

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/309606012>

Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)

Article · January 2016

DOI: 10.4230/DagRep.6.4.110

CITATIONS

251

READS

1,775

4 authors, including:



Paris Avgeriou

University of Groningen

332 PUBLICATIONS 7,052 CITATIONS

[SEE PROFILE](#)



Philippe Kruchten

University of British Columbia - Vancouver

290 PUBLICATIONS 13,801 CITATIONS

[SEE PROFILE](#)



Ipek Ozkaya

Carnegie Mellon University

98 PUBLICATIONS 2,673 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Managing technical debt in software Intensive systems [View project](#)



Software Development Methods [View project](#)

Managing Technical Debt in Software Engineering

Edited by

Paris Avgeriou¹, Philippe Kruchten², Ipek Ozkaya³, and
Carolyn Seaman⁴

1 University of Groningen, Groningen, NL, paris@cs.rug.nl

2 University of British Columbia, Vancouver, BC, CA, pbk@ece.ubc.ca

3 Carnegie Mellon University, Pittsburgh, PA, US, ozkaya@sei.cmu.edu

4 University of Maryland, Baltimore County, MD, US, cseaman@umbc.edu

Abstract

This report documents the program and outcomes of Dagstuhl Seminar 16162, “Managing Technical Debt in Software Engineering.” We summarize the goals and format of the seminar, results from the breakout groups, a definition for technical debt, a draft conceptual model, and a research road map that culminated from the discussions during the seminar. The report also includes the abstracts of the talks presented at the seminar and summaries of open discussions.

Seminar April 17–22, 2016 – <http://www.dagstuhl.de/16162>

1998 ACM Subject Classification coding tools and techniques, design tools and techniques, management, metrics, software engineering

Keywords and phrases software decay, software economics, software evolution, software project management, software quality, technical debt

Digital Object Identifier 10.4230/DagRep.6.4.110

Edited in cooperation with Robert Nord

1 Executive Summary

Ipek Ozkaya

Philippe Kruchten

Robert Nord

Paris Avgeriou

Carolyn Seaman

License © Creative Commons BY 3.0 Unported license
© Ipek Ozkaya, Philippe Kruchten, Robert Nord, Paris Avgeriou,
and Carolyn Seaman

The term *technical debt* refers to delayed tasks and immature artifacts that constitute a “debt” because they incur extra costs in the future in the form of increased cost of change during evolution and maintenance. The technical debt metaphor provides an effective mechanism for communicating design trade-offs between developers and other decision makers. When managed effectively, technical debt provides a way to gauge the current maintainability of a system and correct the course when that level is undesirable. While other software engineering disciplines – such as software sustainability, maintenance and evolution, refactoring, software quality, and empirical software engineering – have produced results relevant to managing technical debt, none of them alone suffice to model, manage, and communicate the different facets of the design trade-off problems involved in managing technical debt.



Except where otherwise noted, content of this report is licensed
under a Creative Commons BY 3.0 Unported license

Managing Technical Debt in Software Engineering, *Dagstuhl Reports*, Vol. 6, Issue 4, pp. 110–138

Editors: Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman



Dagstuhl Reports

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Despite recent progress by the research community in understanding technical debt, increased attention by tool vendors on assessing technical debt through code conformance checking, and collaboration with industry in sharing data and challenges, there are several open questions about the role of technical debt in software development. The goal of this seminar was to establish a common understanding of key concepts of technical debt and build a road map for future work in this area to address these open questions.

How do we define and model technical debt? The software engineering community is converging on defining *technical debt* as making technical compromises that are expedient in the short term, but that create a technical context that increases complexity and cost in the long term. While the conceptual roots of technical debt imply an idealized, deliberate decision-making process and rework strategy as needed, we now understand that technical debt is often incurred unintentionally and catches software developers by surprise. Hence, it is mostly observed during maintenance and evolution. Technical debt as a metaphor serves as a strong communication mechanism, but the community now understands that technical debt is also a software development artifact. This overloaded nature creates confusion, especially for newcomers to the field. In addition, there is a risk of associating anything detrimental to software systems and development processes with technical debt. This risk necessitates crisply defining both technical debt and related concepts.

How do we manage technical debt? Managing technical debt includes recognizing, analyzing, monitoring, and measuring it. Today many organizations do not have established practices to manage technical debt, and project managers and developers alike are longing for methods and tools to help them strategically plan, track, and pay down technical debt. A number of studies have examined the relationship between software code quality and technical debt. This work has applied detection of “code smells” (low internal code quality), coupling and cohesion, and dependency analysis to identify technical debt. However, empirical examples collected from industry all point out that the most significant technical debt is caused by design trade-offs, which are not detectable by measuring code quality. Effective tooling to assist with assessing technical debt remains a challenge for both research and industry.

How do we establish an empirical basis and data science for technical debt? Well-defined benchmarks provide a basis for evaluating new approaches and ideas. They are also an essential first step toward creating an empirical basis on which work in this area can grow more effectively. Effective and well-accepted benchmarks allow researchers to validate their work and tailor empirical studies to be synergistic. 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 (TD items). New approaches to identify technical debt could be run against these artifacts to see how well the approaches reveal TD items. Industry needs guidance for how and what data to collect and what artifacts they can make available to enable progress in understanding, measuring, and managing technical debt.

Seminar Format

In this seminar, we brought together researchers, practitioners, and tool vendors from academia and industry who are interested in the theoretical foundations of technical debt and how to manage it from measurement and analysis to prevention. Before the seminar,

the organizers created a blog where attendees could post positions and start discussions to facilitate seeding of ideas.

Before the seminar, the organizers grouped discussions and blog entries into relevant themes that included creating a common definition and conceptual model of technical debt, measurement and analysis of technical debt, management of technical debt, and a research road map for managing technical debt.

Our goal was to make this seminar a working week; hence we had a dynamic schedule. We did not feature any long talks. Each day had three types of sessions. There was a plenary session for “lightning talks,” in which each presenter had 10 minutes for presentation and questions on each day except for the last day of the seminar. The second type of session was for breakout discussions. Breakout sessions focused on themes that emerged from the blog and the goals of the seminar. Participants first discussed these in randomly assigned small groups in order to maximize cross-pollination of ideas. Last, we had plenary discussion sessions to collate and summarize the discussions during the breakouts. At the end of each day, the organizers asked for feedback and adjusted the flow of the following day accordingly. As a result, we dedicated the fourth day of the seminar to an “un-conference” format in which the discussion topic emerged based on the interests and votes of the attendees. The summaries of these sessions are included in Section 5: Open Problems.

The Definition of Technical Debt and a Conceptual Model

At the conclusion of the seminar, attendees agreed on the following working definition of technical debt, which we refer to as the 16162 definition of technical debt:

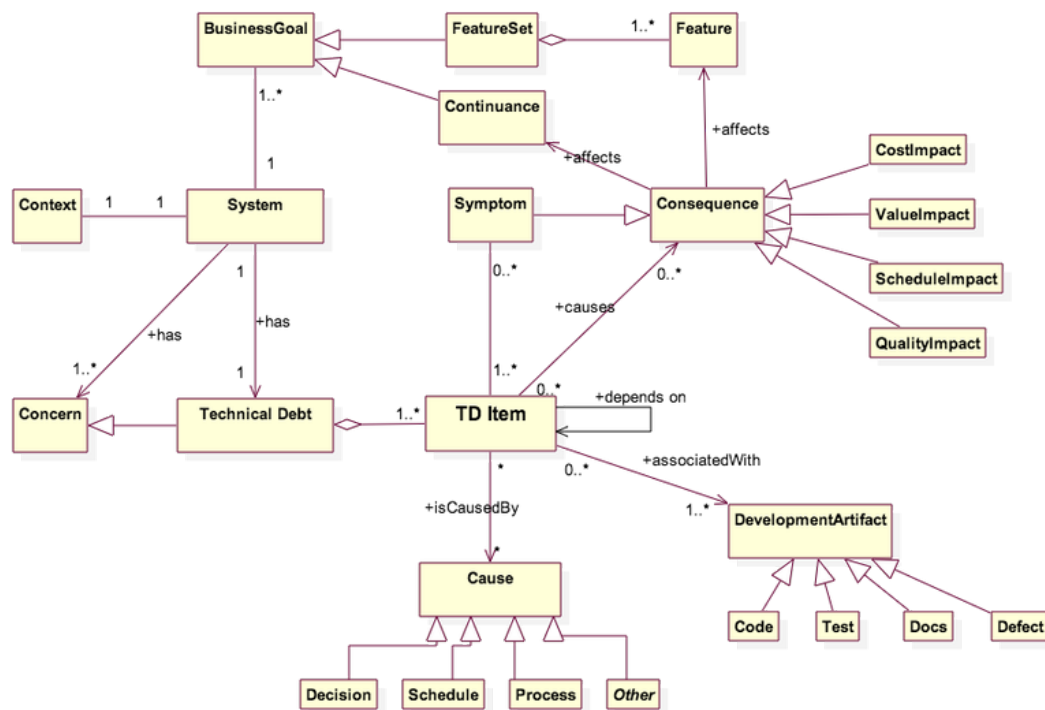
In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.

A significant outcome of the week was the recognition that, similar to other complex software engineering artifacts, technical debt is best described through multiple viewpoints. Concepts related to technical debt in particular should be discussed based on two related viewpoints:

1. the viewpoint describing the properties, artifacts, and elements related to technical debt items
2. the viewpoint articulating the management- and process-related activities to perform, or the different states that debt may go through

Figure 1 shows the initial conceptual model that served as the starting point for discussions. This model helped the group converge on key concepts. Mismatches occurred when the discussions focused on causes that may or may not be input to measurement and analysis. The dynamic view is intended to articulate these aspects.

The technical debt associated with a software-intensive system is composed of a set of TD items, and this technical debt is one of many concerns associated with a system. TD items have both causes and consequences. The cause of technical debt can be a process, a decision, an action (or lack thereof), or an event that triggers the existence of that TD item, such as schedule pressure, unavailability of a key person, or lack of information about a technical feature.



■ **Figure 1** Conceptual Model for Technical Debt.

The consequences of a TD item are many: technical debt can effect the value of the system, the costs of future changes, the schedule, and system quality. The business objectives of the sponsoring organization developing or maintaining the software system are affected in several ways: through delays, loss of quality for some features of the system, and difficulties in maintaining the system operations (continuance).

A TD item is associated with one or more concrete, tangible artifacts of the software development process, primarily the code, but also to some extent the documentation, known defects, and tests associated with the system.

To keep with the financial metaphor, the cost impact of technical debt can be seen as composed of principal and interest. The principal is the cost savings gained by taking some initial approach or shortcut in development (the initial principal, often the initial benefit) or the cost that it would now take to develop a different or better solution (the current principal).

The interest is comprised of costs that add up as time passes. There is recurring interest: additional cost incurred by the project in the presence of technical debt, due to reduced velocity (or productivity), induced defects, and loss of quality (maintainability is affected). And there are accruing interests: the additional cost of the developing new software depending on not-quite-right code (evolvability is affected).

This view summarizing the elements related to technical debt, however, does not capture causes that may or may not be input to measurement and analysis, the activities that need to be conducted to manage technical debt, and the states debt may go through. Another view is intended to articulate these aspects.

This definition and the model serve as the starting point for the community to build on and improve.

Research Road Map

One outcome of the seminar was a broad agenda for future work in technical debt research. While this road map needs to be fleshed out in the future with more detailed research questions and problem statements, it lays out three areas that require attention. First is the identification of a core concept – value – that is central to the technical debt metaphor and that needs definition and operationalization. Second is a recognition that there is an important context to technical debt that should be studied. There are attributes of the context of any particular instance of technical debt in a real environment that must be understood. But there are also other phenomena that are related to technical debt that should be studied, such as other types of “debt.” Third, the road map lays out the community’s basic infrastructure needs, which will enable further collaboration and progress in this area. The research road map that arose out of the discussions at Dagstuhl is described in more detail in this report under Section 4: Working Groups.

Follow-up Work

At the seminar, participants recognized that a carefully considered conceptual model and research road map would be useful outputs for the broader community interested in managing technical debt. Hence, more comprehensive explanation of a conceptual model and the research road map are planned as publications in appropriate venues once the community has a chance to vet the ideas further. The blog established before the seminar will continue to facilitate this interaction.

2 Table of Contents

Executive Summary

<i>Ipek Ozkaya, Philippe Kruchten, Robert Nord, Paris Avgeriou, and Carolyn Seaman</i>	110
--	-----

Overview of Talks

Technical Debt: Financial Aspects <i>Areti Ampatzoglou</i>	117
Towards a New Technical Debt Index: A Code and Architecture-Driven Index <i>Francesca Arcelli Fontana and Riccardo Roveda Marco Zanoni</i>	117
Dynamic and Adaptive Management of Technical Debt: Managing Technical Debt @Runtime <i>Rami Bahsoon</i>	118
Towards Measuring the Defect Debt and Building a Recommender System for Their Prioritization <i>Ayse Basar Bener</i>	119
Technical Debt Management in Practice <i>Frank Buschmann</i>	119
Relative Estimates of Technical Debt <i>Alexandros Chatzigeorgiou</i>	120
Measuring and Communicating the Technical Debt Metaphor in Industry <i>Bill Curtis</i>	120
On the Interplay of Technical Debt and Legacy <i>Johannes Holvitie and Ville Leppänen</i>	122
Technical Debt Aware Modeling <i>Clemente Izurieta</i>	122
Business Value of Technical Debt <i>Heiko Koziolok and Klaus Schmid</i>	123
Prioritization of Technical Debt <i>Antonio Martini and Jan Bosch</i>	123
Technical Debt in Scientific Research Software <i>John D. McGregor</i>	124
Google Experience Report Engineering Tradeoffs and Technical Debt <i>J. David Morgenthaler</i>	124
Technical Debt in Product Lines <i>Klaus Schmid</i>	125
On Concept Maps for TD Research <i>Carolyn Seaman</i>	126
Technical Debt Concepts in Architectural Assessment <i>Andriy Shapochka</i>	126
An Approach to Technical Debt and Challenges in the Acquisition Context <i>Forrest Shull</i>	127

From Technical to Social Debt and Back Again <i>Damian Andrew Tamburri and Philippe Kruchten</i>	128
Technical Debt Awareness <i>Graziela Tonin, Alfredo Goldman, and Carolyn Seaman</i>	128
Working Groups	
A Research Road Map for Technical Debt <i>Carolyn Seaman</i>	129
Open Problems	
The Interplay Between Architectural (or Model Driven) Technical Debt vs. Code (or Implementation) Technical Debt <i>Clemente Izurieta</i>	132
From Technical Debt to Principal and Interest <i>Andreas Jedlitschka, Liliana Guzmán, and Adam Trendowicz</i>	133
Community Datasets and Benchmarks <i>Heiko Koziol and Mehdi Mirakhorli</i>	133
Deprecation <i>J. David Morgenthaler</i>	134
Report of the Open Space Group on Technical Debt in Product Lines <i>Klaus Schmid, Andreas Jedlitschka, John D. McGregor, and Carolyn Seaman</i> . . .	134
Automating Technical Debt Removal <i>Will Snipes and Andreas Jedlitschka</i>	135
Social Debt in Software Engineering: Towards a Crisper Definition <i>Damian Andrew Tamburri, Bill Curtis, Steven D. Fraser, Alfredo Goldman, Jo- hannes Holvitie, Fabio Queda Bueno da Silva, and Will Snipes</i>	136
An Advanced Perspective for Engineering and Evolving Contemporary Software <i>Guilherme Horta Travassos</i>	136
Participants	138

3 Overview of Talks

3.1 Technical Debt: Financial Aspects

Areti Ampatzoglou (University of Groningen, NL)

License © Creative Commons BY 3.0 Unported license
© Areti Ampatzoglou

Joint work of Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Paris Avgeriou

Main reference A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, “The financial aspect of managing technical debt: A systematic literature review”, *Information & Software Technology*, Vol. 64, pp. 52–73, 2015.

URL <http://dx.doi.org/10.1016/j.infsof.2015.04.001>

The concept of technical debt is closely related to the financial domain, not only due to the metaphor that bonds it with financial debt, but also because technical debt represents money. On the one hand, it represents money saved while developing at a lower quality or money earned when delivering the product in time, whereas on the other hand, it represents money spent when applying a refactoring. As a result, financial terms are broadly used in TD literature. In order to work towards a framework for managing technical debt, we have attempted to organize a glossary of the most common financial terms that are used in the state of the art. The glossary presents these terms and a definition for each one. The definitions are a result of synthesizing the way that the terms are used in literature and in some cases they reflect our understanding of how these notions could prove beneficial for technical debt management. Additionally, we have illustrate our view on how financial terms are used in technical debt literature and the way they are linked to each other.

3.2 Towards a New Technical Debt Index: A Code and Architecture-Driven Index

Francesca Arcelli Fontana (University of Milano-Bicocca, IT) and Riccardo Roveda Marco Zanoni

License © Creative Commons BY 3.0 Unported license
© Francesca Arcelli Fontana and Riccardo Roveda Marco Zanoni

Main reference F. Arcelli Fontana, V. Ferme, M. Zanoni, R. Roveda, “Towards a Prioritization of Code Debt: A Code Smell Intensity Index”, in *Proc. of 7th IEEE International Workshop on Managing Technical Debt (MTD@ICSME’15)*, pp. 16–24, IEEE CS, 2015.

URL <http://dx.doi.org/10.1109/MTD.2015.7332620>

URL <http://essere.disco.unimib.it/>

In our laboratory of Evolution of Software SystEms and Reverse Engineering (ESSeRE Lab) of University of Milano Bicocca, we have experimented five different tools able to provide a Technical Debt Index, sometime called in different ways, but with the same or similar purpose. We found that, often, architectural issues are not taken into account, and when they are considered the main focus is on the detection of cyclic dependencies or other dependency issues. Many other architectural problems/smells are not considered, e.g., the relations (structural or statistical) existing among code and architectural problems. Moreover, different architectural smells/problems can be identified only by analyzing the development history of a system, and TD indexes do not take into account this kind of information too. We would like to work on the definition of a new TD index, with a focus on code and architectural debt, and experiment it on a large dataset of projects. In the TD index computation we would like to consider:


- Code and architectural smells detection
- Code and architecture/design metrics

- History of a system, including code changes, lifespan of the code and architectural smells
- Identification of problems more critical than others, to weight the collected analysis elements (e.g., metrics, smells, issues) according to their relevance in existing (past) projects

To this aim, we are working with colleagues of other two Universities on the definition of a catalogue of architectural smells (AS) and their classification. The classification could be useful to better explore possible relations existing among the architectural smells. We are currently working on the identification of some architectural smells by exploiting different metrics, the history of a system and their possible correlations (structural and/or statistical). The choice of thresholds considered for the applied metrics is a critical problem and can be determined through statistical analysis and/or regression (machine learning). Moreover, we have to identify the most critical problems/AS in order to prioritize their removal. The information on the code parts of a system that have been subjected to more changes in their history and, that we expect that will be more changed or extended in the future, can be used for the prioritization of the problems to be removed, together with the information on the code parts with low quality in terms of metric values and code/architectural smells.

3.3 Dynamic and Adaptive Management of Technical Debt: Managing Technical Debt @Runtime

Rami Bahsoon (University of Birmingham, GB)

License  Creative Commons BY 3.0 Unported license
© Rami Bahsoon

The talk has highlighted ongoing effort on managing technical debt in open, dynamic and adaptive environments, where we have looked at dynamic and adaptive composition of cloud-based architecture as an example. We have motivated the need for treating technical debt as a “moving target” that needs to be dynamically and adaptively monitored for prevention and/or transforming the debt into value. We have argued that the much of the debts can be linked to ill- and poorly- justified runtime decisions that can carry short-term gains but not geared for long-term benefits and future value creation. The debt can be observed on utilities linked to Quality of Services (QoS), Service Level violations, need for excessive and costly adaptation etc. The talk has highlighted examples of these decisions and has looked at two interconnected angles for managing debt at runtime: (i) predicative and preventative design support for debt-aware dynamic and adaptive systems and (ii) using online and adaptive learning as mechanisms for proactive runtime management of debts. We have revisited the conceptual technical debt model of Kruchten et al. to make runtime debt concerns, items, artefacts, consequences, etc. explicit. We have seen a need for enriching the model with time-relevant information to address pragmatic needs for runtime management of debt.

3.4 Towards Measuring the Defect Debt and Building a Recommender System for Their Prioritization

Ayşe Basar Bener (Ryerson University – Toronto, CA)

License © Creative Commons BY 3.0 Unported license
 © Ayşe Basar Bener
Joint work of Shirin Akbarinasaji
Main reference S. Akbarinasaji, A. B. Bener, A. Erdem, “Measuring the Principal of Defect Debt”, in Proc. of the 5th Int’l Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE@ICSE’16), pp. 1–7, ACM, 2016.
URL <http://dx.doi.org/10.1145/2896995.2896999>
URL <http://www.ryerson.ca/~abener/pubs.html>

Due to tight scheduling and limited budget, the software development may not be able to resolve all the existing bugs in a current release. Similar to the concept of technical debt, there are also defects that may be postponed to be fixed in the upcoming releases. Such lingering defects are left in the system intentionally or unintentionally, but they themselves create debt in the system similar to the technical debt metaphor. In this research, we particularly focus on defect debt. The accumulation of the deferred bugs in the issue tracking system leads to the rise of the defect debt. In order to manage the defect debt, software managers require to be aware of amount of debt (principal) and interest in their system. There are several studies in the literature which measure the principal, however, only few researchers investigate the interest amount. In this study, we propose two novel approaches to calculate the interest amount for defect debt based on the severity, priority of defect and graph theory analysis. We then propose a dynamic model such as reinforcement learning that learns dynamically from its environment in order build a recommender system to prioritize defects based on the debt (principal and interest).

3.5 Technical Debt Management in Practice

Frank Buschmann (Siemens AG – München, DE)

License © Creative Commons BY 3.0 Unported license
 © Frank Buschmann

In real-world development Technical Debt is always present. Technical Debt Management is thus a continuous activity; almost on a day-to-day basis. This puts the following four requirements on any Technical Debt management environment: Technical Debt identification and assessment must be performed automatically. Appropriate tools must retrieve and handle information from multiple sources, such as code, backlogs, and architecture documentation. A defined quality model is necessary to assess whether or not Technical Debt has actually occurred or is beyond a defined threshold. Identified Technical Debt must be put automatically into the context of development to decide where in the system it is of value to manage it. For instance, Technical Debt in code to be extended for the next product release is more critical than Technical Debt in system areas that stay untouched. Technical Debt to manage must be prioritized according to concrete release goals. For instance, Technical Debt that causes rippling effects to other modules it is likely of higher priority than Technical Debt that stays internal to a module. Measures to address Technical Debt must balance effort and value. Features are a system’s asset, code is its liability. In the end it is important to deliver a competitive system, not a debt-free system. It is also important to minimize the root causes of Technical Debt – regardless of whether it is taken consciously or accidentally Development

processes should be freed from practices that put technical challenges on development teams or invites them to game with the process. Architecture styles like Microservices reduce the occurrence and even more the outreach of Technical Debt in a system. Technical trainings for teams on deliberate design and coding practices can substantially limit the occurrence of Technical Debt.

3.6 Relative Estimates of Technical Debt

Alexandros Chatzigeorgiou (University of Macedonia – Thessaloniki, GR)

License © Creative Commons BY 3.0 Unported license

© Alexandros Chatzigeorgiou

Joint work of Ampatzoglou, Areti; Ampatzoglou, Apostolos; Avgeriou, Paris; Amanatidis, Theodoros
Main reference A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, “The Financial Aspect of Managing Technical Debt: A Systematic Literature Review”, *Information and Software Technology*, Vol. 64, pp. 52–73, 2015.
URL <http://dx.doi.org/10.1016/j.infsof.2015.04.001>

Industry and academia agree that technical debt has to be managed and assessed. However, the TD community lacks commonly agreed methods/tools for TD estimation. Almost all current approaches for TD measurement are based on the identification of individual inefficiencies, usually at the design and code level. However, developers are reluctant to accept as a panacea any approach that sets arbitrary thresholds for the desired quality. To address this practical challenge we argue that design/code inefficiencies should not be assessed against ‘hard’ thresholds but on the basis of relative measurements. We believe that the theory and practice of search-based software engineering (SBSE) can be exploited to extract a design that optimizes a selected fitness function. The ‘distance’ between an actual design and the corresponding optimum one can serve as an estimate of the effort that has to be spent to repay TD. Although the notion of an ‘optimum’ system might sound utopic, the benefit from the use of a relative estimate is twofold: a) it expresses a potentially achievable level of TD, b) the aforementioned distance can be mapped to actual refactoring activities. A side benefit of such a quantification approach is that it allows the assessment of individual developer contribution to TD. Through the analysis of software repositories we can assess the impact of individual commits. This information could be exploited to compile individual TD reports for open-source contributors and facilitate the collection of information by providing an additional motivation for participating in surveys related to TD.

3.7 Measuring and Communicating the Technical Debt Metaphor in Industry

Bill Curtis (CAST – Fort Worth, US)

License © Creative Commons BY 3.0 Unported license

© Bill Curtis

Joint work of Douziech, Philippe-Emmanuel; Curtis, Bill

Managers and executives across industry have embraced the technical debt metaphor because it describes software issues in a language that industry understands. However, their tacit understanding of the metaphor differs from the typical formulation in the technical debt research community. Whereas researchers often limit this metaphor to suboptimal design choices that should eventually be corrected, most in industry think of technical debt as the

collection of software flaws that need to be corrected regardless of their type. Industry wants the principal of a technical debt to estimate their corrective maintenance expense and total cost of ownership. Thus, industry is using the technical debt metaphor to describe phenomena it needs to quantify in financial terms. Industry does not care whether a structural flaw fits Cunningham's original concept of technical debt. Rather it cares that IT must spend money to correct these flaws, and that there may be a marginal cost in inefficient use of human and machine resources (i.e., interest) until they are fixed.

Going forward there will probably be at least two divergent perspectives on technical debt, each proper and valuable for its intended use. The research community will focus on sub-optimal design decisions that primarily affect the maintenance and evolvability of software. The industrial community will take a broader perspective on the flaws categorized as technical debt in order to use the metaphor to communicate the cost of software quality problems to the business in terms the business understands. These divergent views can co-exist because they serve different purposes and audiences. If technical debt were limited to architectural smells, industry would eventually turn to other concepts for explaining its broader issues in financial terms.

Measuring technical debt as an estimator of corrective maintenance costs implies measuring software flaws that must be fixed. If a flaw is not sufficiently damaging that its correction can be deferred endlessly, it is not an item of technical debt because the organization does not plan to spend money correcting it. The Consortium for IT Software Quality (CISQ) has defined measures of structural quality related to Reliability, Security, Performance Efficiency, and Maintainability based on counting structural flaws at the architectural and code unit level that can be detected through static analysis. The criterion for including a structural flaw in calculating these measures was that a panel of industry experts had to consider it severe enough to require correction. For instance, the CISQ Security measure was constructed from the Top 25 CWEs (Common Weakness Enumeration, the basis for the SANS Top 25 and OWASP Top 10) that hackers exploit to gain unauthorized entrance into a system.

The CISQ measures not only provide estimates of corrective maintenance costs, they also provide indicators related to the risk that systems can experience outages, data corruption, performance degradation, and other operational problems. The CISQ measure of technical debt will aggregate the structural flaws in the four quality characteristic measures and apply an estimate of corrective effort for each. When aggregated across the four CISQ measures, this measure should provide a good estimator of corrective maintenance costs as well as operational risks. The measure can be modified as empirical results from industry indicate improvement opportunities. Industry's willingness to participate in empirical validation studies is crucial for validating and improving the explanatory power of the technical debt metaphor.

3.8 On the Interplay of Technical Debt and Legacy

Johannes Holvitie (University of Turku, FI) and Ville Leppänen

License © Creative Commons BY 3.0 Unported license
© Johannes Holvitie and Ville Leppänen

Joint work of Holvitie, Johannes; Hyrynsalmi, Sami; Leppänen, Ville

Main reference J. Holvitie, V. Leppänen, S. Hyrynsalmi, “Technical Debt and the Effect of Agile Software Development Practices on It – An Industry Practitioner Survey”, in Proc. of the 6th Int’l Workshop on Managing Technical Debt (MTD@ICSME’14), pp.35–42, IEEE CS, 2014.

URL <http://dx.doi.org/10.1109/MTD.2014.8>

Different methods for technical debt accumulation have been discussed, but they have mainly focused on immediate accumulation. Arguably, there is also delayed accumulation, wherein the environment around static software assets changes. Here, updates to the assets no longer deliver the environments assumptions to the assets, they become detached from the environment, and we find the assets to have accumulated debt. This debt closely reminds of software legacy, as it represents assets which can no longer be subjected under the same development actions as newly created ones. In a recent multi-national survey, it was discovered that over 75% of technical debt instances’ have perceived origins in software legacy. This communicates about the close relation between the legacy and the debt concepts. While it encourages us to further explore applying established legacy management methods for technical debt, the relation also exposes challenges. Notably, we should consider if legacy is being merely rebranded into the more favorable technical debt. And if this is the case, how do we ensure that all aspects of the legacy instances are identified so as to convert them into fully manageable technical debt assets. Failure to do so, will result into having technical debt assets with varying levels of accuracy, and this undoubtedly hinders technical debt management efforts overall. Nevertheless, while considering these challenges, the software legacy domain should be further researched as both a commonality of technical debt instances and as a possible interface for enhancing existing management approaches.

3.9 Technical Debt Aware Modeling

Clemente Izurieta (Montana State University – Bozeman, US)

License © Creative Commons BY 3.0 Unported license
© Clemente Izurieta

Joint work of Rojas, Gonzalo

Main reference I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, and B. Williams, “The Correspondence Between Software Quality Models and Technical Debt Estimation Approaches”, in Proc. of the 6th International Workshop on Managing Technical Debt (MTD@ICSME’14), pp. 19–26, IEEE CS, 2014.

URL <http://dx.doi.org/10.1109/MTD.2014.13>

The Software Engineering Laboratories (SEL) at Montana State University has been engaged in active Technical Debt research for approximately five years. Our research and development goal is to find a balance between practical applications of research findings in the form of very useful tools tailored for commercial customers. We are currently developing dashboard technology that builds on the SonarQube framework infrastructure to provide functionality that calculates the quality of software according to various ISO standards, as well as providing technical debt measurements. The framework is tailored to be extensible by allowing for additional plug-ins. For example, we are currently focusing on building the Risk Management Framework quality model (RMF). SEL is also focusing on researching the various modeling decisions that affect the technical debt of associated code generated from such models.

This research will lead to a plug in that is tailored for modelers and architects, but that is technical debt aware by providing different model smell refactoring alternatives that a modeler may choose from. Each choice has decisively different outcomes in the technical debt measurements performed in the corresponding generated code. This line of research will lead to better traceability and to the understanding of the relationship between architecture and code.

3.10 Business Value of Technical Debt

Heiko Kozirolek (ABB AG Forschungszentrum – Ladenburg, DE) and Klaus Schmid (Universität Hildesheim, DE)

License © Creative Commons BY 3.0 Unported license
© Heiko Kozirolek and Klaus Schmid

Technical debt prioritization is challenging, because the benefit of resolving technical debt (TD) items varies depending on the planned evolution of a software system. Fixing TD items in code parts that will not be modified provides no immediate benefits. The decision on how many TD items to resolve depends on the business context a software system is developed in. In an innovative, rapidly growing market, time-to-market delivery may be essential for product success, so taking on technical debt in these situations may be warranted. These observations call for more emphasis on modeling the business context of software systems, capturing the view of product managers and decision makers. However, most TD analysis and resolution methods and tools are developed from the perspective of the software engineers and architects, not explicitly accounting for market analysis or future evolution scenarios. Therefore it appears useful to develop method and tools in collaboration with both developer and decision makers. We envision an Integrated Technical Debt Analysis Environment that can integrate both the output of software artifact analysis tools as well as the output of modelling the business context and development road maps. Such an instrument would allow to make an informed decision about taking or resolving technical debt respecting both the developer and management perspective.

3.11 Prioritization of Technical Debt

Antonio Martini (Chalmers UT – Göteborg, SE) and Jan Bosch

License © Creative Commons BY 3.0 Unported license
© Antonio Martini and Jan Bosch
Joint work of Antonio Martini, Jan Bosch
Main reference A. Martini, J. Bosch, “An Empirically Developed Method to Aid Decisions on Architectural Technical Debt Refactoring”, in Proc. of the 38th Int’l Conf. on Software Engineering (ICSE’16) – Companion Volume, pp. 31–40, ACM, 2016.
URL <http://dx.doi.org/10.1145/2889160.2889224>
Main reference A. Martini and J. Bosch, “Towards Prioritizing Architecture Technical Debt: Information Needs of Architects and Product Owners”, in Proc. of the 41st Euromicro Conf. on Software Engineering and Advanced Applications (EUROMICRO-SEAA’15), pp. 422–429, IEEE CS, 2015.
URL <http://dx.doi.org/10.1109/SEAA.2015.78>

A Technical Debt item needs to be prioritized against features and among other TD items. There is a need for mechanisms, methods and tools to aid the stakeholders in prioritizing the refactoring (repayment) of Technical Debt. An important step is to understand the Technical Debt impact (interest). We found that the estimated impact of TD items provides useful

information when the stakeholders (technical and non-technical) prioritize with respect to aspects such as Lead Time, Maintenance Cost and Risk. However, there are aspects that are considered of higher priority in commercial software organizations, such as Competitive Advantage, Specific Customer Value and Market Attractiveness. It is important to understand if and how the impact of Technical Debt affects such aspects (directly or indirectly). In order to understand if and when a Technical Debt item should be refactored with respect to other items, we proposed an approach based on the calculation of the ratio Principal/Interest. Such an approach (AnaConDebt) is based on a checklist of key factors that compose Principal and Interest. The assignment of weights to such factors, based either on expert experience or on metrics available at the organizations, gives a result that is simple to interpret and useful for comparison. The comparison of the ratio with other TD items' ratios helps the prioritization. Multiple ratios can also be calculated at different points in time for the same item, to estimate if the refactoring can be postponed or not. This approach can be repeated iteratively to adjust and reprioritize the refactoring according to new information available to the stakeholders.

3.12 Technical Debt in Scientific Research Software

John D. McGregor (Clemson University, US)

License  Creative Commons BY 3.0 Unported license
© John D. McGregor

Developing scientific research software is expanding rapidly as research methods are more software-based. The process for developing that software is a breeding ground for technical debt. In most research groups the students of physics, chemistry or other discipline are not trained in software development and the managers of those students are professors with even less software experience than their younger students. The problems with these systems can be considered technical debt because the work of that research group depends on building on the software built by previous students. Research progress is slowed if that existing software must be modified before new work can be started. One research study recently was delayed due to memory limitations. They ported to a larger system only to find there were inherent assumptions throughout the software that limited the ability to take advantage of the larger system. Our work on scientific software ecosystems is investigating how to establish and maintain a supply chain of software which allows students to select software that will be suitable for their work.

3.13 Google Experience Report Engineering Tradeoffs and Technical Debt

J. David Morgenthaler (Google Inc. – Mountain View, US)

License  Creative Commons BY 3.0 Unported license
© J. David Morgenthaler

Software development at Google sits at one extreme on the spectrum of development environments, and as such may not represent the challenges typically seen by the vast majority of other companies, developers, or projects. Yet I believe the issues Google faces with rapid technical evolution of our heavily reused internal components point to the future

direction the industry as a whole is heading. Google's underlying infrastructure, whether hardware or software, is changing so quickly that developers often face a new type of tradeoff – between platforms, frameworks and libraries that are stable but superseded, and those that are new and state-of-the-art, but currently incomplete. The up-front design decision is not whether to take on technical debt, but which form. Either speedy development using tried and true, but soon to be unsupported, platforms, or slower, more painful, work as a guinea pig for the next cool development approach with its promise of a longer life span.

Google also imposes these debt decisions on external developers who use open-sourced Google platforms. The Android OS is one well-documented example, with major revisions shipping nearly every year. This fast pace of innovation also leads to version fragmentation and rapid software aging, and therefore higher maintenance costs. The road ahead is indeed fraught with danger. Yet with an 'installed base' of 1.5 billion active devices and 2 million available apps, this ecosystem represents a tremendous opportunity, both for developers to reach billions of future users, and for future technical debt research.

3.14 Technical Debt in Product Lines

Klaus Schmid (Universität Hildesheim, DE)

License © Creative Commons BY 3.0 Unported license
© Klaus Schmid

Joint work of Sascha El-Sharkawy, Adam Krafczyk

Main reference S. El-Sharkawy, A. Krafczyk, and K. Schmid, "Analysing the Kconfig Semantics and Its Analysis Tools", in Proc. of the 2015 ACM SIGPLAN Int'l Conf. on Generative Programming: Concepts and Experiences (GPCE'15), pp. 45–54, ACM, 2015.

URL <http://dx.doi.org/10.1145/2814204.2814222>

Traditionally technical debt (TD) research focused particular on individual systems and the research on managing TD aimed at supporting a single project. However, it should be recognized that also product lines may contain (significant) forms of TD. – And what is more, this TD may come in new and different forms that are not yet addressed by existing tools. In our research we are currently focusing on technical debt in software product lines. Ideally, product lines consist of a variability model along with corresponding assets that are managed to create the resulting product. In our research we focus in particular on logical anomalies that complicate the product line realization. Examples of this are dead code, undead code or over-constrained code that reduces the potential number of code configurations below the range of configurations described by the variability model. This is of course not the only way to address the problem of smells – similar to metrics in traditional technical debt research one could also search for code smells using metrics. A complexity that arises in particular in product line research is that technical debt related to configurations cannot be addressed without taking also into account the build space. Hence, we need to integrate information from variability model, code variability, and build information to arrive at reasonable conclusions regarding product technical debt.

3.15 On Concept Maps for TD Research

Carolyn Seaman (University of Maryland, Baltimore County, US)

License  Creative Commons BY 3.0 Unported license
© Carolyn Seaman

From my point of view, the motivation for developing a concept map for the TD research community is two-fold:

- To standardize terminology.
- To aid in categorization of new and existing research.

However, there are other relevant motivations for concept mapping. I presented in this talk three concept maps that I have found in the TD literature, that are very different and have different motivations, in an effort to convey the breadth of ideas we should consider when developing a concept map for our community.

The first concept map I presented was a very old, very simple framework that I have used to organize work in my own lab. It divides TD research into TD identification approaches, TD measurement and analysis approaches, approaches to making decisions using TD information and approaches to organizing and storing information about TD. While I have found the distinction between these areas of TD research, I think the community could use a more comprehensive and sophisticated model.

The second model I presented was derived from an in-depth historical case study of the events and decisions related to a single instance of TD. The goal was a grounded descriptive model of TD decision making. It's described in detail in a paper currently under review (Siebra, et al.).

The third model I talked about was based on an in-depth analysis of early (pre-2011), mostly non-scholarly, TD literature (Tom et al., 2013). The resulting map provides a useful categorization of TD precedents, TD outcomes, TD attributes, and software development dimensions (e.g. development phases).

3.16 Technical Debt Concepts in Architectural Assessment

Andriy Shapochka (SoftServe – Lviv, UA)

License  Creative Commons BY 3.0 Unported license
© Andriy Shapochka

An architectural assessment has become one of the most important tools in the architect's toolbox. Built on a mature methodologies such as ATAM and CBAM it serves well to evaluate software architectures and their implementations against the set of high priority quality attributes, constraints, and other architectural drivers relevant to the assessed system.

The assessment process is well defined, timeboxed to a few weeks and can be applied to the system in any phase of the software development lifecycle. It combines both qualitative and quantitative analysis techniques leading to a set of architecture improvement recommendations.

Architectural assessments are exceptionally well suited to evaluate maintainability and evolvability of the system at hand which essentially means its technical debt analysis including search for and categorization of potential technical debt items on the levels from overall system architecture to component design, to implementation details, their prioritization, value

for cost analysis of technical debt elimination, improvement metrics setup and monitoring, and other important activities.

As technical debt can manifest itself on different levels of abstraction and in other system perspectives such runtime quality attributes (performance, reliability, security, etc.) affected by the technical debt related tradeoffs, testability influenced by the complexity of the system implementation, metrics related to defects (frequencies, time to fix, defect density in components, etc.) proper analysis for the technical debt often involves comprehensive evaluation of the entire system architecture, its history, and plans for evolution.

There are multiple challenges still in need of addressing in the context of the technical debt assessments: quick and efficient semi-automated localization of the technical debt items, consistent value for cost analysis methodology meaningful and understood by business, technical debt interpretation and prioritization correlated with the business needs.

3.17 An Approach to Technical Debt and Challenges in the Acquisition Context

Forrest Shull (Carnegie Mellon University – Pittsburgh, US)

License  Creative Commons BY 3.0 Unported license
© Forrest Shull

This talk demonstrated a metrics-driven approach to TD management which has been used effectively with development organizations. The approach elicits quality goals and rules that the team feels will help achieve those goals, and for which compliance could be detected in the codebase. In an example case study, the quality goal was to minimize maintenance costs and the associated rule was to follow a reference architecture. Tools were used to search for rule violations (i.e. inconsistencies with the reference architecture) through all prior commits in the codebase, and to associate data from the change and defect tracker. Thus, the team could understand how often those rules were broken in the past and whether TD symptoms (e.g. increased change cost, decreased velocity) were correlated with noncompliance. If not, then the rules should be refined or replaced by new rules with more impact. One result from this work is the indication that no one set of TD detection rules consistently correlates with important TD symptoms across different projects and quality goals. Understanding TD in software that is being acquired / purchased presents unique challenges that are not amenable to such methods, however. In such instances the acquiring organization needs to understand the amount of extant technical debt, to estimate future sustainment costs: Will a substantial part of every dollar devoted to new functionality instead go to dealing with accumulated debt? Yet, for acquired software, many of the necessary data sources are not available for analysis. The talk ended by discussing the approaches relevant for acquisition, which need to include manual analysis. For further reading:

References

- 1 Nico Zazworka, Victor Basili, and Forrest Shull. *Tool Supported Detection and Judgment of Nonconformance in Process Execution*. 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM), 2009
- 2 Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn B. Seaman, Michele A. Shaw. *Building Empirical Support for Automated Code Smell Detection*. ESEM 2010

- 3 Nico Zazworka, Michele A. Shaw, Forrest Shull, Carolyn Seaman. *Investigating the Impact of Design Debt on Software Quality*. MTD2011: 2nd Workshop on Managing Technical Debt,” ICSE, 2011, Honolulu, Hawaii, USA

3.18 From Technical to Social Debt and Back Again

Damian Andrew Tamburri (Polytechnic University of Milan, IT) and Philippe Kruchten (University of British Columbia – Vancouver, CA)

License © Creative Commons BY 3.0 Unported license

© Damian Andrew Tamburri and Philippe Kruchten

Main reference D. A. Tamburri, P. Kruchten, P. Lago, H. van Vliet, “Social debt in software engineering: insights from industry”, *Journal of Internet Services and Applications*, 17 pages, Springer, 2015; available open access.

URL <http://dx.doi.org/10.1186/s13174-015-0024-6>

An established body of knowledge discusses the importance of technical debt for product quality. In layman’s terms, Technical debt represents the current state of a software product as a result of accumulated technical decisions (e.g., architecture decisions, etc.). Techniques to study product quality limitations inherent to technical debt include technical data mining, code-analysis.

Nevertheless, quite recently we figured out that technical debt is only one face of the coin. In fact, social debt, the social and organisational counterpart to technical debt, plays a pivotal role in determining software product and process quality as much as technical debt helps phrase the additional technical project cost. Social debt reflects sub-optimal socio-technical decisions (e.g., adopting agile methods or outsourcing) which can compromise the quality of software development communities, eventually leading to software failure.

Studying and harnessing social debt is paramount to ensure successful software engineering in large-scale software development communities. I focus on explaining the definition and relations between technical and social debt while delineating techniques inherited from technical debt, organisational research and social networks research that could be rephrased to investigate social debt in software engineering (e.g., socio-technical code analysis, social-code graphs, socio-technical debt patterns, etc.).

3.19 Technical Debt Awareness

Graziela Tonin (University of Sao Paulo, BR), Alfredo Goldman (University of Sao Paulo, BR), and Carolyn Seaman (University of Maryland, Baltimore County, US)

License © Creative Commons BY 3.0 Unported license

© Graziela Tonin, Alfredo Goldman, and Carolyn Seaman

We conducted a study with teams of students (60 people total) on technical debt awareness. We asked the participants about the effects of explicitly identifying and tracking technical debt on their projects. Awareness of potential technical debt items sometimes led all team members to avoid contracting the debt. They often discussed such decisions, thus improving the team communication. As a result, several team members were more comfortable sharing their difficulties. Consequently, some teams had fewer ‘untouchable’ experts, and thus worked better as a real team. Decisions made to contract a debt were explicit and strategic, e.g. to deliver a new version to the customer faster. Moreover, making the technical debt list

visible allowed them to negotiate more time with the customer to perform refactoring. The technical debt list was used as a historical memory of the immature parts of the project by all teams. They could use it to check if there was debt to be paid in parts of the project that were about to be modified. The list was also a good indicator of the health of the project; it showed if code quality was improving or not. The teams created a culture of continuous improvement. In summary, making technical debt explicit had direct implications on the team's behavior. The majority of the team members said that, after they became aware of technical debt, they communicated more with other team members, thought more about the real need to contract debt, discussed in more deep about code quality, refactored more frequently, and understood the problems in the project better.

4 Working Groups

4.1 A Research Road Map for Technical Debt

Carolyn Seaman (University of Maryland, Baltimore County, US)

License © Creative Commons BY 3.0 Unported license
© Carolyn Seaman

Main reference N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. L. Nord, I. Ozkaya, R. S. Sangwan, C. B. Seaman, K. J. Sullivan, N. Zazworka, "Managing Technical Debt in Software-Reliant Systems", in Proc. of the FSE/SDP workshop on Future of Software Engineering Research, pp. 47–52, ACM, 2010.

URL <http://dx.doi.org/10.1145/1882362.1882373>

Main reference P. Avgeriou, P. Kruchten, R. L. Nord, I. Ozkaya, C. B. Seaman, "Reducing Friction in Software Development", IEEE Software, Vol. 33(1), pp. 66–73, 2016.

URL <http://dx.doi.org/10.1109/MS.2016.13>

At the end of a week of discussions about many aspects of technical debt (TD) research – past, ongoing, and envisioned – the attendees of the Dagstuhl workshop spent a morning sharing ideas about the most important TD-related research problems for the community to work on. The discussion of a research road map was grounded in the definition of TD formulated during the seminar:

In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.

The group spent some time envisioning how the world would be different if all our research efforts in this area were successful. This vision included the following points:

- TD would be managed as carefully as we currently manage defects and new features.
- We would have a clear, operational definition of “good-enough” software, including how much TD is acceptable.
- We would have a way to translate between developer concerns and manager concerns, and this translation would form a basis for making decisions about allocating effort to various tasks.
- TD would be incurred intentionally most of the time.
- Projects that manage TD would be more efficient, effective, and sustainable than projects that don't.

- The notion that up-front architectural work (vs. emergent architecture) is worth it would be well supported and accepted.
- There would be tools to support all aspects of TD management that are adopted and used by all stakeholders.
- TD-aware development (practices and tools) would be an accepted standard way of producing software.
- Architectural assessment would be part of standard development policy.

Achieving this vision will require changing development practices through effective communication between research and practice and, most importantly for the research community, showing the effectiveness of proposed approaches. The research community must also recognize that our practitioner audience consists of two different groups: managers who make decisions about spending money on eliminating TD and developers who write and maintain code. Proposed solutions must ultimately appeal to both, but researchers should also be clear about which group is primarily targeted with a new approach or tool.

The group discussed research activities in three broad areas, which are outlined in the sections below:

1. the core: defining, understanding, and operationalizing the concept of value with respect to TD
2. the essential context: understanding phenomena that fall outside the core definition of TD and that have an essential relationship with how TD plays out in practice
3. the necessary infrastructure: building the shared infrastructure that facilitates all our research activities

The Core: Understanding Value

The concept of value and what it means with respect to TD came up often and in many contexts throughout the seminar. It is not clear that we all mean the same thing when we talk about value, but it is clear that value is central to delivering effective mechanisms for managing TD in practice. Value comes into play both when deciding whether to incur TD (i.e., the value of a proposed TD item) as well as when deciding whether, when, and which TD to pay off (i.e., the value of eliminating an existing TD item). At one level, the concepts of principal and interest capture the short-term benefits (principal) and long-term costs (interest) of incurring TD, as well as the cost/benefit analysis related to paying it off. But these concepts do not adequately capture all aspects of value.

One aspect of value that is not easily captured by the concepts of principal and interest is the opportunity cost. One common benefit of incurring TD is the ability to take advantage of an opportunity (because resources are freed up) that might not be available at another time. In theory, this benefit is part of the principal of the TD being considered, but principal does not capture the time element (the fact that the opportunity is time-sensitive) nor does it allow for the fact that the resources freed by incurring TD and the resources needed to take advantage of the opportunity might not be comparable.

Another aspect of value that is even harder to make concrete is the value of TD management, and TD-related information, to the quality of decision making. Capturing the “quality” of decisions is difficult to begin with, but somehow demonstrating the benefits of considering TD in management decisions is a key area for TD researchers. A related question is how to define good-enough software. When does software have a level of quality (and a level of TD) that cannot be cost-effectively improved? Helping managers define “good enough” in their

context would be extremely useful, and better defining the value of TD is an important step in this process.

The quantification of value (and other TD properties) has always been an important aspect of TD research and was part of the research agenda set out in the first TD research road map in 2010 (Brown et al., 2010). Efforts to quantify principal and interest, and to somehow combine them into a measure of “value,” have proved difficult, but continued work in this area is crucial to furthering our vision of effective TD management in practice. Useful measures would both be at a level of granularity that is useful to developers in tracking and monitoring activities and be understandable in a business sense.

Steps toward providing effective operationalizations of value include proposing proxies for value, collecting data based on these proxies, designing validation procedures (i.e., study designs) to show that these measures are meaningful and useful, and conducting case studies to carry out these validations. Case studies can also be used to explore existing notions of value in practice, which would ground the choice of proposed metrics. Such exploratory case studies could also be used to discover grassroots approaches to TD management – that is, environments without an explicit strategy to manage TD, but where, out of necessity, techniques have emerged to deal with the most important aspects.

The Essential Context: Understanding Related Phenomena

When the Dagstuhl group agreed on our definition of TD, we also recognized that the definition does not encompass all topics that are important to study. TD exists in a context, and it exists in a variety of forms.

The TD definition limits TD to phenomena closely tied to source code (e.g., design debt, code debt, architecture debt), which leaves out other important types of debt that are at least partially analogous to financial debt, such as social debt, people debt, process debt, and infrastructure debt. These other types of debt are related to TD in various ways. Some are part of the causal chain leading to TD; the impacts of these other types of debt can lead to TD as defined. Others appear to co-occur with TD or create an environment that affects the ability to manage TD in some way.

Other aspects of TD context that need to be studied within the TD research agenda include

- uncertainty: building representations of uncertainty into TD value measures and using that information about uncertainty in decision making
- development and organizational context: studying what aspects of context affect how TD is most effectively managed
- time: better accounting for the passage of time (measured not necessarily in time units but in relevant events, such as code changes) in all aspects of TD management, such as the value of TD over time and the effect of time-based events on value
- dependencies and interactions: between TD items, between TD items and development artifacts
- knowledge management: what TD-related information to capture and disseminate for effective TD decision making
- causal chains: the constellation of “causes” leading up to the creation of TD
- TD maturity: the creation of a maturity model that depicts the different levels at which TD could be managed, or possibly incorporating TD concerns into existing maturity models

The Necessary Infrastructure: Data, Tools, and Other Components

One advantage of forming a coordinated, active research community in a particular area of study is the ability to share resources. The group at Dagstuhl identified a number of resources that members of the community could develop and contribute to the TD research infrastructure to enhance the level of progress of the entire field. Identifying relevant pieces of infrastructure is not possible until a community reaches a certain level of maturity. We feel that the TD research community has reached that level and that identifying and building common infrastructure would now be useful. This common infrastructure should include the following components:

- data sets: OSS projects provide some useful data, but they generally do not include effort data, which is essential to addressing most relevant questions in TD research. “Data” provided with published studies is often really the analysis of the raw data, not the data itself.
- benchmarks: Ground truth in this area is difficult to come by, but we can strive for intersubjectivity, or widespread agreement on an essentially subjective proposition. This can apply to, for example, tools that detect code smells or calculate code-based quality measures. Designating certain tools as “reference” implementations of these functions could also be useful.
- common metrics for outcome variables (maintenance effort, defects, etc.)
- tools: pluggable, validated, benchmarked. Tools must implement an underlying, understandable process, as some contexts will not allow for using a specific tool
- infrastructure for replication as well as for new work, so that it can be usefully compared to existing work

5 Open Problems

5.1 The Interplay Between Architectural (or Model Driven) Technical Debt vs. Code (or Implementation) Technical Debt

Clemente Izurieta (Montana State University – Bozeman, US)

License © Creative Commons BY 3.0 Unported license
© Clemente Izurieta

Joint work of Rojas, Gonzalo; Izurieta, Clemente

Main reference C. Izurieta, G. Rojas, I. Griffith, “Preemptive Management of Model Driven Technical Debt for Improving Software Quality”, in Proc. of the 11th Int’l ACM SIGSOFT Conf. on Quality of Software Architectures (QoSA’15), pp. 31–36, 2015; pre-print available from author’s webpage.

URL <http://dx.doi.org/10.1145/2737182.2737193>

URL <http://www.cs.montana.edu/izurieta/pubs/qosa39s.pdf>

The purpose of this open space session was to discuss the differences between architectural and model driven technical debt vs. code or implementation technical debt. Although terminology is used interchangeably, the community agrees that architectural debt occurs at a higher level of abstraction than implementation. Further, architectural debt encompasses tools that may be textual (i.e., RBML) or visual (i.e., UML, AADL), although the majority tends to be visually oriented. There is however a middle layer that overlaps between code-based implementation technical debt and architectural technical debt and it aggregates various metrics that can be measured at either level of abstraction with differing maintainability (architecture or code) consequences. This is an area that requires further research. Amongst other issues brought up was the technical debt associated with the automatic generation

of code from models. In some cases the code generated may be smelly because changes at the model level (that address a model smell) may generate constructs in the code that are deemed code smells. For example, empty classes and “to-do” tags. We also discussed the issue of traceability, and although a large community exists, its focus is on requirements; whereas traceability in this space refers to the interplay between technical debt observed in the code and its corresponding model or architecture. The latter may also lead to potential cause and effect studies.

5.2 From Technical Debt to Principal and Interest

Andreas Jedlitschka (Fraunhofer IESE – Kaiserslautern, DE), Liliana Guzmán, and Adam Trendowicz

License © Creative Commons BY 3.0 Unported license
© Andreas Jedlitschka, Liliana Guzmán, and Adam Trendowicz

Technical debt leads to interest payments in the form of additional effort that software practitioners need to do in future development of a software system because of quick and dirty design decisions. Deciding whether continuing paying interest or paying down the principal by refactoring the software is one of the most important challenges in managing technical debt. Recent experiences have shown that context-specific checklists to assess the implications of technical debt might be useful to support software practitioners in making this decision. However, software practitioners and researchers agreed on the need for context-specific estimation and prediction models regarding technical debt principal and interest. Furthermore, they claimed building such models involves the analysis of several factors including (1) the business goals and context associated to a software system, (2) the business implications associated to technical debt, (3) business as well as software development, maintenance, and refactoring costs, and (4) the complexity and benefits of the necessary changes.

5.3 Community Datasets and Benchmarks

Heiko Koziolk (ABB AG Forschungszentrum – Ladenburg, DE) and Mehdi Mirakhorli (Rochester Institute of Technology, US)

License © Creative Commons BY 3.0 Unported license
© Heiko Koziolk and Mehdi Mirakhorli

Research in managing technical debt could be improved if researchers shared information about their systems under analysis, datasets they produced, software tools they created and agreed on benchmarks to compare new methods and tools. In an Open Space Discussion, we brainstormed prerequisites and contents for a community repository that could contain these items. First, the technical debt community could appoint certain open source systems as community reference systems, which could be analyzed by different researchers. Besides the source code, also architectural knowledge, information about the development history, evolution scenarios and an issue tracker should be available for these systems, best facilitated by access to the system’s developers. Second, datasets could be shared in a similar fashion as in the PROMISE repository, for example anonymized analysis results from the company CAST or a set of reference technical debt metrics as proposed in context of the OMG’s Consortium

For IT Software Quality (CISQ) Seminar. Third, benchmarks could be established from these prerequisites, similar to the DEEBEE and BEFRIEND benchmarks used for evaluating design pattern detection and reverse engineering tools. A controlled experiment design could be another form of benchmark, allowing to evaluate the benefits of any approach managing technical in terms of quality and time-to-delivery. Finally, software tools, such as parsers, code smell detectors, pattern-detectors, issue tracker miners, etc., could be packaged and prepared for reuse by other researchers.

5.4 Deprecation

J. David Morgenthaler (Google Inc. – Mountain View, US)

License  Creative Commons BY 3.0 Unported license
© J. David Morgenthaler

Deprecation refers to a process for marking methods, classes, APIs, or entire systems whose use is to be discouraged. According to Wikipedia: While a deprecated software feature remains in the software, its use may raise warning messages recommending alternative practices; deprecated status may also indicate the feature will be removed in the future. Features are deprecated rather than immediately removed, to provide backward compatibility and give programmers time to bring affected code into compliance with the new standard.

The entire idea of software deprecation therefore maps directly to technical debt; it can be viewed as software aging made manifest. Deprecated features are the embodiment of an opening technology gap.

At Google, deprecation is often applied to discourage the use of features, but lacking discrete versions of most of our shared components, it is difficult to require users to upgrade their systems. Since everything is built from head, a feature cannot be deleted while existing users and their tests remain. Deprecation in this environment can become a tragedy of the commons. Yet our environment is closed, in that Google engineers can readily determine all users of a given feature. In fact, engineers are encouraged to find and update all uses of features they deprecate. As Google grows, however, this approach does not scale. As software everywhere become more dependent on services provided by the external environment, this type of technical debt will also become more prevalent. What can be done to avoid this onrushing crisis?

5.5 Report of the Open Space Group on Technical Debt in Product Lines

Klaus Schmid (Universität Hildesheim, DE), Andreas Jedlitschka (Fraunhofer IESE – Kaiserslautern, DE), John D. McGregor (Clemson University, US), and Carolyn Seaman (University of Maryland, Baltimore County, US)

License  Creative Commons BY 3.0 Unported license
© Klaus Schmid, Andreas Jedlitschka, John D. McGregor, and Carolyn Seaman

Product lines extend the traditional field of technical debt research as they widen the view towards whole sets of systems. With this wider view additional complexity comes in as relations among the systems may add to the debt. However, the first observation clearly is: all problems and issues of technical debt in single systems are relevant to product lines as well.

While, ideally, a product line is characterized by common reusable assets, often (code) clones may occur. This may happen either because the various products are not fully integrated, or because the development gets separated in an attempt to speed up development. This cloning can be a major source of technical debt. However, even if the product line is integrated and consists of a variability model and configurable assets, the chance of technical debt exists. Actually this situation may give rise to a particular form of technical debt: technical debt may exist due to (unnecessary) complexity created by variability models or variability markups in the code. Various indications may make such variability-related smells visible. These may either be described as metrics (e.g., total number of dependencies relative to features could be very high) or they can be described as logical anomalies (e.g., dependencies that indirectly lead to equivalences).

5.6 Automating Technical Debt Removal

Will Snipes (ABB – Raleigh, US) and Andreas Jedlitschka (Fraunhofer IESE – Kaiserslautern, DE)

License © Creative Commons BY 3.0 Unported license
© Will Snipes and Andreas Jedlitschka

Main reference N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou. “JDeodorant: Identification and Removal of Type-Checking Bad Smells”, in Proc. of the 12th European Conf. on Software Maintenance and Reengineering (CSMR’08), pp. 329–331, IEEE CS, 2008.

URL <http://dx.doi.org/10.1109/CSMR.2008.4493342>

Organizations with legacy software may have technical debt in that software that they carry forward with each release. Although this debt can be identified, the manual process to identify and in particular remove debt items poses both challenges of economics and risk. Improvements are typically recommended only when changing the code for new features, because the changes pose risk to the system that must be mitigated with additional testing. One way to mitigate the risk and improve the effort required is to provide automated tool support for addressing technical debt items. Common tools that support automated refactoring cover only a few simple refactoring patterns that do not address more elaborate technical debt items such as code smells and code patterns. Recently, a couple of approaches have attempted more complex automation. DoctorQ by Dr. Jörg Rech is a plug-in for Eclipse that identifies the presence of anti-patterns while a developer works. Jdeodorant by Nikolaos Tsantalis is an Eclipse plugin that can automatically remove a few code smells, for example feature envy, from Java code. These tools provide a glimpse of what is possible with human-in-the-loop automation for addressing technical debt. Because of the risk of change previously mentioned, it is likely that any automation will require the developer to arbitrate between considerations to approve any proposed changes. Thus the automated technical debt challenge becomes to create automated solutions that address more complex technical debt patterns while the developer works in the code.

5.7 Social Debt in Software Engineering: Towards a Crisper Definition

Damian Andrew Tamburri (Polytechnic University of Milan, IT), Bill Curtis (CAST – Fort Worth, US), Steven D. Fraser (HP Inc. – Palo Alto, US), Alfredo Goldman (University of Sao Paulo, BR), Johannes Holvitie (University of Turku, FI), Fabio Queda Bueno da Silva (Federal University of Pernambuco – Recife, BR), and Will Snipes (ABB – Raleigh, US)

License © Creative Commons BY 3.0 Unported license

© Damian Andrew Tamburri, Bill Curtis, Steven D. Fraser, Alfredo Goldman, Johannes Holvitie, Fabio Queda Bueno da Silva, and Will Snipes

Sustainable and scalable software systems require careful consideration over the force known as technical debt, i.e., the additional project cost connected to sub-optimal technical decisions. However, the friction that software systems can accumulate is not connected to technical decisions alone, but reflects also organizational, social, ontological and management decisions that refer to the social nature and any connected social debt of software – this nature is yet to be fully elaborated and understood. In a joint industry & academia panel, we refined our understanding of the emerging notion of social debt in pursuit of a crisper definition. We observed that social debt is not only a prime cause for technical debt but is also tightly knit to many of the dimensions that were observed so far concerning technical debt, for example software architectures and their reflection on organizations. Also we observed that social debt reflects and weighs heavily on the human process behind software engineering, since it is caused by circumstances such as cognitive distance, (lack of or too much of) communication, misaligned architectures and organizational structures.

The goal for social debt in the next few years of research should be to reach a crisp definition that contains the essential traits of social debt which can be refined into practical operationalizations for use by software engineering professionals in need of knowing more about their organizational structure and the properties/cost trade-off that structure currently reflects.

5.8 An Advanced Perspective for Engineering and Evolving Contemporary Software

Guilherme Horta Travassos (UFRJ / COPPE, BR)

License © Creative Commons BY 3.0 Unported license

© Guilherme Horta Travassos

URL <http://www.cos.ufrj.br/~ght>

Software engineers continuously wonder about software and its nature. At the same time, software engineers should build high quality, low cost, on time and useful (software) products. However, the development of software does not follow a smooth path. The nature of software is more than technical (it is at least socio-technical), contributing to making the development process less predictable and prone to risks. Some issues, such as software defects, can be early identified, measured, and mitigated contributing to increasing its quality. Others, we lack evidence on their identification, measurement, and mitigation. In fact, software engineers are aware that some technical and socio-technical issues can jeopardize the software and its evolution. The decisions made by software engineers throughout the software life cycle are sources of technical and socio-technical issues. Such decisions can get a software project into “debt” and will affect the software project in the future. Some “debts” are invisible (software engineers can feel, but cannot see), others perceivable (software engineers can see,

but not feel), others concrete (software engineers can see and feel) and finally others are still intangible (no feel, no see, and something still happens in the project). All of them can influence positively or negatively the software project depending on the software ecosystem conditions. Such debts represent a promising perspective to deal with the quality and risks in contemporary software projects. The characterization and understanding of technical and no technical software debts represent an important step towards a better comprehension of the engineering process, the software nature, and its evolution.

Participants

- Areti Ampatzoglou
University of Groningen, NL
- Francesca Arcelli Fontana
University of Milano-Bicocca, IT
- Rami Bahsoon
University of Birmingham, GB
- Ayse Basar Bener
Ryerson Univ. – Toronto, CA
- Frank Buschmann
Siemens AG – München, DE
- Alexandros Chatzigeorgiou
University of Macedonia –
Thessaloniki, GR
- Zadia Codabux
Mississippi State University, US
- Bill Curtis
CAST – Fort Worth, US
- Steven D. Fraser
HP, Inc. – Palo Alto, US
- Alfredo Goldman
University of São Paulo, BR
- Christine Hofmeister
East Stroudsburg University, US
- Johannes Holvitie
University of Turku, FI
- Clemente Izurieta
Montana State University –
Bozeman, US
- Andreas Jedlitschka
Fraunhofer IESE –
Kaiserslautern, DE
- Sven Johann
innoQ GmbH – Monheim am
Rhein, DE
- Heiko Koziolk
ABB AG Forschungszentrum –
Ladenburg, DE
- Philippe Kruchten
University of British Columbia –
Vancouver, CA
- Jean-Louis Letouzey
inspearit – Paris, FR
- Antonio Martini
Chalmers UT – Göteborg, SE
- John D. McGregor
Clemson University, US
- Mehdi Mirakhorli
Rochester Institute of
Technology, US
- J. David Morgenthaler
Google, Inc. –
Mountain View, US
- Robert Nord
Carnegie Mellon University –
Pittsburgh, US
- Ipek Ozkaya
Carnegie Mellon University –
Pittsburgh, US
- Fabio Queda Bueno da Silva
Federal University of
Pernambuco – Recife, BR
- Klaus Schmid
Universität Hildesheim, DE
- Carolyn Seaman
University of Maryland,
Baltimore County, US
- Andriy Shapochka
SoftServe – Lviv, UA
- Forrest Shull
Carnegie Mellon University –
Pittsburgh, US
- Will Snipes
ABB – Raleigh, US
- Damian Andrew Tamburri
Polytechnic Univ. of Milan, IT
- Graziela Tonin
University of São Paulo, BR
- Guilherme Horta Travassos
UFRJ / COPPE, BR

