This article will appear in *IEEE Software*, vol. 33(1), January 2016

Reducing friction in software development

Paris Avgeriou, *University of Groningen, The Netherlands*
Philippe Kruchten, *University of British Columbia, Vancouver, Canada*
Robert L. Nord, *Software Engineering Institute, Pittsburgh, PA, USA*
Ipek Ozkaya, *Software Engineering Institute, Pittsburgh, PA, USA*
Carolyn Seaman, *University of Maryland Baltimore County, Baltimore, MD, USA*

Abstract

Software is being produced at such a rate that its growth hinders its sustainability. Technical debt, as a concept encompassing internal software quality, evolution and maintenance, re-engineering and economics is growing to become dominant as a driver of progress in the future of software engineering. Technical debt spans the entire software engineering lifecycle and its management capitalizes on recent advances made in fields such as source code analysis, quality measurement, and project management. Managing technical debt in the future will be an investment activity applying economics theories, will effectively address the architecture level, will offer specific processes and tools employing data science and analytics to support decision making, and will be an essential part of the software engineering curriculum. Getting ahead of the software quality and innovation curve will inevitably involve establishing technical debt management as a core software engineering practice from theory to its applications.

Keywords: technical debt, software management, software economics

Dark clouds ahead

Software development is an industrial activity, and it can only be sustained if it is economically viable. As the global inventory of software grows, the long-term sustainability of maintaining and evolving it, while producing new solutions, becomes less and less viable. The markets demand new applications and systems, very rapidly; some of these applications are ephemeral and have a “shelf life” of a few months or a couple of years, but some –the most successful ones, and usually the largest ones– must be maintained for the long term, over many years or decades.

This is our biggest hurdle in software engineering: how to cope economically with this rapidly growing software base. The problem is not new, it’s been with us for the last 20 years, but it is becoming acute today. Many large software system are, like most of the world’s state economies, in deep debt, though not a financial debt, but a *technical debt*. CAST Software

Consulting estimated in 2012 that the average software application of 300 KSLOC has about US$1,083,000 of technical debt, defined as the cost to eliminate all structural quality problems that seriously threaten the business viability of the application [5]. Major software failures, for example the recent United Airlines failure and the New York Stock Exchange glitch in July 2015, are being recognized as the result of accumulating technical debt in the popular media. While there are a variety of ways (still under debate and highly dependent on the context) to assign some numerical value to technical debt, the undeniable message is that technical debt exists in real and significant terms.

Steve McConnell defines 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" [10]. In the sidebar "Essential glossary of technical debt," we expand on this definition, articulating elements of the financial metaphor on the software side: principal, interest, both recurring and accruing, causes and consequences of technical debt.

You can think of the effect of technical debt on software development as an analog to *friction* in mechanical devices; the more friction due to wear-and-tear or lack of lubrication or bad design, the harder it is to move the device, the more energy you have to apply to get the same effect. Grady Booch said in 2002: "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."

Technical debt is pervasive; it affects all aspects of software engineering, from the way we handle requirements to the way we deploy to the user base, in the way we write code, in the tools we use to analyse code and modify it, and to a greater extent in the design decisions we make at the system and software architecture level. Technical debt even manifests in the way we run software development organizations, in the social aspect of software engineering, such as when people are not assigned responsibilities matching their skills and expertise. 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." [7].

We envision the future of software engineering as revolving around this new kind of friction called technical debt: how to avoid it by design, how to identify it, how to cope with it when it's there, and how to wisely and purposely incur some technical debt to gain commercial advantage. While the financial debt metaphor provides a good starting point, technical debt behaves differently in many aspects. Consequently, we claim that the software development industry has no choice but to treat technical debt management as one of the first class citizen software engineering practices.

So, there are dark clouds on the horizon of software engineering, maybe not visible from the academic labs, but very menacing from where the CIOs and CTOs sit. Technical debt must be understood, analysed, measured, prioritized, tracked, and even taught in school. It is the *next big thing*, and it is messy.

A Watershed Moment

The phrase “technical debt” is not new, introduced by Ward Cunningham in 1992, and neither are the concepts it covers. We have for 35 years been examining this issue under other names: software maintenance, evolution, aging, decay, re-engineering, sustainability, etc. But progress has been piecemeal, the topic was not considered very “sexy,” and it was rarely taught in school. Who wants to make a career of maintaining massive amounts of software written by others? New code in a brand new programming language on the latest platform or “stack” is way more fun and trendy.

Slowly over the last ten years many large companies, whose success depends on software, are waking up to the realization that technical debt, under this or any other name, is real and is hurting them badly. This technical debt 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, have started to recognize when they are being somewhat misled by the software industry, and are demanding justifications of total costs of ownership, and not only initial development costs.

The community found a forum to vent its growing unrest and discuss potential solutions five years ago at the first *Managing Technical Debt workshop* [8]. We are now experiencing a ‘watershed moment,’ facilitated not only by growing interest in the topic, but by a long productive history in several sub-disciplines of software engineering. These streams of research are all at a unique point of development in which they have matured to be part of the answer to the technical debt question [4] (for an in-depth discussion on how these streams contribute to technical debt management see [1] and [2]). For example, program analysis techniques, while not new, have only recently become sophisticated enough to be useful in industrial contexts, and to be incorporated into development environments [4]. Thus, they are currently positioned to play a role in identifying technical debt, in a way that they were not a few years ago. Similarly, the use of software quality metrics, qualitative research methods and software risk management approaches, have progressed to the point where they can contribute to both research in, and practical approaches to, managing technical debt [1, 2]. Building on these streams, an overwhelming amount of research is being published in scientific literature [1] and a very lively discourse is taking place in industry through blogs, white papers and conferences [9] .

The number of research papers published on the topic from both industry and academia has soared since 2010 [1]. Despite the initial focus on source code level issues, research on technical debt now encompasses the entire lifecycle from requirements to testing and building, as well as horizontal processes like versioning and documentation. There have been several glossaries and ontologies proposed to explain and exploit the metaphor of technical debt; the most commonly used terms (with a certain consensus) are ‘principal,’ ‘interest’ and ‘risk’ (see sidebar). Tooling has also been proposed to support the aforementioned approaches, both research prototypes and commercial tools, although only a handful of these tools are dedicated to technical debt and quantification remains a challenge. Since technical debt originated as a metaphor borrowed from economics and has predominantly financial consequences, many approaches in industry and academia leverage economics terms as well as theories like Cost-Benefit analysis, Portfolio Management and Real Options [2, 11].

The concept of technical debt 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 the challenges they observe related to it. More importantly, software developers in the trenches are looking for well-defined approaches to help communicate, identify and resolve technical debt throughout the software development life cycle [3]. It is rare that research and industry come this close around a common problem, also supported by tool vendors’ increasing focus and interest. It is a watershed moment that can only accelerate progress if managed well. In the following sections, we present our vision on where the progress will take us in the future, from five viewpoints: technical debt management process and tooling, software economics and sustainability, software architecture and design, empirical and data science basis, and software engineering education.

Technical debt management process and tooling

Technical debt affects the entire software engineering lifecycle. There are a number of core sub-processes that could be used to manage technical debt [1]. First, technical debt needs to be **identified,** for example**,** through static code analysis or stakeholder workshops on design decisions. Then it needs to be **measured** in terms of benefit and cost. Benefit is usually approximated with subjective methods but for cost there are many metrics proposed that translate into effort. Even though such metrics are debatable, they are able to stir discussion among stakeholders and provide a point of reference for assessing progress. Next, technical debt needs to be **prioritized,** that is, identify items that have the highest payoff and should be repaid first. This is in essence an investment process, where the available limited resources need to be optimally allocated to the most pressing technical debt items. Economic investment theories like real options have been used to perform prioritization [2]. Next comes the actual **repayment** of technical debt, through refactorings. Items that are not repaid need to be **monitored** as their cost or value may change over time. This is crucial as certain technical debt items may escalate to the point of becoming unmanageable.

There are also management sub-processes that are orthogonal and play a supporting role to the core ones above. The **documentation** of technical debt items can take on a number of forms, such as design documents, backlogs, or code comments. Documenting technical debt is a prerequisite for **communicating** it 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 knowledge of design decisions and architecture components that are affected, or re-negotiating system requirements. Finally, **prevention** of technical debt before it occurs can be prudent in cases where potential debt can accumulate quickly and ominously, and where incurring technical debt does not carry a strategic short-term benefit.

This list of core and supporting sub-process is long, so it is reasonable to ask how much management is necessary and feasible. Recent experience in implementing technical debt management processes has shown that exhaustively following such process is excessively resource-intensive, so 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, while the rest can be listed with no further analysis. An alternative approach is to streamline debt management, i.e. use tools or cut corners on things like estimation and documentation. So there is a process of prioritizing technical debt management activities, in some ways analogous to the process of prioritizing technical debt items, that must be investigated and carried out in practice.

We envision a future where:

- Tools will emerge that go beyond source code analysis to help identify and measure technical debt at the architecture level, with input from users 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 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 assist monitoring technical debt items or even preventing them from occurring.

The core and the supporting processes are to a certain extent part of daily practice. We have also seen tool support that is effectively integrated in the daily work of practitioners [1]. In addition data mining techniques were proposed that provide smarter ways of managing technical debt without an underlying top-down theory or model. More importantly industrial studies are being published [3, 11, 12] that provide strong evidence of effective processes being elaborated and gradually becoming part of industrial practice.

Software Economics and Sustainability

Software development is a business-driven investment activity. There is more often than not a divide between how executives and managers define and foresee value and how software developers can accelerate or hinder those value propositions through their design decisions. Bridging this divide is only possible through a better understanding of software economics and sustainability.

In the world of financial markets, there is historical and often reliably collected data, there is working machinery, the stock market, that helps create models that analysis can be based on, and variables and data to be collected are often proven by experience. This framework does not translate easily to software development project management and system design and development. Our current software economic models are limited to either treating software production as a small percentage of product development costs, or over-simplified application of basic financial theories [2]. However, with the advances of machine learning and software data analytics techniques, we will be able to better fine tune the impact of the development decisions and move to a mental model where collecting and analyzing software quality data is a seamless activity. This will bring the challenge and opportunity of building software economic models that help anticipate and plan for how to take on and pay back technical debt. As a result the future will hold:

- Concrete application of technical debt as an investment activity based on prioritization of different technical debt items. -- This will be supported by known software product development timeline strategies for assigning business value to intrinsic system qualities like maintainability and evolvability.

- Well-known application of software economic theories and models to software development activities -- Instead of shying away from the divide between technical and business stakeholders, we will develop and employ data-driven approaches that incorporate a deep understanding of the complexities of the software business.

- Availability of software development data. -- We are already seeing better and more data being a natural byproduct of improving tools and ecosystems. Easier access and

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

Earlier work in software economics had similar aspirations, however, failed to be relevant to both technical and managerial stakeholders. Evidence from the field demonstrates that [3] both technical and managerial stakeholders relate to technical debt and the underlying technical and managerial issues, providing an avenue for enhanced communication and an opportunity for success. In addition, recent case studies demonstrate how to incorporate such thinking into software development, for example for managing modifiability decisions [11].

Design, architecture and code

Software development is an engineering activity. Original definitions of technical debt 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, assist in identifying these types of problems and related issues with documentation and testing.

Recent studies investigating technical debt 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 avoid debt accumulation is among the leading topics in software architecture research. Industry studies show that the leading concerns of practitioners also tend to stem from an architectural design root [3]. Research in this area includes efforts to come up with architectural measures, identification of architectural dependencies, and examination of pattern drift and decay, as well as providing an uncertainty-based approach to prioritizing architectural refactoring opportunities. A key difference, as compared to code-based technical debt, is that architectural level technical debt is hard to detect just with tools, and most often requires interaction with the architects.

Although large intentional architectural debt was not what Cunningham had in mind when he proposed the metaphor, we know that these deliberate debts can considerably speed up time-to-market, allow an organization to put its code in the hands of its end-users earlier, get feedback, and evolve it. For startup ventures, it is key to preserve capital in early stages. The major issue is to clearly identify the corresponding debt, and plan for its repayment. Furthermore, technical debt cannot be seen as one big problematic blob in the system, but it needs to be broken down into ‘items’ of technical debt connecting development artifacts with consequences for quality, cost, and value of the system. Each item has a unique location, such as a code or design smell, or an architecture decision violation. Consequently each item has a specific type (e.g., 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 some software development effort to situations where all development is crippled.

Solid architectural approaches that take into account short-term and long-term quality goals will push technical debt management forward in the lifecycle:

- Researchers and tool vendors will bridge the gap between implementation environments and architectural models. This will improve communication of architectural decisions, bring architecture closer to implementation and thus link technical debt at the architecture level with source code.

- Looking at multiple views of the architecture, especially views mined from source code and development and deployment infrastructure will result in recognizing technical debt earlier.

- Using architectural approaches to take advantage of technical debt as a design strategy will be a conscious, mainstream approach, while trade-offs involving technical debt will be commonly discussed during architecture evaluations. Risks that relate to accumulating technical debt will be mitigated through reusable architecture refactorings.

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 runtime and deployment perspectives as well.

Empirical and data science basis

Well-defined benchmarks provide a basis against which new approaches and ideas can be evaluated. The evolving definition of technical debt 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, with known technical debt items. New approaches for identifying technical debt could be run against these artifacts to see how many technical debt items they reveal. Similarly, while small scale case studies are emerging and organizations are starting to develop their own internal technical debt initiatives [4], the advances observed need to be shared as case studies. This will contribute to establishing better foundations and an empirical basis for work on technical debt to progress.

In order to claim success, the future has to focus on empirical foundations and data science approaches for analyzing development artifacts and providing inputs to software economics models. We expect to see:

- Incremental progress in improving analysis techniques to focus on gaps observed in industry, for example repurposing code quality and metrics to help alleviate architectural issues
- Tool vendors support collection of software development data seamlessly without burdening software developers.
- Software economic models and software development data collection and analytics activities are designed and tooled to integrate easily with software development practices, mimicking the data focused approaches in the financial industry and facilitating improved software economic models for technical debt management.

Recent secondary studies [1, 2, 9] have shown an increasing trajectory of case studies (as well as other types of empirical work) that will help to build consensus and guide the choice of benchmarks. We are already seeing increased collaboration between researchers and industry, leading to a coming heyday of empirical research progress [3, 11]. Initial efforts to integrate data collection and analysis into usable tools (e.g., SonarQube) have seen some success as well, indicating that progress towards our ambitious vision is underway.

Evolving the software engineering curriculum

Technical debt is a concept that should be introduced in a full computer science or software engineering curriculum. The curricula for Computer science and Software engineering developed by IEEE/ACM [6] identifies a knowledge area called *Software evolution*, with 2 knowledge units: *evolution processes* (6 hours) and *evolution activities* (4 hours), but this is not very satisfactory, as it focuses on evolving an existing body of code. We know that technical debt is introduced constantly, right from the start of a software project, and the processes and activities involved in evolution are not completely distinct or separated from the processes and activities of software development.

We cannot simply add yet another course on “technical debt” or “software evolution”, but we should progressively introduce students to the concept of technical debt throughout of the curriculum, by inserting some of these concepts in courses, exercises and projects.

Exercises and projects should not solely focus on developing new, green-field applications, but on 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 would not be “it runs” or “we cannot find any bug.” We need to teach a broader range of evaluation criteria in terms of internal software quality, potential technical debt items, cost/features trade-offs taken, and resource allocation as an investment.

Introducing technical debt progressively through a computer science or software engineering curriculum will enable students of the future to:

- *Explain realistic tradeoffs.* Students need to face early the reality that there is not one best path forward, and that all design choices, even at the code level, have to be compromises amongst multiple tensions, involving different stakeholders.
- *Use interactive tools to improve software.* We have powerful tools to do static analysis of the code and these tools can assist in developing better code, and refactoring code. Getting “the program to compile and run” once is not the end, only the start.
- *Apply estimation models and economic models.* Since much of the decisions are driven by cost *vs.* value, we can use this as an incentive to use 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 would *add* to existing content in courses. We should explicitly introduce economic concepts into the curriculum as the software engineers we train need to be more aware of economic issues and reasoning.

Conclusion: Forging ahead debt-free

This article is a “call for action” for the communities of practitioners, tool developers, researchers and educators to work together towards a multi-faceted vision:

- **Process and tools**: New processes and tools that manage technical debt holistically throughout the lifecycle will be put into place, enabling communication between stakeholders by evaluating intrinsic quality attributes.

Software development teams should start aggressive initiatives to bring visibility to their existing technical debt as a first step towards this goal.

- **Software economics**: The marriage between software engineering and economics implied by technical debt will stimulate a fresh wave of work on software economics and sustainability.

The vision of a successful technical debt management initiative implies using technical debt as a strategic software development approach. Software development teams should make economic and business trade offs that influence technical decisions explicitly as a first step towards this goal.

- **Software architecture**: 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.

Software architecture should not be treated as an after the fact documentation activity, but concretely related to development, testing and operations activities.

- **Empirical and data science basis:** Utilizing software development data for technical debt analysis will become mainstream with improved tools targeting software developer productivity and efficiency. Validated models will provide an empirical basis for decision making.

Instrumenting small changes in software development activities can easily enable data to be collected without overhead to development teams. Such information is essential in making progress in establishing an empirical basis for technical debt management.

- **Education**: Technical debt will become an integral part of the curriculum not as a separate course but as a learning thread permeating through the course work.

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 codebase that society depends on.

Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002511.

References

[1] Z. Li, P. Avgeriou, P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, Vol.101, March 2015, pp. 193-220.

[2] Ar. Ampatzoglou, Ap. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Information and Software Technology*, Vol. 64, August 2015, pp. 52-73.

[3] N. Ernst, S. Bellomo, I. Ozkaya, R. Nord, I. Gorton, “Measure it? Manage it? Ignore it? Software Practitioners and Technical Debt,” *Conference on the Foundations of Software Engineering*, Aug.30-Sept.4, 2015, ACM.

[4] F. Shull, D. Falessi, C. Seaman, M. Diep, L. Layman. “Technical Debt: Showing the Way for Better Transfer of Empirical Results,” in: J. Münch and K. Schmid (eds.), *Perspectives on the Future of Software Engineering*. Berlin: Springer Verlag, 2013, pp.179-190.

[5] B. Curtis, J. Sappidi, A. Szynkarski. “Estimating the Principal of an Application’s Technical Debt”, *IEEE Software*, Nov/Dec 2012, pp. 34-42.

[6] ACM/IEEE joint task force, *Computing Curriculum - Software Engineering* v.3.1, ACM/IEEE February 6th, 2004.

[7] Ch. Becker, et al (eds), *The Karlskrona Manifesto for Sustainability Design and Software*, 2015, http://bit.ly/RE14Manifesto

[8] Technical debt workshop series. *http://www.sei.cmu.edu/community/td2015/series/*

[9] E. Tom, A. Aurum, R. Vidgen, “An exploration of technical debt.” *J. of Systems & Software*, vol. 86 (6), 2013, pp. 1498–1516.

[10] S. McConnell, Technical debt. 2007. Retrieved from http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx

[11] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A Case Study in Locating the Architectural Roots of Technical Debt," presented at the 37th IEEE International Conference on Software Engineering (ICSE), 2015, pp. 179-188.

[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* (2015).

Sidebar: Essential glossary of software technical debt

Accruing interest	Additional costs incurred by building new software depending on an element of *technical debt*, a non-optimal solution. These accrue over time into the initial *principal* to lead to the current principal
Cause	The process, decision, action, lack of action, or external event that triggers the existence of a *technical debt item*.
Consequence	A *consequence* is the effect on the value, quality or cost of the current or the future state of the system associated with *technical debt items*
Cost	Financial burden of developing or maintaining the product, which is mostly paying the people working on it.
Principal	The cost savings gained by taking some initial approach or “shortcut” in development (the initial principal). Or the cost it would take now to develop a different or “better” solution (the current principal).
Risk	The probability or threat that a technical debt item accumulates to the extent that it hinders system viability.
Recurring interest	Additional costs incurred by the project in the presence of technical debt, due to reduced productivity (or velocity), induced defects or loss of quality (maintainability and evolvability). These are sunk costs, that are not recoverable.
Symptom	An observable qualitative or measurable *consequence* of *technical debt items*
Quality	The degree to which a system, component, or process meets customer or user needs or expectations (from IEEE std. 610)
Technical debt Item	One atomic element of *technical debt* connecting: (1) a set of *development artifacts*, with (2) *consequences* on quality, value and cost of the *system* and triggered by (3) some *causes* related to process, management, context, or business goals.
Technical debt	The complete set of *technical debt items* associated with a software system or product.
Value	The business value derived from the ultimate consumers of the product: its users, or acquirers, the people who are going to pay good money to use it, and the perceived utility of the product.

Authors' bios:

Paris Avgeriou is professor of Software Engineering at the University of Groningen, the Netherlands. His interests lie in the area of software architecture, with strong emphasis on architecture modeling, knowledge, metrics and technical debt. He works towards closing the gap between industry and academia. paris@cs.rug.nl

Philippe Kruchten is professor of Software Engineering at the University of British Columbia, in Vancouver, Canada. His interests are in software architecture, the software development process, and technical debt at the intersection of the former two. pbk@ece.ubc.ca

Robert L. Nord is principal researcher at the Software Engineering Institute, Pittsburgh, PA, USA. He's engaged in activities focusing on agile and architecture at scale and works to develop and communicate effective methods and practices for software architecture. rn@sei.cmu.edu

Ipek Ozkaya is principal researcher at the Software Engineering Institute, Pittsburgh, PA, USA. Her most recent work focuses on building the theoretical and empirical foundations of managing technical debt in large-scale, complex software intensive systems. ozkaya@sei.cmu.edu

Carolyn Seaman is an associate professor of Information Systems at UMBC, Baltimore, MD, USA, and a Research Fellow at the Fraunhofer Center USA in College Park, MD, USA. Her interests encompass software maintenance, metrics, management, and teams. cseaman@umbc.edu