Five Reasons for Including Technical Debt in the Software Engineering Curriculum

Davide Falessi
California Polytechnic State University
San Luis Obispo, CA, USA
dfalessi@calpoly.edu

Philippe Kruchten
University of British Columbia
Vancouver, BC, Canada
pbk@ece.ubc.com

ABSTRACT

Technical Debt is a useful metaphor to explain some of the difficulties of software evolution. The concept of Technical Debt is gaining importance from a scientific perspective, as the number of related papers, special issues, and international events grow over the years. From a practical perspective, the number of tools related to Technical Debt and their industrial adoption grow as well. Despite this high interest, Technical Debt is not yet included in the software engineering curriculum and hence the greater part of students graduating in computer science or software engineering does not know about Technical Debt. The aim of this paper is to discuss the inclusion of Technical Debt in the software engineering curriculum. We claim that Technical Debt should be treated as a first class entity the same as Requirements Engineering, Software Design and Architecture, and Software Testing. We support our claim by presenting five reasons why Technical Debt should be included in the software engineering curriculum.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: General.

General Terms

Economics, Human Factors.

Keywords

Software engineering curriculum, technical debt.

1 INTRODUCTION

It is often remarked that software engineering is a highly dynamic field. Since the term was coined at the 1968 NATO conference, the field has seen an explosion in terms of the number of applications and products that use software, an immense growth in the sophistication and capabilities of those products, multiple revolutions in the way software relates to the hardware and networks over which it runs, and an ever-changing set of technologies, tools, and methods promising more effective software development.

Technical Debt is a metaphor introduced by Ward Cunningham in 1992 [2] showing how doing things in a "quick and dirty" way sets us up with a Technical Debt, which is roughly similar to a financial debt. Like a financial debt, such as a mortgage, the Technical Debt incurs interest, which comes in the form of the extra effort that we have to dedicate in future development because of this quick and dirty design choice. We can choose to continue paying the interest, or we can pay down the principal by refactoring the quick and dirty design into a better design.

Although there is a cost to paying down the principal, we gain by reduced interest payments in the future as noted by Fowler [15]. A more recent definition of Technical Debt which seems to aptly convey the current consensus is given by Steve McConnell [11]: "*A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time).*" Finally, another view on Technical Debt is "the invisible results of past decisions about software that negatively affect its future" [7].

A further cause of Technical Debt is organic decay. Specifically, new solutions, technologies or frameworks become available over time that renders the originally adopted ones obsolete. For instance, a specific design could be good enough for the first releases of a project but it could make the system too slow once the number of users increases over time.

Because Technical Debt is a metaphor, it can be applied to almost any aspect of software development, broadly encompassing anything that stands in the way of deploying, selling, or evolving a software system, or anything that adds to the friction from which software development endeavors suffer: test debt, people debt, architectural debt, requirement debt, documentation debt, code quality debt, etc. [14]

The metaphor saw little use for many years, but suddenly around 2000, and curiously in parallel with the advent of agile methods (though this may be just coincidental), it gained increased attention in both scientific and industry domain. From a scientific perspective, Technical Debt is gaining importance as the number of related conference papers (e.g., [17]), journal papers (e.g., [10] [18]), book chapters (e.g., [13] [1]), special issues (e.g., [7]), and international events (e.g., [8] [5]) grow over the years.

ECSAW'15, September 07-11, 2015, Dubrovnik/Cavtat, Croatia.
© 2015 ACM. ISBN 978-1-4503-3393-1/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2797433.2797462

From a practical perspective, the number of related tools (e.g., [9] [3]) and their industrial adoption grow as well.

Despite this high interest, Technical Debt is not yet included in the software engineering curriculum. As a result, the greater part of students graduating in computer science or software engineering does not know about Technical Debt.

Technical Debt is multidisciplinary in nature. It spans across multiple topics: code smells, software architecture, software quality, software maintenance, and it can be used to recap several topics already taught and describe how they relate to each other.

The aim of this paper is to discuss the inclusion of Technical Debt in the software engineering curriculum, using as a reference the "Curriculum Guidelines for Undergraduate Degree Programs in Computer Science" published by ACM and IEEE in 2013 [16], and the "Curriculum Guidelines for Graduate Degree Programs in Software Engineering" published by Stevens Institute of Technology in 2009 [12].

We claim that Technical Debt should be treated as a first class entity the same as Requirements Engineering, Software Design and Architecture and Software Testing. In the reminder of this paper we support our claim with five top reasons:

1. Technical Debt is a useful means to explain the realistic tradeoffs among business goals (e.g., immediate customer satisfaction vs. long-term maintainability).
2. Technical Debt supports tool interaction, which is something the students prefer over a teacher monologue.
3. Technical Debt supports the application of estimation models to support software engineering decisions.
4. Technical Debt supports the understanding of internal software quality, and the need of quantitative observations of that internal quality.
5. Technical Debt supports the application of economic models to support software engineering decisions.

The reminder of this paper elaborates on the abovementioned reasons.

2 REASONS FOR ADDING TECHNICAL DEBT

2.1 Explaining Realistic Tradeofss

Maintainability could be naively treated as a goal to achieve in every context. The same applies for other quality aspects like security, reusability, and customer satisfaction. However, because resources are limited in practice, not all these aspects can be met at the same time. These concepts are taught in the "SE - Tools and Environments" knowledge unit in undergraduate course [16] and in the "Systems Engineering and Software Engineering Processes" graduate course [12].

The concept of tradeoffs, because it is practical rather than theoretical, requires a "hands-on" explanation. The Technical Debt metaphor explicitly links the tradeoffs between maintainability and time-to-market and allows the students to realize that it is impossible to meet all the goals at the same time, regardless of the efficiency or the effectiveness of the development team. For instance, conversations with architects, developers, testers and project managers have revealed a large consensus over the fact that all projects have some Technical Debt as no project is perfect [4]. At first this may seem unexpected since high maturity organizations should produce nearly perfect products. However, high maturity implies an increased capability of controlling and managing Technical Debt because important historical data is available to quantitatively manage and control key project and organizational processes. Thus, we observe a balancing act [10]: even in mature companies, the acceptance of some Technical Debt may be required to achieve profitable time-to-market goals.

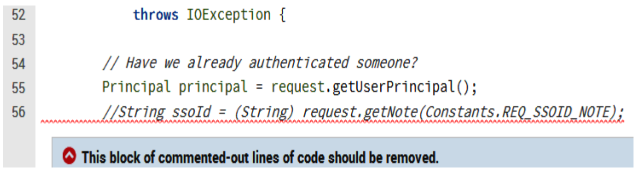
In conclusion, we claim that Technical Debt is a viable means for teaching students both the theoretical and the practical importance of tradeoffs in software engineering.

2.2 Supporting Interactivity

Technical Debt management requires interactions with static analyzer tools (e.g., [9] [3]). These concepts are taught in the "SE - Tools and Environments" knowledge unit in undergraduate course [16] and in the "Systems Engineering and Software Engineering Processes" graduate course [12].

It is well known that students prefer interactions over a teacher monologue. Moreover, tools for monitoring Technical Debt are an effective way to show what quality rules violations (i.e., code smells) are, and what their consequences are (e.g., extra defects or extra effort). Current static analyzers support the analysis of several hundreds of different code smells including code clones, god classes, wrong variable naming, etc.

Fig. 1 below shows an excerpt of rule violation as reported by SonarQube for line 56 of the "SSLAuthenticator.java" class of the Apache Tomcat system, see online (http://goo.gl/9l4vmP) for the full SonarQube output. The quality rule being violated in the example is titled "Avoid commented-out lines of code", which dictates that code should not be commented out but should be simply removed. The rationale of the rule is that commenting out code only has disadvantages and no advantages. Specifically, there is no advantage in commenting out code because all the old versions of the code are already available in the version control system (e.g., Git). The disadvantage of commenting out code is that comments may be removed by mistake causing that code to become unintentionally active and eventually produce undesirable system behaviors. In conclusion, the reported example clearly shows that the violation is a system property invisible to the user. As such, it is clearly different from a defect, although the violation can lead to a defect.



```
52              throws IOException {
53
54          // Have we already authenticated someone?
55          Principal principal = request.getUserPrincipal();
56          //String ssoId = (String) request.getNote(Constants.REQ_SSOID_NOTE);
```

This block of commented-out lines of code should be removed.

Fig. 1 Example of a rule violation as reported by SonarQube.

This variety of quality rules is probably wider than the one presented in any textbook on software engineering. Therefore, we claim that tools for monitoring Technical Debt are effective means for teaching software engineering practices that complement the use of standard textbooks.

2.3 Supporting the Application of Estimation Models

In the context of Technical Debt, the interest refers to the cost that will be incurred by not fixing the technical problem (i.e. the consequences of not removing the debt). For instance, the interest related to a component of a system with a high coupling and cohesion refers to the extra effort that will be necessary to maintain the component in the future.

We note that while the principal is certain in the presence of Technical Debt, the interest has an associated probability of occurrence. In other words, you can be sure that refactoring a component will cost you something (i.e., $500), but you cannot be sure about the consequences of not refactoring it. Therefore, estimating interest means estimating both the amount and its probability of occurrence.

The concept of estimation is taught in the "SE - Software Project Management" knowledge unit in undergraduate course [16] and in the "Software Engineering Management Knowledge Area" in graduate course [12].

In conclusion, managing Technical Debt is an effective means for connecting advanced statistics topics to the software engineering domain.

2.4 Supporting Empiricism

Empirical activities like surveys or controlled experiments are applied, to some extents, by every graduate student involved in research in software engineering. Technical Debt encapsulates new aspects of empiricism by providing a context-dependent way of thinking about software quality across lifecycle phases, and is tractable to quantitative analysis and hence objective observations.

Technical Debt provides us a useful framework for guiding all kinds of empirical work (especially the reminder to measure not only instances of rework but the opportunity costs if that rework is not performed) and effectively transmitting the results to practitioners (by recognizing the existence and rational management of tradeoffs).

In conclusion, we claim that Technical Debt is a viable means for teaching students both the theoretical and the practical importance of empiricism.

2.5 Supporting the Application of Economic Models

Economic models are taught in the "SP - Economies of Computing" knowledge unit in undergraduate course [16] and in the "Engineering Economics Knowledge Area" in graduate course [12].

Managing Technical Debt consists of exploiting information about principal and interest to support tradeoff analysis and decision-making. Different economic models could be used to manage Technical Debt including:

- Portfolio management: In finance, a portfolio refers to a bundle of assets of different types held by an investor. Different types of assets or even different assets of the same type (e.g. real estate holdings), usually exhibit different volatility and performance patterns. Therefore, holding a portfolio of assets, i.e. diversifying the investment, is less risky than investing in a single asset because it is unlikely that all assets will succumb to their associated risks at the same time. Thus, portfolio management is similar to the process of Technical Debt management in which software managers make decisions on when and what Technical Debt items should be paid or kept to maximize the benefit to the organization. By definition, different Technical Debt items represent different types of risk, some of which might be related. The portfolio approach allows for the specification of these relationships so that they can be taken into account when calculating the optimal portfolio. The Modern Portfolio Theory is a mean-variance analysis model that uses the expected return and the return variance to measure the performance of a portfolio [2]. These models can be tailored for Technical Debt management [6].
- Real options: Paying off an instance of Technical Debt (e.g., by refactoring) can be seen as an investment decision, in that it incurs relatively certain short-term costs (a commitment of time, money and/or resources) in the pursuit of more uncertain longer-term benefits in the form of maintenance cost-savings. The value of such an investment is in the form of options. Previously, researchers have considered applying the Black-Scholes valuation formula [46] or Baldwin and Clark's Net Option technique [3] to value architectural investments. These approaches, however, require the estimation of key parameters that are difficult to determine in practice, such as a "replicating portfolio" to estimate the future value of an asset, or the "technical potential" of each module. Another advantage is that most of the inputs to this model can be inferred from the interest probability, interest amount, and principal associated with a Technical Debt item.

In conclusion, we claim that Technical Debt is a viable means for introducing or connecting already taught economic models.

3 CONCLUSIONS

Despite the fact that Technical Debt is widely adopted in practice and subject to extensive research effort, it is not yet taught at the university. In this paper we presented five reasons why Technical Debt should be included in the software engineering curriculum. These reasons are meant to drive a discussion about the inclusion of the Technical Debt topic in the Software Engineering curriculum.

4 ACKNOWLEDGMENT

The work of Davide Falessi has been partially supported by Cisco research grant CG# 583194.

5 REFERENCES

[1] Cai, Y. et al. 2014. A Decision-Support System Approach to Economics-Driven Modularity Evaluation. *Economics-driven Software Architecture*. I. Mistrik et al., eds. Morgan Kaufmann.

[2] Cunningham, W. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*. 4, 2 (Apr. 1993), 29–30.

[3] Curtis, B. et al. 2012. Estimating the Principal of an Application's Technical Debt. *IEEE Software*. 29, 6 (Nov. 2012), 34–42.

[4] Falessi, D. et al. 2013. Practical Considerations, Challenges, and Requirements of Tool-Support for Managing Technical Debt. *Proceedings of the 4th International Workshop on Managing Technical Debt* (2013), 16–19.

[5] Falessi, D. et al. 2014. Technical Debt at the Crossroads of Research and Practice: Report on the Fifth International Workshop on Managing Technical Debt. *ACM SIGSOFT Software Engineering Notes*. 39, 2 (2014), 1–15.

[6] Guo, Y. and Seaman, C. 2011. A portfolio approach to technical debt management. *Proceedings of the 2nd Workshop on Managing Technical Debt*. (2011), 31–34.

[7] Kruchten, P. et al. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*. 29, 6 (Nov. 2012), 18–21.

[8] Kruchten, P. et al. 2013. Technical debt: towards a crisper definition report on the 4th international workshop on

managing technical debt. *ACM SIGSOFT Software Engineering Notes*. 38, 5 (Aug. 2013), 51.

[9] Letouzey, J.-L. and Ilkiewicz, M. 2012. Managing Technical Debt with the SQALE Method. *IEEE Software*. 29, 6 (Nov. 2012), 44–51.

[10] Lim, E. et al. 2012. A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *IEEE Software*. 29, 6 (Nov. 2012), 22–27.

[11] Managing Technical Debt: 2013. *http://www.sei.cmu.edu/community/td2013/program/upload/technicaldebt-icse.pdf*.

[12] Pyster, A. Graduate Software Engineering 2009(GSwE2009) -Curriculum Guidelines for Graduate Degree Programs in Software Engineering. *Stevens Institute*.

[13] Shull, F. et al. 2013. Technical Debt: Showing the Way for Better Transfer of Empirical Results. *Perspectives on the Future of Software Engineering*. J. Münch and K. Schmid, eds. Elsevier. 179–190.

[14] Sterling, C. 2010. *Managing Software Debt: Building for Inevitable Change (Agile Software Development Series)*. Addison-Wesley Professional.

[15] Technical Debt Quadrant: 2009. *http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html*.

[16] The Joint Task Force on Computing Curricula (Association for Computing Machinery IEEE-Computer Society) 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*.

[17] Zazworka, N. et al. 2013. A case study on effectively identifying technical debt. *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering - EASE '13* (2013), 42.

[18] Zazworka, N. et al. 2014. Comparing four approaches for technical debt identification. *Software Quality Control*. 22, 3 (2014), 1–24.