Managing Technical Debt in Software Development: Report on the 2nd International Workshop on Managing Technical Debt, held at ICSE 2011

Ipek Ozkaya,[1] Philippe Kruchten,[2] Robert L. Nord[1], and Nanette Brown[1]
[1]Software Engineering Institute, Carnegie Mellon University, USA
ozkaya, rn, nb@sei.cmu.edu
[2]University of British Columbia, Canada
pbk@ece.ubc.ca
DOI: 10.1145/2020976.2020979
http://doi.acm.org/10.1145/2020976.2020979

Abstract

The technical debt metaphor is gaining significant traction in the software development community as a way to understand and communicate about issues of intrinsic quality, value, and cost. This is a report on a second workshop on managing technical debt, which took place as part of the 33rd International Conference on Software Engineering (ICSE 2011). The goal of this second workshop was to discuss the management of technical debt: to assess current practice in industry and to further refine a research agenda for software engineering in this area.
Keywords: technical debt, software economics, software quality

Introduction

Software developers and corporate managers frequently disagree about important decisions regarding how to invest scarce resources in development projects, especially for internal quality aspects that are crucial to system sustainability, but are largely invisible to management and customers, and do not generate short-term revenue. These aspects include code and design quality and documentation. Engineers and developers often advocate for investments in these areas, but executives question their value and frequently decline to approve them, to the long-term detriment of software projects. The situation is exacerbated in projects that must balance short deadlines with long-term sustainability.

The technical debt metaphor is gaining significant traction in the software development community, as a way to understand and communicate issues regarding intrinsic quality, value, and cost. Ward Cunningham first coined the metaphor in his 1992 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) experience report in defense of relentless refactoring as a means of managing debt [3].

Technical debt is based on the idea that developers sometimes accept compromises in a system in one dimension (e.g., modularity) to meet an urgent demand in some other dimension (e.g., a deadline). Such compromises incur a debt on which interest must be paid and which should be repaid at some point for the long-term health of the project.

There is a key difference between debt that results from employing bad engineering practices and debt that is incurred through intentional decision-making in pursuit of a strategic goal [9]. While technical debt is an appealing metaphor, theoretical foundations for its identification and management are lacking. In addition, while the term was originally coined in reference to coding practices, today the metaphor is applied more broadly across the project life cycle and may include practices of refactoring [5], test-driven development [6], iteration management [4][7][12], software architecture [2][8], and software craftsmanship [10].

The concept of technical debt can provide a basis on which the various stakeholders can reason about the best course of action for the evolution of a software product. As reflected by the composition of our program committee that includes practitioners, consultants, and researchers, this area has significant relevance to practicing software engineers and software engineering.

A first workshop on technical debt was held at the Software Engineering Institute in Pittsburgh on June 2 to 4, 2010. Its outcomes were published as a research position paper [1] summarizing the open research questions in the area.

The goal of the second workshop was to come up with a more in-depth understanding of technical debt, its definition(s), characteristics, and various forms. One objective related to this goal was to understand the processes that lead to technical debt and its indicators, such as degrading system quality and inability to maintain code. A second objective was to understand how to handle technical debt by examining payback strategies and investigating the type of tooling that may be required to assist software developers and development managers to assess its cost.

The Workshop

The workshop was structured to facilitate a dialog between two particular groups: 1) software engineers who need to elicit, communicate, and manage technical debt pertaining to different facets of their projects; and 2) researchers who examine different aspects of technical debt, with particular interest in applying their research in practice and collecting empirical evidence related to their research as it applies to technical debt.

The workshop had four sessions, each dedicated to a specific subject. We had 11 paper presentations and two guided discussions [11]. Below is a summary of these sessions,1 highlighting new insights that emerged.

Maintenance and Code Quality Aspects of Technical Debt

In this session, we had one extended presentation on industry challenges for the research community and five shorter presentations that provided research perspectives on maintenance and code quality.

John Heintz, owner of Gist Labs, discussed his industry experiences with technical debt in presenting the paper "Investigating from Assessment to Reduction: Reining in Millions" he coauthored with Israel Gat. He made the following points on industry challenges for the research community.

- Current practice: Commercial context is typically a business already struggling. Too much code is checked by hand; best practice is to build automation-assisted analysis in continuous integration and static analysis. Automation-assisted analysis applies to different kinds of technical debt: complexity, code coverage, rules violations, duplicate code, and documentation of APIs.
- When performing analysis it is more important to focus on trends than on absolute numbers (e.g., total technical debt ex-

[1] For the complete set of papers see ICSE proceedings at the ACM Digital Library.

ceeds x dollar amount). Trending is more useful as it shows whether improvement is taking place.
- Reducing debt requires more than focusing on code and refactoring; it also involves training, unit test, design principles, and changing work habits.
- In recent efforts a Technical Debt Agile SWAT team was established to focus on enabling Agile to shorten product feature cycles. Duplication and complexity provided low-hanging fruit for reduction. System changes included build script fixes, unit testing infrastructure, version control, and modularizing the system.
- Hard work to come includes scaling to include more teams in learning Agile. Additional study is needed to provide insights into comparing cost and benefits of alternatives, and knowing when to pay back or retire the system.

The following presentations provided a research perspective on maintenance and code quality.
- N. Zazworka, C. Seaman, F. Shull. "Prioritizing Design Debt Investment Opportunities" presented by Nico Zazworka—Fraunhofer Center, USA
- N. Zazworka, M. A. Shaw1, F. Shull, C. Seaman. "Investigating the Impact of Design Debt on Software Quality" presented by Nico Zazworka—Fraunhofer Center, USA
- J. Bohnet, J. Döllner. "Monitoring Code Quality and Development Activity by Software Maps" presented by Johannes Bohnet—Hass-Plattner-Institute at the University of Potsdam, Germany
- R. Gomes, C. Siebra, G. Tonin, A. Cavalcanti, F. Q. B. da Silva, A. L. M. Santos, R. Marques. "An Extraction Method to Collect Data on Defects and Effort Evolution in a Constantly Modified System" presented by Fabio Q. B. da Silva—CIn/Samsung Laboratory of Research and Development – UFPE, Brazil)
- W. Nichols. "A Cost Model and Tool to Support Quality Economic Trade-off Decisions" presented by William Nichols—Software Engineering Institute, USA

The presenters made the following points.
- Design debt: What is design debt – any debt that is related to the design of the system, to the ideal design of the system? Can we find evidence that design debt slows down development? This can be very relevant in acquisition environments. One way to eliminate design debt is to refactor (pay it off). Do god classes (i.e., large classes as defined by Martin Fowler) have an effect on the maintainability of the system? Can we provide guidance on which design debt to pay off first?
- Visualization techniques: The goal of engaging these techniques is to make internal quality more visible to the managers. The visualization technique in the form of a software map is structured according to the modularity of the system. Complex files (as indicated by their McCabe complexity measure) are highlighted in 3D and by color on the map. Some challenges in visualization techniques are integrating time and making the technique fully interactive. The ultimate goal is to provide early warnings to detect costs and risks (e.g., "watch out for this class, it might be growing too big").
- Social and human aspects of software engineering: the goal is to understand how business and organizational decisions create technical debt.
- Cost models: The best strategy is to not incur technical debt. We know 80% of the costs are caused by 20% of the defects.

Discussion on Industry Challenges

In this session, participants discussed the presentations from the previous session and formulated industry problems and challenges based on experiences and limitations in the state of the art.
- The real technical debt lies not in the lines of code. The analysis that can be done based on profiling the code does not show the real insights.
- There is always technical debt, whether in an agile context or not.
- Measurement is important but not most critical. Engineers are concerned with getting the job done and ensuring the system will work. They need an upfront framework before they start building to conduct a what-if analysis. For example, what if we shrink the timeline of a five year project to four years to reduce cost. Can we still be assured the project is on target and will produce the right answer?
- Metrics are desirable for determining where to focus (given limited time and resources).
- Business people need to communicate actual needs, developers need to understand them. Developers understand the technical aspect of risk; business knows the business value. The product owner needs to consolidate multiple viewpoints.
- Many models are incorrect; they are linear. They must be adapted to show more development life-cycle phases.
- The key to success is acknowledging technical debt to support business goals.

Other Forms of Technical Debt

In this session, we had one extended presentation on architecture and four shorter presentations that provided perspectives on definitional framework and other forms of technical debt.

Peri Tarr, from IBM Watson Research, discussed the results of interviewing four technical architects in presenting the paper "An Enterprise Perspective on Technical Debt," she coauthored with Tim Klinger, Patrick Wagstrom, and Clay Williams. She made the following points.
- Financial risk and value are managed in the aggregate and are always the first thing people think of. Whether technical debt can be used as leverage or not is an open question.
- The situational nature of technical debt is the most worrisome. Technical debt is relative to goals, requirements, stakeholders, and ecosystem. In the architects' experience, the decisions were managed ad hoc and were not recorded. They were propagated by tribal memory and nobody went back to evaluate them. The financial costs were obvious and revisited but the cost of technical debt was not clear at all. Stakeholders lacked effective ways to communicate and reason about debt.
- Quality issues were a small subset of the issues. The architects looked at the quality metrics but did not worry about them; issues such as architectural debt were more critical for them. Debt was really important when it was active debt causing critical situations. However it is hard to know whether some problems will become active. Reasoning based on uncertainty is essential to any realistic approach to debt.
- Architects did not reason about the debt in terms of absolute quantifiable measures, but they can do relative measurement like "this one is better or worse than the other one."

The following presentations provided a perspective on a definitional framework and other forms of technical debt.

- N. Brown, R. Nord, I. Ozkaya, P. Kruchten. "Quantifying the Value of Architecting within Agile Software Development via Technical Debt Analysis" presented by Ipek Ozkaya—Software Engineering Institute, USA
- T. Theodoropoulos, M. Hofberg, D. Kern. "Technical Debt from the Stakeholder Perspective" presented by Ted Theodoropoulos, USA
- A. Nugroho, J. Visser, T. Kuipers. "An Empirical Model of Technical Debt and Interest" presented by Ariadi Nugroho—Software Improvement Group, Netherlands
- Y. Guo, C. Seaman. "A Portfolio Approach to Technical Debt Management" presented by Yuepu Guo—University of Maryland Baltimore County, USA

The presenters made the following points.

- Analysis of architectural dependencies provides an empirical basis for making decisions regarding technical debt.
- Technical debt is any gap between the technical framework and the required quality of the system.
- Technical debt is a software risk. Reduce investment risk through diversification.
- Technical debt is an asset to be managed as part of the portfolio.

Discussion on Research Challenges

In this session, participants discussed the presentations from the previous session and formulated research challenge problems.

- Technical debt is not a crisp technical reality. Technical debt needs a mission statement describing what it does and what it needs to do.
- Technical debt is commonly considered a bad thing. But there exist forms that can be strategic. Technical *investment* has a more appealing twist to it.
- Technical debt has been around for a while; this is a multifaceted reality that is related to the following topics: Maintenance, Evolution, Erosion, Aging, Value-Based Software Engineering.
- Different disciplines might need different measurements of technical debt.
- Anything you cannot quantify you discard. Nobody really gets architectural risk as nobody knows how to quantify it.
- Perspective is very important. It's necessary to get a handle on indicators (e.g., god classes).
- Indicators for where to spend money would be useful, such as when it is no longer efficient to carry technical debt and it is time to be repaid.

Summary

The main future directions that were discussed are

- What should the research agenda look like? It should include models to show where technical debt slows development and where it speeds it up and where the breaking point exists such that it is no longer efficient to carry technical debt.
- A collection of examples of technical debt—having a catalog of examples from various stakeholder points of view could help us develop a better taxonomy.
- Creation of a web portal on technical debt, to collect pointers to papers, books, blog entries, discussion, and tools related to the subject, and to foster discussion and collaboration.
- While technical debt has a strong negative connotation, it can also be seen in a more positive light as a tactical investment in a project, something to gain a temporary advantage to later be repaid or not.

Acknowledgments

We extend our thanks to all those who have participated in the organization of this workshop, particularly to the program committee members:

- Eric Bouwers, Technical University Delft, Netherlands
- Yuangfang Cal, Drexel University, USA
- Rafael Capilla, Universidad Rey Juan Carlos, Spain
- Jeromy Carriere, eBay, USA
- Ward Cunningham, AboutUs, USA
- Hakan Erdogmus, Kalemun Research, Canada
- David Garlan, Carnegie Mellon University, USA
- Israel Gat, Cutter Consortium, USA
- Jim Highsmith, ThoughtWorks, USA
- Rick Kazman, University of Hawaii and the Software Engineering Institute, USA
- Tobias Kuipers, Software Improvement Group, Netherlands
- Erin Lim, University of British Columbia, Canada
- Alan MacCormack, MIT, USA
- Steve McConnell, Construx, USA
- Don O'Connell, Boeing, USA
- Raghu Sangwan, Penn State University, USA
- Carolyn Seaman, University of Maryland Baltimore County, USA
- Kevin Sullivan, University of Virginia, USA

Disclaimer

The views and conclusions contained in this document are solely those of the individual creator(s) and should not be interpreted as representing official policies, either expressed or implied, of the Software Engineering Institute, Carnegie Mellon University, the U.S. Air Force, the U.S. Department of Defense, or the U.S. Government.

References

[1] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N., 2010. Managing Technical Debt in Software-Reliant Systems, 2010 FSE/SDP Workshop on the Future of Software Engineering Research.

[2] Brown, N., Nord, R., Ozkaya, I. 2010. Enabling Agility through Architecture, Crosstalk, Nov/Dec 2010.

[3] Cunningham, W. 1992. The WyCash Portfolio Management System. *OOPSLA' 92 Experience Report*.

[4] Cohn, M. 2006. Agile Estimation and Planning, Prentice Hall.

[5] Fowler, M. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

[6] Erdogmus, H., Morisio, M., and Torchiano, M. 2005. On the Effectiveness of the Test-First Approach to Programming. IEEE Trans. Softw. Eng. 31, 3 (Mar. 2005), 226-237.

[7] Highsmith, J. 2009. Agile Project Management 2nd ed. Addison Wesley.

[8] InfoQ: What Color is your Backlog? Interview with Philippe Kruchten, May 02, 2010. Available from: http://www.infoq.com/news/2010/05/what-color-backlog

[9] McConnell, S. 2007. Technical Debt. 10x Software Development [cited 2010 September 17]; http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx

[10] Martin, Robert C. 2008. Clean Code: A Handbook of Agile Software Craftsmanship. Addison Wesley.

[11] Second International Workshop on Managing Technical Debt http://www.sei.cmu.edu/community/td2011/

[12] Sutherland, J. 2005. Future of Scrum: Parallel Pipelining of Sprints in Complex Projects. Proceedings of the Agile 2005 Conference, pp. 90-102.