POLITECNICO DI TORINO
Repository ISTITUZIONALE

Organizing the Technical Debt Landscape

Original
Organizing the Technical Debt Landscape / Clemente, I., Vetro', A., Nico, Z., Yuanfang, C., Carolyn, S., Forrest, S.. - STAMPA. - (2012), pp. 23-26. (Third International Workshop on Managing Technical Debt (MTD '12) Zurich, Switzerland 5 June) [10.1109/MTD.2012.6225995].

Availability:
This version is available at: 11583/2497507 since:

Publisher:
IEEE COMPUTER SOC

Published
DOI:10.1109/MTD.2012.6225995

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

11 November 2026

Organizing the Technical Debt Landscape

Clemente Izurieta[1], Antonio Vetrò [2,5], Nico Zazworka[5],
Yuanfang Cai[3], Carolyn Seaman[4,5], Forrest Shull[5]

[1]*Dept. of Computer Science*
Montana State University
Bozeman, MT, USA
clemente.izurieta
@cs.montana.edu

[2]*Automatics and Informatics Dept.*
Politecnico di Torino
Torino, Italy
antonio.vetro@polito.it

[3]*Dept. of Computer Science*
Drexel University
Philadelphia, PA, USA
yfcai@cs.drexel.edu

[4]*Dept. of Information Systems*
UMBC
Baltimore, MD, USA
cseaman@umbc.edu

[5]*Fraunhofer CESE*
College Park, MD, USA
nzazworka@fc-md.umd.edu
fshull@fc-md.umd.edu

***Abstract*—To date, several methods and tools for detecting source code and design anomalies have been developed. While each method focuses on identifying certain classes of source code anomalies that potentially relate to technical debt (TD), the *overlaps* and *gaps* among these classes and TD have not been rigorously demonstrated. We propose to construct a seminal *technical debt landscape* as a way to visualize and organize research on the subject.**

***Keywords*-technical debt; Design Debt; Code Smells; Landscape.**

I. INTRODUCTION

The technical debt (TD) metaphor [22], created and initially driven by the agile community, is often discussed in blogs and other development forums. However, work aiming at putting the metaphor into a scientific context is only beginning. The first attempts to formalize TD into a scientific framework have been made by the authors and other participants in the MTD workshops [1]. These workshops, with participants from both industry and research communities, have confirmed that many practitioners are lacking a generally applicable body of knowledge on how TD can be organized, visualized, identified, and managed in their software projects. Thus, many developers are trying to implement their own customized TD solutions.

Herein, we propose work that aims to facilitate this process, by providing practitioners and researchers with a seminal *landscape* of existing approaches for organizing, visualizing, and identifying one important form of TD, i.e. debt resulting from anomalies in the source code. This initial landscape serves as an invitation for researchers to contribute to its further development. As the landscape evolves, we expect that tools and processes will become better able to cover existing gaps and handle overlaps in decision making abilities, TD management, payoff techniques, and the inter-relationships that exist between them. Gaps are types of technical debt that are important to practitioners but cannot be detected by any existing technique or tool solution. This will address the shortcomings of the current state of the art in this area, namely that the overlaps and gaps between the methods are not known, that the relationship between specific source code anomalies and TD has not been demonstrated, and that the use of techniques for organizing, visualizing, identifying and managing TD have not been provided in a form that is easily integrated into software practice.

II. RESEARCH APPROACH

In order to investigate the relations among various code and design anomaly analysis techniques, as well as their ability to find and diagnose TD, we have formulated two main research questions:

1. What are the overlaps and gaps among existing techniques?
2. To what extent do existing techniques help in identifying TD?

To address these questions, we propose a two-staged iterative research methodology, as illustrated in Fig. 1. Through a diverse set of empirical studies we can build support for understanding the landscape of TD with respect to the commonalities and differences between different existing techniques and tools, and which types of TD are worth managing by practitioners. In the first stage the focus is on identifying and characterizing the types of TD, resulting in a "draft" of the TD landscape. This information is extracted from:

i. Existing techniques and tools. For example, the existing approaches described in section III constitute the techniques and tools that the authors have worked with thus far. Current unpublished work [2], summarized later in this paper, suggests little overlap among different techniques. However, there are other such techniques described in the literature, and we expect that the proposed landscape will help serve as a way of framing and organizing further research on the subject.

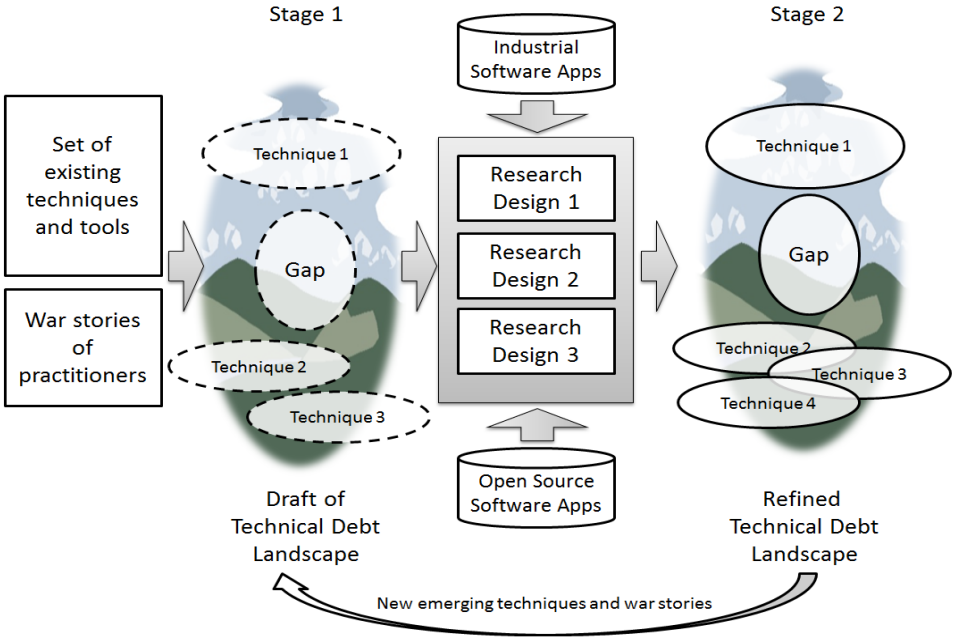

Figure 1. Technical Debt Landscape

ii. War stories reported by practitioners. Qualitative analysis of war stories (a particular interview technique meant to yield illustrative examples [3]) provides insight into the types of TD which practitioners struggle with most (and therefore are important to manage). Some of our colleagues have already collected a significant number of such war stories, which we and others should analyze in an effort to contribute to the TD landscape.

Stage 2 of our proposed approach involves refining and validating the draft landscape through targeted and coordinated empirical studies, resulting in a more accurate and complete TD landscape. We envision the following types of studies that would contribute to stage 2:

- **Design 1: Direct comparisons of TD identification techniques**: This design compares the output of two or more source code analysis techniques (e.g. those described in section III) applied to the same software system, to understand differences and commonalities between the outputs of these techniques. Variations of this design will include open source and commercial software systems and different combinations of techniques. Our unpublished study, described briefly in section IV, implements this design by comparing the output of four different techniques in an open-source software context, showing that the problems each detects are different but with some overlaps.
- **Design 2: Evaluating TD identification techniques for identifying real debt**: Existing TD identification techniques do, in fact, detect various forms of anomalies in the source code. It is not always clear, however, that these anomalies constitute TD, i.e. that they result in future maintenance problems if not corrected. This study design characterizes the usefulness of a technique to identify and quantify TD properties. This case study design would begin with the application of one of the TD detection techniques, followed by a focus group involving the developers of the code that was analyzed. The focus group participants will be asked to comment on how they would use the output of the source code analysis to manage TD, including how they could quantify the debt, how they would decide when (or if) to pay off the debt, and what the consequences of the debt are likely to be. An example of a study of this type is described in [24]. Studies following this design will help us to refine techniques for quantifying different types of debt.
- **Design 3: Evaluating the relationship between types of TD and future maintenance**: Experiments following this design would test the negative effects of TD on software quality using various indicators, e.g. introducing elevated defect rates, lowered maintainability, and higher cost of future changes. The basic design is to first produce two versions of a software module (e.g. a class or set of related classes), a "clean" version and a version that contains some type of code-based TD (e.g. grime, a code smell, etc.). Then subjects will be divided into two groups and both groups will be given the same maintenance task. One group will perform the maintenance task on the "clean" version and the other will modify the version with debt. The maintenance effort and resulting quality will be compared between the two groups. These controlled

designs will provide insight into which kinds and amounts of debt actually result in lower maintainability.

We expect that contributions from the research community in the form of these study designs and others, will help refine and validate our draft landscape, as depicted in stage 2 of Fig. 1.

III. Existing Approaches

There are a number of techniques and tools that could potentially be useful in the identification of source code-based TD, even if many of them were not developed with that aim in mind. We will not attempt to list them all here. However, very few, if any, have been validated with respect to their contribution to TD identification. To that end, we have begun the work of building the TD landscape by examining and comparing four specific techniques, described below in terms of their basic concepts and related work.

Modularity Violations (tool: CLIO) [4]. During software evolution, if two components always change together to accommodate modification requests but they belong to two separate modules that are designed to evolve independently, then there is a discrepancy. Such discrepancies can indicate TD as they may be caused by side effects of a quick and dirty implementation, or requirements may have changed such that the original designed architecture could not easily adapt. When such discrepancies exist, the software can deviate from its designed modular structure, which is called a *modularity violation*. Wong et al. [4] have demonstrated the feasibility and utility of this approach. In their experiment using Hadoop, they identified 231 modularity violations from 490 modification requests, of which 152 (65%) violations were conservatively confirmed by the fact that they were either indeed addressed in later versions, or were recognized as problems in the developers' subsequent comments.

Design Patterns and Grime Buildup. Design patterns are popular for a number of reasons, including but not limited to claims of easier maintainability and flexibility of designs, reduced number of defects and faults [5], and improved architectural designs. Software designs decay as systems, uses, and operational environments evolve, and decay can involve design patterns. Classes that participate in design pattern realizations accumulate *grime* – non-pattern-related code. Design pattern realizations can also *rot*, when changes break the structural or functional integrity of a design pattern. Both grime and rot represent forms of TD, in that the effort to keep the patterns cleanly instantiated has been deferred. In prior work Izurieta and Bieman [6, 22] introduced the notion of design pattern grime and performed a study of the effects of decay on three open-source systems, JRefactory, ArgoUML and eXist. They studied pattern realizations and found that coupling increased and namespace organization became more complex due to design pattern grime, but they did not find changes that "break" the pattern (design pattern *rot*). Izurieta and Bieman [7] also examined the effects of design pattern grime on the testability of JRefactory, a handful of patterns were examined, and they found that there are at least two potential mechanisms that can impact testability: 1) the appearance of design anti-patterns [8] and 2) the increases in relationships (associations, realizations, and dependencies) that in turn increase test requirements. They also found that the majority of grime buildup is attributable to increases in coupling.

Code Smells (tool: CodeVizard). The concept of code smells (aka bad smells) was first introduced by Fowler [9] and describes choices in object-oriented systems that do not comply with widely accepted principles of good-object oriented design (e.g., information hiding, encapsulation, use of inheritance). Code smells indicate where effort to improve the design has been deferred, hence indicate TD, and can be roughly classified into identity, collaboration, and classification disharmonies [10]. Automatic approaches (detection strategies [11]) have been developed to identify code smells. Schumacher et al. [12] focused on evaluating these automatic approaches with respect to their precision and recall, and others [13] [14] have evaluated the relationship between code smells (e.g., god classes) and the defect and change proneness of software components. This work showed that automatic classifiers for god classes work with high recall and precision when studied in industrial environments. Further, in these environments, god classes were up to 13 times more likely affected by defects and up to seven times more change prone than their non-smelly counterparts.

ASA issues (tool: FindBugs). Automatic static analysis (ASA) tools analyze source or compiled code looking for violations of recommended programming practices ("issues") that might cause faults or might degrade some dimensions of software quality (e.g., maintainability, efficiency). Some ASA issues can indicate TD as they are good candidates for removal through refactoring to avoid future problems. In previous work Vetró et al. [15] [16] analyzed the issues detected by FindBugs [17] on two pools of similar small programs (85 and 301 programs respectively), each of them developed by a different student, in order to verify which FindBugs issues were related to real defects in the source code. By analyzing the changes and test failures in both studies they observed that a small percentage of issues were related to known defects in the code. Some of the issues identified as good/bad defect detectors by the authors in these studies were also found in similar studies with FindBugs, both in industry [18] and open source software [19]. Similar studies have also been conducted with other tools [20] [21] and the overall finding is: a small set of ASA issues is related to defects in the software, but the set depends on the context and type of the software.

IV. Hadoop Case Study

Our strategy of investigating the research questions proposed in Section II is to apply different TD identification technologies to the same set of subject systems. Each technique reports a set of files to be problematic. We then study how these results overlap and what the gaps are. We also study the relation between these detected problematic files and quality issues, such as the existence of bugs. The

purpose is to investigate which techniques can detect problems that most likely lead to quality issues. These issues, presumably, contain more expensive TD, and should be taken care of sooner than others.

Our unpublished case study [2] using Hadoop produced three main findings: **a)** different TD techniques point to different classes and therefore to different problems; **b)** dispersed coupling, god classes, modularity violations and multithread correctness issues are located in classes with higher defect-proneness; and **c)** modularity violations are strongly associated with change proneness. These findings contribute to building an initial picture of the TD landscape. The initial result showed that these TD techniques are loosely overlapping and only a subset of them is strongly associated with defect and change proneness. This indicates that, in practice, multiple TD indicators should be used.

V. FUTURE WORK

In addition to comparing existing techniques for their overlaps, we will also investigate the *gaps* between existing techniques and TD identification. Gaps are types of TD that are important to practitioners but cannot be detected by any existing technique or tool solution. Research is needed to study or find techniques able to fill those gaps. We will also investigate quality factors other than defect and change proneness, such as productivity and maintenance difficulties. We envision a future when designers can use a well-developed, well-validated TD landscape to select and combine the results from different techniques to detect the TD items with most significance and impact, and further associate values and costs to make well-informed decisions on refactoring.

ACKNOWLEDGMENT

The participation of Seaman, Guo, Zazworka, Vetró, and Shull in this work is supported by the US National Science Foundation, award #0916699.

REFERENCES

[1] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," FoSER, Santa Fe, NM, USA, pp. 47-52, Nov. 2010.

[2] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, "Comparing Four Approaches for Technical Debt Identification," unpublished.

[3] W. Lutters and C. Seaman "The Value of War Stories in Debunking the Myths of Documentation in Software Maintenance." Information and Software Technology. 49(6):576-587, January. 2007.

[4] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in Proc. 33th International Conference on Software Engineering, May 2011, pp. 411–420.

[5] Y.G. Guéhéneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects," in Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), ser. TOOLS '01. Washington, DC, USA: IEEE Computer Society, 2001.

[6] C. Izurieta and J.M. Bieman, "How software designs decay: A pilot study of pattern evolution," First International Symposium on Empirical Software Engineering and Measurement, ESEM 2007, sept. 2007, pp. 449 –451.

[7] C. Izurieta and J.M. Bieman, "Testing consequences of grime buildup in object oriented design patterns,", 1st International Conference on Software Testing, ICST '08, april 2008, pp. 171 –179.

[8] W. J. Brown, R. C. Malveau, and T. J. Mowbray, "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis." Wiley, Mar. 1998.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code, 1st ed. Addison-Wesley Professional, Jul. 1999.

[10] M. Lanza and R. Marinescu, Object-oriented Metrics in Practice. Berlin: Springer-Verlag, 2006.

[11] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," IEEE International Conference on Software Maintenance, vol. 0, pp. 350–359, 2004.

[12] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:10.

[13] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in Proceeding of the 2nd working on Managing technical debt, ser. MTD '11. New York, NY, USA: ACM, 2011, pp. 17–23.

[14] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in Proceedings of the 2010 IEEE International Conference on Software Maintenance, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[15] A. Vetro', M. Torchiano, and M. Morisio, "Assessing the precision of findbugs by mining java projects developed at a university," in Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, may 2010, pp. 110 –113.

[16] A. Vetro', M. Morisio, and M. Torchiano, "An empirical validation of findbugs issues related to defects," in Evaluation and Assessment in Software Engineering (EASE), EASE 2011, April 2011.

[17] D. Hovemeyer and W. Pugh, "Finding bugs is easy," SIGPLAN Not., vol. 39, pp. 92–106, December 2004.

[18] N. Ayewah and W. Pugh, "The google findbugs fixit," in Proceedings of the 19th international symposium on Software testing and analysis, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 241–252.

[19] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in Proceedings of the Fourth International Workshop on Mining Software Repositories, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007.

[20] C. Boogerd and L. Moonen, "Evaluating the relation between coding standard violations and faultswithin and across software versions," in Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on, may 2009, pp. 41 –50.

[21] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 45–54.

[22] C. Izurieta and J.M. Bieman, "A Multiple Case Study of Design Pattern Decay, Grime, and Rot in Evolving Software Systems." Springer Software Quality Journal, ISSN: 0963-9314, DOI: 10.1007/s11219-012-9175-x, February 2012.

[23] W. Cunningham, "The wycash portfolio management system," in Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum), ser. OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 29–30. [Online]. Available: http://doi.acm.org/10.1145/157709.157715

[24] Y. Guo, C. Seaman, N. Zazworka, and F. Shull, "Domain-Specific Tailoring of Code Smells: An Empirical Study," in the 32nd ACM/IEEE International Conference on Software Engineering, 2010, pp. 167-170.