecture evaluations will regularly discuss trade-offs involving technical debt. Reusable architecture refactorings will mitigate risks related to technical-debt accumulation.

Earlier work was manual and error prone. Developers today have access to powerful tools to describe and analyze software development artifacts of all kinds, not only static class structures but also runtime and deployment perspectives.

An Empirical and Data Science Basis

Well-defined benchmarks provide a basis for evaluating new approaches and ideas. Technical debt's evolving definition and its sensitivity to context have inhibited the development of benchmarks so far. An ideal benchmark for technical-debt research would consist of a code base, architectural models (perhaps with several versions), and known technical-debt

items. New approaches to identify technical debt could be run against these artifacts to see how many technical-debt items the approaches reveal.

Similarly, although small-scale case studies are emerging and organizations are developing internal technical-debt initiatives,⁶ the observed advances must be shared as case studies. This will help establish better foundations and an empirical basis for technical-debt research to progress.

To claim success, the future must focus on empirical foundations and data science approaches for analyzing development artifacts and providing inputs to software economic models. We expect the following developments:

- Analysis techniques will incrementally improve to focus on gaps observed in industry—for example, repurposing code quality and metrics to help alleviate architectural issues.
- Tool vendors will support collection of software development data seamlessly without burdening software developers.
- Software economic models and software development data collection and analytics activities will be designed and tooled to integrate easily with software development practices. These models and activities will mimic the financial industry's data-focused approaches and facilitate improved software economic models for technical-debt management.

Recent secondary studies have shown an increasing trajectory of case studies (as well as other types of empirical work) that will help build

consensus and guide the choice of benchmarks.⁷⁻⁹ We're already seeing increased collaboration between researchers and industry, leading to a coming heyday of empirical-research progress.^{10,11} Initial efforts to integrate data collection and analysis into usable tools (for example, SonarQube) have seen some success as well, indicating that progress toward our ambitious vision is under way.

Evolving the Software Engineering Curriculum

The IEEE/ACM software engineering curriculum identifies a software evolution knowledge area, with two knowledge units: evolution processes (six hours) and evolution activities (four hours).¹³ This doesn't provide the full context for introducing the technical-debt concept because it focuses on evolving an existing body of code. Technical debt occurs constantly, right from the start of a software project, and the processes and activities involved in evolution aren't completely distinct or separate from software development processes and activities.

We can't simply add yet another course on technical debt or software evolution. We should progressively introduce students to technical debt throughout the curriculum, by inserting related concepts in courses, exercises, and projects.

Exercises and projects should focus on not only developing new, greenfield applications but also evolving or adding features to existing applications (taking, for example, some open source software), and not necessarily the nicest and cleanest examples. The primary outcome wouldn't be "it runs" or "we can't find any bug." We need to teach a broader range of evaluation criteria in terms of internal software quality,

potential technical-debt items, cost or feature tradeoffs taken, and resource allocation as an investment.

Introducing technical debt progressively throughout the curriculum will help students in these ways:

- They'll be able to explain realistic tradeoffs. Students need to realize early on that there's no one best path forward and that all design choices, even at the code level, must be compromises among multiple tensions, involving different stakeholders.
- Students will be able to use interactive tools to improve software. We have powerful tools for static code analysis; they can assist in developing better code and refactoring code. Getting the program to compile and run once isn't the end, but the start.
- Students will be able to apply estimation models and economic models. Cost versus value drives many decisions; we can use this as an incentive for employing estimation models, not just once but multiple times to feed some of the decision making. Similarly, for economic models, we can show net present value or real options in action.

So, technical debt will add to existing content in courses. We should explicitly introduce economic concepts into the curriculum because the software engineers we train need to be more aware of economic issues and reasoning.

Practitioners, tool developers, researchers, and educators need to work together toward the following multifaceted vision.

New processes and tools that manage technical debt holistically throughout the life cycle will be put into place, enabling communication between stakeholders by evaluating intrinsic quality attributes. As an initial step toward that goal, software development teams should start aggressive initiatives to bring visibility to their existing technical debt.


The marriage between software engineering and economics implied by technical debt will stimulate a fresh wave of research on software economics and sustainability. The vision of a successful technical-debt management initiative implies using technical debt as a strategic software development approach. As an initial step toward the marriage of software engineering and economics, development teams should make economic and business tradeoffs that explicitly influence technical decisions.

The initial focus on the source code level will give way to managing technical debt at the level of architecture decisions and associated tradeoffs and risks. Developers and management shouldn't treat software architecture as an after-the-fact documentation activity but as concretely related to development, testing, and operations activities.

Using software development data for technical-debt analysis will become mainstream, with improved tools targeting developer productivity and efficiency. Validated models will provide an empirical basis for decision making. Instrumenting small changes in development activities can easily enable data collection without overhead for development teams. Such information is essential to establishing an empirical basis for technical-debt management.

Technical debt will become an integral part of the curriculum, not as

a separate course but as a learning thread permeating the course work. As we mentioned before, educators should include discussions of technical debt across the curriculum.

The convergence of efforts on these multiple fronts is necessary to make software development technically and economically sustainable. Otherwise, the friction that slows down the machinery of software evolution will threaten the discipline's ability to maintain the code base on which society depends. 

Acknowledgments

This article is based on research that the US Department of Defense funded and supported under contract FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded R&D center. This material has been approved for public release and unlimited distribution. DM-0002511

References

1. Z. Tufekci, "Why the Great Glitch of July 8th Should Scare You," *The Message*, 8 July 2015; <https://medium.com/message/why-the-great-glitch-of-july-8th-should-scare-you-b791002fff03>.
2. 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.
3. S. McConnell, "Technical Debt," blog, Construx, 1 Nov. 2007; www.construx.com/10x_Software_Development/Technical_Debt.
4. G. Booch, "The Future of Software" (presentation abstract), *Proc. 22nd Int'l Conf. Software Eng. (ICSE 00)*, 2000, p. 3.
5. C. Becker et al., "The Karlskrona Manifesto for Sustainability Design,"

- 2015; <http://sustainabilitydesign.org/karlskrona-manifesto>.
6. F. Shull et al., “Technical Debt: Showing the Way for Better Transfer of Empirical Results,” *Perspectives on the Future of Software Engineering*, J. Münch and K. Schmid, eds., Springer, 2013, pp. 179–190.
 7. Z. Li, P. Avgeriou, and P. Liang, “A Systematic Mapping Study on Technical Debt and Its Management,” *J. Systems and Software*, Mar. 2015, pp. 193–220.
 8. A. Ampatzoglou et al., “The Financial Aspect of Managing Technical Debt: A Systematic Literature Review,” *Information and Software Technology*, Aug. 2015, pp. 52–73.
 9. E. Tom, A. Aurum, and R. Vidgen, “An Exploration of Technical Debt,” *J. Systems & Software*, vol. 86, no. 6, 2013, pp. 1498–1516.
 10. R. Kazman et al., “A Case Study in Locating the Architectural Roots of Technical Debt,” *Proc. 37th IEEE Int’l Conf. Software Eng. (ICSE 15)*, 2015, pp. 179–188.
 11. N. Ernst et al., “Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt,” *Proc. 10th Joint Meeting Foundations of Software Eng.*, 2015, pp. 50–60.
 12. A. Martini, J. Bosch, and M. Chaudron, “Investigating Architectural Technical Debt Accumulation and Refactoring over Time: A Multiple-Case Study,” *Information and Software Technology*, Nov. 2015, pp. 237–253.
 13. *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Program in Software Engineering*, ACM/IEEE, 2004; <http://sites.computer.org/ccse>.

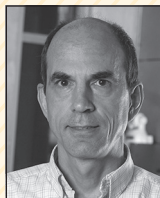
ABOUT THE AUTHORS



PARIS AVGERIOU is a professor of software engineering at the University of Groningen. His research interest is software architecture, especially architecture modeling, knowledge, metrics, and technical debt. Avgeriou received a PhD in software engineering from the Technical University of Athens. He’s on the *IEEE Software* editorial board. Contact him at paris@cs.rug.nl.



PHILIPPE KRUCHTEN is a professor of software engineering at the University of British Columbia. His research interests are software architecture, the software development process, and the technical debt at their intersection. Kruchten received a doctorate in information systems from Ecole Nationale Supérieure des Télécommunications. He’s on the *IEEE Software* editorial board. Contact him at pbk@ece.ubc.ca.



ROBERT L. NORD is a principal researcher at the Software Engineering Institute. His activities focus on agile methods and software architecture at scale; he works to develop and communicate effective methods and practices for software architecture. Nord received a PhD in computer science from Carnegie Mellon University. Contact him at rn@sei.cmu.edu.



IPEK OZKAYA is a principal researcher at the Software Engineering Institute. Her most recent research focuses on building the theoretical and empirical foundations of managing technical debt in large-scale, complex software-intensive systems. Ozkaya received a doctorate in computational design from Carnegie Mellon University. She’s on the *IEEE Software* advisory board. Contact her at ozkaya@sei.cmu.edu.



CAROLYN SEAMAN is an associate professor of information systems at the University of Baltimore, Maryland County and a research fellow at Fraunhofer USA. Her interests encompass software maintenance, metrics, management, and teams. Seaman received a PhD in computer science from the University of Maryland, College Park. Contact her at cseaman@umbc.edu.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.