A Case Study on Effectively Identifying Technical Debt

Nico Zazworka
Fraunhofer USA Center for Experimental Software Engineering[1]
College Park, USA
+49 69 5050 4265
zazworka@gmail.com

Rodrigo O. Spínola
Graduate Program in Systems and Computer – UNIFACS[2]
Salvador, Brasil
+55 71 3330-4630
rodrigo.spinola@pro.unifacs.br

Antonio Vetro'
Dept. of Control and Computer Engineering – Politecnico di Torino
Torino, Italy
+39 011 090 71 69
antonio.vetro@polito.it

Forrest Shull
Fraunhofer USA Center for Experimental Software Engineering
College Park, USA
+1 240 487 2904
fshull@fc-md.umd.edu

Carolyn Seaman
Department of Information Systems
UMBC
Baltimore, USA
+1 410 455 3937
cseaman@umbc.edu

ABSTRACT

Context: The technical debt (TD) concept describes a tradeoff between short-term and long-term goals in software development. While it is highly useful as a metaphor, it has utility beyond the facilitation of discussion, to inspire a useful set of methods and tools that support the identification, measurement, monitoring, management, and payment of TD. **Objective**: This study focuses on the identification of TD. We evaluate human elicitation of TD and compare it to automated identification. **Method**: We asked a development team to identify TD items in artifacts from a software project on which they were working. We provided the participants with a TD template and a short questionnaire. In addition, we also collected the output of three tools to automatically identify TD and compared it to the results of human elicitation. **Results**: There is little overlap between the TD reported by different developers, so aggregation, rather than consensus, is an appropriate way to combine TD reported by multiple developers. The tools used are especially useful for identifying defect debt but cannot help in identifying many other types of debt, so involving humans in the identification process is necessary. **Conclusion**: We have conducted a case study that focuses on the practical identification of TD, one area that could be facilitated by tools and techniques. It contributes to the TD landscape, which depicts an understanding of relationships between different types of debt and how they are best discovered.

Categories and Subject Descriptors

K.2.7 [**Management of Computing and Information Systems**]: Software Management - *Software Maintenance*

General Terms

Management, Measurement, Experimentation, Human Factors.

Keywords

Technical Debt, Software Maintenance, Automatic Static Analysis, Code Smells.

1. INTRODUCTION

Technical debt (TD) is a metaphor that describes the tradeoff between the short-term payoffs (such as a timely software release) of delaying some technical development activities and the long-term consequences of those delays [1]. It has facilitated discussion among practitioners and researchers by providing a familiar vocabulary from the financial domain and has potential to become a truly universal language for communicating technical tradeoffs.

Our vision for TD management, however, goes beyond facilitating communication, to the development of a whole set of tools and techniques, inspired by the TD metaphor. This toolset must include tools for TD identification, which is essential to make TD manageable and explicit, creating a TD "portfolio" that allows better control of the debt situation. TD identification approaches can be broadly categorized as those that elicit TD instances from humans (i.e. developers and other stakeholders), and automated tools of various kinds to detect potential debt in the source code. Human, manual approaches are likely to be more time-consuming, but have two advantages (at least in theory) over automated approaches. One is that they might be more accurate, i.e. more likely to identify TD that is most significant, while automated analyses may reveal many anomalies that turn out to be unimportant. The other advantage is that human stakeholders might be able to provide additional important contextual information related to each instance of TD (e.g. effort estimates, impact, decision rationale, etc.) that is difficult or even impossible to glean from analysis tools.

The first contribution of our work is to better understand how to elicit TD from humans. We propose and assess a TD template [2] that can be used to capture, store, and communicate essential

[1] Dr. Zazworka is currently employed at Elsevier Information Systems GmbH, Frankfurt, Germany.

[2] Dr. Spínola was previously employed at Kali Software, Rio de Janeiro, Brazil, and was a visiting post-doc at UMBC for the time of the study.

properties of TD that can feed into further decision making processes about debt repayment. Besides the template, our case study gives some insight into the dynamics of eliciting TD from a team of developers, all familiar with different aspects of the system being analyzed.

As a second contribution we compare several types of tool support for TD identification. Some recent research has addressed the issue of how close automated approaches can get to the results of manual TD identification. Some of these studies have indicated that it is possible to identify certain classes of potential TD (in particular design debt) with computer-assisted methods [3][4]. Moreover, they have demonstrated that detection approaches can succeed in finding issues that are of value to developers [3]. However, despite the fact that these approaches point to system code fragments that need improvement, it is not clear yet if they point to the most important TD, from software project stakeholders' point of view. Based on the current state-of-the-art, we studied three automated approaches (code smells, automated static analysis issues, and collection of code metrics), and how their output compares to TD that is elicited from humans. All of these approaches are well-established and have been extensively studied, however, to be fair, none of these approaches claims to identify a wide range of TD. Each of them aims to identify specific deficiencies in code or measure a specific quality dimension of software. Since these tools are available and stable we are interested in exploring the extent to which they can support TD identification, how big of a gap they leave if used without human elicitation of TD, and whether new tools might be warranted, possibly derived from knowledge of manual TD inspections. This understanding can help address questions such as how tools can best be used, instead of or in addition to manual approaches, in the identification of TD.

This study, and others like it, will help evolve the TD landscape [5], which lays out the different types and flavors of TD that exist in real software projects with respect to their importance and overlaps, and how those types are best identified by tools and other methods. Such an evolved landscape is essential to achieve our vision of building an effective toolkit for identifying, quantifying and managing TD.

2. BACKGROUND AND RELATED WORK

According to Seaman and Guo [2], the management of TD can center on a TD list, which is similar to a task backlog. The backlog contains TD items (in the following simply referred to as items), each of which represents a task that was left undone, but that runs a risk of causing future problems if not completed. Each item includes a description of what part of the system the debt item is related to, why that task needs to be done, and estimates of the TD's principal and interest, as well as some other attributes, as shown in Table 1.

The principal refers to the cost to fully eliminate the debt, i.e. to completely repair the technical imperfection. Depending on the type of TD this can translate into different kinds of activities, such as adding missing documentation, refactoring code that is hard to maintain, or maintaining a set of regression tests to align with the code and requirements. The cost of TD repair might be understood better in some cases than in others. For example, adding missing documentation might be more straightforward to estimate than a more complex code refactoring. Seaman and Guo [2] propose to initially estimate the principal on a rough ordinal scale from *low* to *medium* to *high*, that allows enough understanding to contribute in iteration planning. To further help in estimating principal, historical effort data can be used to make a more accurate and reliable estimation beyond the initial high/medium/low assessment. For example, if a debt item is a set of classes that need to be refactored, the historical cost of modification of those classes can be used as the future modification cost (principal of the debt item) estimation.

The second main component of TD is *interest*, which is composed of two parts: (1) the *interest probability* is the probability that the debt, if not repaid, will make other work more expensive over a given period of time or a release; (2) the *interest amount* is an estimate of the amount of extra work that will be needed if this debt item is not repaid.

Interest probability can be estimated using, e.g., historical usage and defect data. In addition, it is also important to consider the time variable because probability varies over time. For example, the probability that a module that needs refactoring will cause problems in the next release (because modifications will need to be made to it) may be very low, but that probability rises if we consider longer periods of time, e.g. over the next year or 5 years.

Finally, interest amount can also be estimated using historical data. Like TD principal, data on past defects, effort, and changes can be useful.

In addition to the financial properties of TD, several properties that support decisions on repayment are captured in the TD template:

1. The type of debt can be helpful to tailor debt payment to critical quality characteristics of interest. For example, known *defect debt* (known latent defects that have not been fixed) may be differently perceived in life critical software applications. Other known TD types are: *design debt* (an imperfection of the software's design or architecture negatively affecting future maintenance), *documentation debt* (missing, outdated, or incomplete documentation), and *testing debt* (missing test cases, test cases that are not executed, or missing test plans).
2. Was the original decision to go into debt made *intentionally or unintentionally*? This information can help to understand how explicit debt and TD decisions are managed in a project.

Table 1. The TD Template

ID	*TD identification number*
Responsible	*Person or role who should fix this TD item*
Type	*design, documentation, defect, testing, or other type of debt*
Location	*List of files/classes/methods or documents/pages involved*
Description	*Describes the anomaly and possible impacts on future maintenance*
Estimated principal	*How much work is required to pay off this TD item on a three point scale:* ***High/Medium/Low***
Estimated interest amount	*How much extra work will need to be performed in the future if this TD item is not paid off now on a three point scale:* ***High/Medium/Low***
Estimated interest probability	*How likely is it that this item, if not paid off, will cause extra work to be necessary in the future on a three point scale:* ***High/Medium/Low***
Intentional?	***Yes/No/Don't Know***

3. Who is *responsible* for fixing the TD? This information is important for administrative reasons and may help to provide a basis for assessing principal and interest.
4. Where is the TD *located*? This information is important to understand impact on the product, relationships between items, and ripple effects in source code when repaying the debt.

The process of managing TD using this approach starts with detecting TD items to construct the TD list. The next step is to measure the debt items on the list by estimating the principal, interest amount and interest probability. Then the debt items are monitored and decisions can be made on when and what debt items should be paid or deferred.

In this work, we are focused on the first step of this process: TD Identification. We can use different strategies to find TD items for each TD type. Two automated strategies that have been proposed to support the identification of design debt in software projects are identification of code smells and issues raised by automatic static analysis (ASA) tools, aka ASA issues.

Code smells refer to potential violations of good object-oriented design principles in source code and can be identified by comparing values of software metrics to defined thresholds [6]. Past studies have shown that some code smells are correlated with defect- and change-proneness [7]. In this study, we use CodeVizard [8] to detect 10 code smells as proposed in [6].

ASA is a reverse engineering technique that consists of extracting information about a program from its source or source code based artifacts using automatic tools [9]. ASA tools look for issues in terms of violations of recommended programming practices and potential defects that might cause faults or might degrade some dimensions of software quality (e.g., maintainability, efficiency). Issues should be removed through refactoring to avoid future problems, and thus may constitute TD. Many ASA tools exist; for this study we selected FindBugs, which is widely used in the literature and already used in past work [10][11].

In addition to code smells and issues, in this study we also collected basic structural code metrics for size and complexity, in order to study whether any relationship exists between high levels of these metrics and the existence of TD.

3. THE STUDY

3.1 Context

The study was conducted at Kali Software, a small software development company located in Rio de Janeiro, Brazil, that develops primarily web applications written in Java and based on the MVC framework. The project we studied consisted of a small application of 25K non-commented lines of code. It is a database-driven web application for the sea transportation domain. It has undergone a full product lifecycle (elicitation, design, implementation, deployment, and maintenance). The project team is composed of five professionals: two developers, one maintainer, one tester, and one project manager who also plays the role of the requirements analyst.

3.2 Goal and research questions

The goal of the research is to understand the human elicitation of TD and compare it to automated TD identification. The study's research questions are:

RQ1- Do the TD identification tools find the TD items that were reported by the developers?

RQ2-How much overlap is there between the TD items reported by different developers?

RQ3-How hard is the TD item template to fill in?

Given the exploratory nature of this case study and the small amount of TD items collected (21), we chose to address the research questions using exploratory analysis and presentation of data, rathter than a confirmatory type of analysis.

3.3 Procedure and Data Collection

The development team was first trained in TD basic concepts by the second author, including an opportunity for Q&A, via Skype. All the material was in Portuguese since this is the natural language of the development team.

During the training, only abstract TD items were used as examples (for instance: "TD items" on house or car repair) to avoid bias on identifying TD items during the second phase of the study. All types of TD (defect, design, etc.) were described, but no examples were given.

After the training, two parallel activities took place: manual and automatic TD identification, i.e. collecting TD items from the development team and collecting the output of tool-based analysis on the source code.

For the manual identification of TD, the development team (project manager, developers, and testers) were asked to report TD items individually. For this, we provided the team members with a short questionnaire to both report the TD items through the TD template (question 1) and provide information about the difficulty of documenting debt items (questions 2 to 5). The respondents were asked to document up to five of the most pressing TD items they knew of in the current version of the software. The questions were the following:

1. If you were given a week to work on this application, and were told not to add any new features or fix any bugs, but only to address TD (i.e. make it more maintainable for the future), what would you spend your time on?
2. How difficult was it to identify TD items?
3. How difficult was it to report TD items (i.e. fill in the template)?
4. How much effort did you need to identify and document all the TD items?
5. Which are the most difficult fields to fill in / which are the least difficult ones?

All answers were given as free text, although the respondents were asked to use the TD template to answer question 1.

In parallel to the questionnaire, we applied the CodeVizard and FindBugs tools to the latest version of the subject project source code, in order to identify code smells and ASA issues. The resulting data described, for each file (i.e. class) in the code base, how many of each type of code smells were identified, and how many of each type of ASA issues were present. Each FindBugs issue has a category, (e.g., Performance, Correctness, etc.), and a priority from 1 (highest) to 3 (lowest).

Regarding the structural metrics, we selected and computed for each file the following ones: Lines of Code, McCabe's Cyclomatic Complexity, Density of Comments, and Sum of Maximum Nesting of all Methods in a Class.

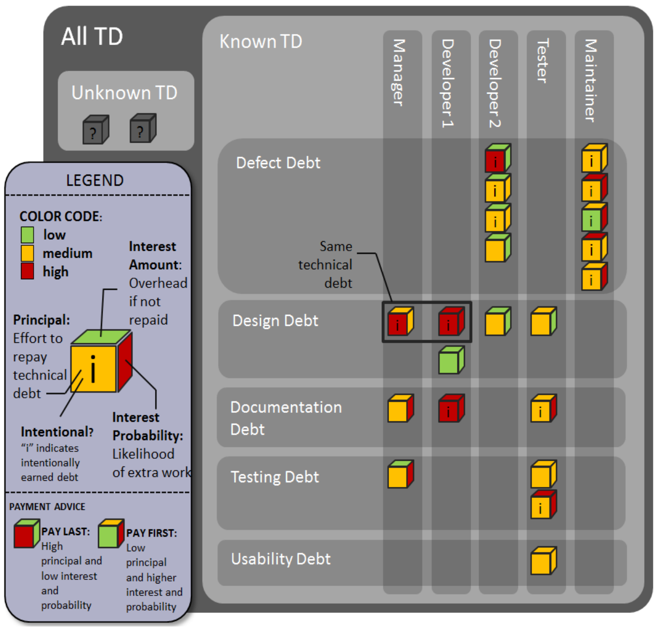
All TD
Unknown TD
Known TD
Manager
Developer 1
Developer 2
Tester
Maintainer
Defect Debt
Same technical debt
Design Debt
Documentation Debt
Testing Debt
Usability Debt
LEGEND
COLOR CODE:
low
medium
high
Interest Amount: Overhead if not repaid
Principal: Effort to repay technical debt
Intentional? "i" indicates intentionally earned debt
Interest Probability: Likelihood of extra work
PAYMENT ADVICE
PAY LAST: High principal and low interest and probability
PAY FIRST: Low principal and higher interest and probability

Figure 1. Results of the human elicitation of TD items

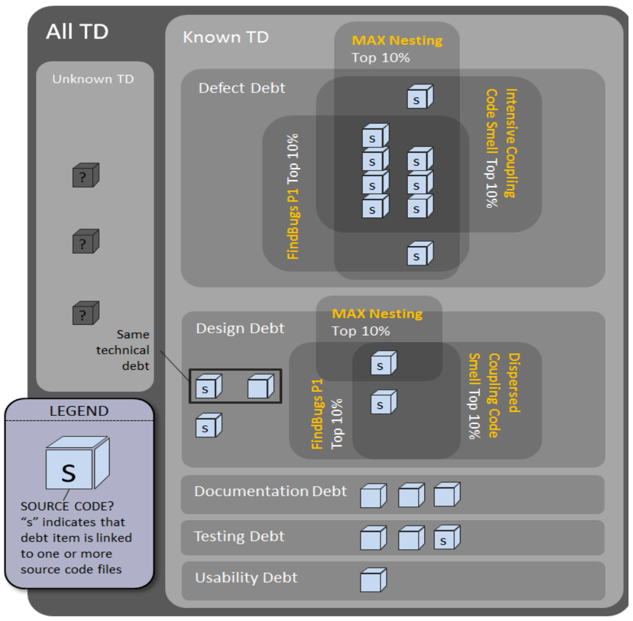

Figure 2. Results of the tools compared to human elicitation.

Lines of Code and McCabe's Cyclomatic Complexity are widely used in the literature of defect and maintainability prediction (e.g., [12], [13]). Higher accidental complexity is hypothesized to point to TD since complexity increases maintenance cost (TD interest). Density of Comments was selected to study whether highly commented code might have a relationship with TD, while Max Nesting measures complexity in depth similar to McCabe's Complexity measure.

The metrics were computed with ad-hoc scripts/tools.

4. RESULTS

Results in Figure 1 show how the 21 TD items identified by the software team, each represented as a colored box, were distributed over project roles and types of debt. Note that one new TD type, usability debt, was introduced by one of the subjects to describe the lack of a common user interface template.

As the legend indicates, each box has three faces, corresponding to principal (front), interest probability (right side) and interest amount (up). Each face can be green, yellow or red with respect to the estimation of the team member (respectively low, medium and high). An "i" on the front face indicates that the debt was intentionally introduced.

Figure 2 shows the results of automated identification approaches (FindBugs, Code Smells, Metrics) compared to the items reported by the development team. Each box in Figure 2 corresponds to one of the boxes (elicited TD items) shown in Figure 1. An "s" on the front face of a box shows that TD item was located in the source code by the subject who reported it.

We pre-filtered the results and we chose only the best automatic predictors of manually elicited TD to present here. For each automatically generated indicator, we sorted the source code files by the number or severity of issues found. For example, we sorted by the number of FindBugs Priority 1 issues, and by the value of the metric MAX nesting, and the number of each kind of code smell found. For each indicator, we examine the top 10% of the sorted list and determined how many source files in that 10% corresponded to TD items reported by developers. The indicators having the most developer-reported source files in the top 10% were FindBugs P1 issues, the MAX nesting metric, and Intensive Coupling code smell. We present results from these three indicators in Figure 2.

We realize that this filtering approach somewhat biases our results by only showing the automated tool results that performed best. Our motivation was to determine simply if any of the automated approaches were related to the TD elicited from developers, and to simplify the presentation of results. Clearly, we cannot claim that these three top performing indicators in this study would also be the best ones in any given case.

The overlapping shaded areas in Figure 2 depict the overlaps between the TD items reported by the human subjects (shown in Figure 1), and the TD items found by the top 3 automated indicators (MAX Nesting, FindBugs P1, and Intensive Coupling). For example, the shaded area labeled "Defect Debt" contains the 9 defect debt items reported by the development team (actually by Developer 2 and the Maintainer), and that are depicted in the "Defect Debt" shaded area of Figure 1. Figure 2 shows that, of these 9 defect debt items, 7 were also found by all three automated indicators.

The answers to the research questions follow.

RQ 1- Do the TD identification tools find the TD items that were reported by the developers?

We observe in Figure 2 that the three top automated approaches do about equally as well as manual elicitation in identifying defect debt. As shown in Figure 2, the three best automated indicators captured all source code components identified by the development team as having defect debt.

For design debt, automated approaches capture about half of the human-reported TD items, although ASA issues (in particular P1 issues) and code smells (i.e. Intensive Coupling) identify more TD items than traditional code metrics (i.e. MAX Nesting). One design debt item was located in a source file identified by all automatic approaches, and another was identified by ASA issues and code smells, but not traditional metrics. Three developer-reported design debt items (two of which were actually the same item) were not identified by any automated approach. The first item (covered by two reports) was described as TD caused by information stored redundantly in two databases. Subjects reporting this TD said that this kind of debt was distributed over many files. Thus, it is likely that the tools investigating single lines, methods, and classes could not point out this type of TD easily. The second missed item was reported as insufficient use of system resources (i.e. memory usage). This type of TD was missed even though one approach (FindBugs) reports on bug patterns related to performance problems.

Seven human-reported TD items were of type documentation debt, testing debt, or usability debt. Only one of these TD items was related to source code files, all other were related to other development artifacts (e.g. requirements documents and test plans).

Summarizing, these results lead to answer RQ1 in the following way: tools can support the identification of defect and design debt in this project, but not other types of debt that were found by developers.

RQ2- How much overlap is there between the TD items reported by different developers?

Only one TD item was reported by two different stakeholders (the manager and one developer). None of the remaining 19 items were reported by more than one stakeholder. This result indicates that, in this project, TD knowledge is dispersed and perceived differently by different stakeholders. The software tester reported the widest range of TD types including one previously unknown type, usability debt.

RQ 3-How hard is the TD item template to fill in?

The five study subjects reported that it took between 50 minutes and 2 hours to identify and document the TD items (average of 19 minutes per item). Answers about difficulty of the task ranged from "easy" to "difficult/high" (all answers were given as free text). Subjects agreed that the fields principal, interest amount, and interest probability were the most difficult to fill in. Location, type, and responsible were commonly noted as the least difficult fields. These results indicate that, in this project, the initial elicitation of TD items could be done in reasonable time, but that the key financial parameters of TD were difficult to estimate and might require better process or tool support in future.

5. DISCUSSION

The results addressing our research questions showed that in the project studied:

- the tools used identified all files affected by developer-identified defect debt, two out of five files affected by design debt items, and no other types of debt identified by developers in code or other artifacts;
- different stakeholders identified different debt;
- the initial human elicitation of TD items could be done in reasonable time, but that the key financial parameters of TD were difficult to estimate;

Some additional observations, beyond or complementary to the scope of the research questions, are possible from the data collected.

The first observation is that the majority of items reported by developers fall into the defect debt category. This indicates that known defects are of concern to the development team of this project.

Secondly, the colors of the TD items in Figure 1, indicating principal, interest amount, and interest probability, are rather equally distributed among the different types of debt. This suggests that debt characteristics are not tied to the type of debt, i.e. no type of debt has noticeably higher overall interest or principal.

Thirdly, we find intentionally earned debt in almost every category, except usability. This is especially interesting for defect debt. Many of the defect debt items were requirements that were not fully implemented. The intentionality of these items indicates that a decision was made to not fully implement those requirements, most likely due to time constraints, which makes these instances conceptually different from defects caused by unintentional programming mistakes.

Further, many developer-identified TD items could not have been found by the tools or metrics since the artifacts they were located are not included in the static code analysis. This suggests that a focus on source code as the single source of TD is too narrow, as developers reported a significant number of such items among their most important. Future studies might consider including or proposing tools for other kinds of development artifacts affected by TD.

Finally, we think that the color coding in Figure 1 hints at how this information can be further used to manage debt and make clearer decisions on which debt to pay. For example, items that have generally a low principal (e.g. green front face), but yellow or red interest characteristics are good candidates for paying off first, since their return on investment is more favorable than for other items. This idea of a cost/benefit decision approach has been previously proposed and discussed in [14].

6. THREATS TO VALIDITY

As with any case study, especially of a small project such as this one, threats to external validity are significant. We accept these threats, and attempt to trade off breadth for depth, by doing a thorough analysis of a small case, yielding deeper insights that would not have been possible in a much larger sample.

An important construct threat derives from the following assumption made in the design of this study: we assumed that the perceptions of software developers about the TD in their code can serve as a "ground truth" against which other types of TD identification can usefully be compared. However, the ultimate and authoritative "ground truth" for studies of TD would be a measure based on future maintenance effort associated with TD items. That is, a "real" TD item is one that leads to higher maintenance effort than would have been incurred if the debt did not exist. However, measuring "real" TD in this way was not possible in this study, nor is it in many studies. For the study in this paper, the assumption we have made represents a threat to construct validity in the sense that the TD reported by the developers might not lead to future increases in maintenance cost.

Another assumption that we have made that may also lead to a construct threat is that, when an automated approach identifies a problem in a source code module that is also indicated in a TD item reported by a developer, that the two indicators are actually pointing to the same TD instance, not to two separate TD instances that happen to reside in the same source module. While this assumption may not be strictly true, from a practical perspective it is reasonable, as fixing one instance of TD in a class (e.g. by refactoring) will very often, as a side effect, fix other instances of TD in the same class.

7. CONCLUSIONS

The TD metaphor has the potential to go beyond a mechanism for communication, to be translated into a whole set of tools and methods for measuring and managing debt. We have presented and evaluated how the TD list can be populated by developers through a common TD template, and how existing tool approaches can help to identify certain types of debt. We have further shown that different stakeholders know about different debt in their project, indicating that TD elicitation should include a range of project team members. Aggregation, not consensus, would appear to be the most effective approach to combining the input from different team members. In addition, three different automated approaches - code smells, ASA issues, and traditional code metrics - did well in pointing to source code files with defect debt, and also could point to a partial set of files with design debt.

This study raised, but did not answer, a number of interesting and important questions that we commend to future researchers (including ourselves):

- How much of the potential TD reported by tools, but not reported by developers, is "real" TD? That is, is there a value in using tools to identify TD that developers are not aware of?
- How can manual TD identification be better integrated into the development process, in order to make it more efficient and feasible?
- Is developer-identified or tool-identified TD more likely to lead to future maintenance issues (i.e. which is more likely to be "real" TD)?

We also encourage practitioners to use the proposed template in their projects and to share results and experiences (e.g. at www.technicaldebt.umbc.edu). It will require evidence from a variety of environments to build a full picture of how different TD identification approaches interact, overlap, and are (or are not) synergistic. This evidence is necessary to further refine and to bring into focus the TD landscape.

8. ACKNOWLEDGMENTS

The participation of Seaman and Zazworka in this work is supported by the US National Science Foundation, award #0916699. The author Spínola would like to thank the Brazilian National Council for Scientific and Technological Development (CNPq) for their support of this research, award #201440/2011-3.

9. REFERENCES

[1] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan,R., Seaman, C., Sullivan, K., and Zazworka, N. 2010. Managing technical debt in software-reliant systems. In Proceedings of the FSE/SDP workshop on Future of Software Engineering Research (FoSER '10). ACM, New York, NY, USA, 47-52.

[2] Seaman, C., and Guo, Y. 2011. Measuring and Monitoring Technical Debt. In Advances in Computers, Vol.82,pp.25-46.

[3] Schumacher, J., Zazworka, N., Shull,F., Seaman, C., and Shaw, M. 2010. Building empirical support for automated code smell detection. In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10). ACM, New York, NY, USA, Article 8 , 10 pages..

[4] Guo, Y., Seaman, C., Zazworka, N., Shull, F. 2011. Domain-Specific Tailoring of Code Smells: An Empirical Study. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, 167-170.

[5] Izurieta, C., Vetro' A., Zazworka, N., Cai, Y., Seaman, C., Shull, F., "Organizing the Technical Debt Landscape." IEEE ACM MTD 2012 3rd International Workshop on Managing Technical Debt. In association with the 34th International Conference on Software Engineering ICSE, Zurich, Switzerland, June 2-9, 2012.

[6] Lanza, M., Marinescu, R., and Ducasse, S. 2005. Object-Oriented Metrics in Practice. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[7] Zazworka, N., Shaw, M.A., Shull, F., Seaman, C. 2011. "Investigating the Impact of Design Debt on Software Quality" In Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11). ACM, New York, NY, USA, 17-23.

[8] Zazworka, N., and Ackermann, C. 2010. CodeVizard: a tool to aid the analysis of software evolution. In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10). ACM, New York, NY, USA, Article 63 , 1 pages (Poster).

[9] Binkley, D."Source code analysis: A road map," in Future of Software Engineering, 2007, pp. 104 –119.

[10] Hovemeyer, D., and Pugh, W. 2004. Finding bugs is easy. SIGPLAN Not. 39, 12 (December 2004), 92-106.

[11] Vetro', A., Morisio, M., Torchiano, M. "An empirical validation of FindBugs issues related to defects," Evaluation & Assessment in Software Engineering (EASE 2011), 15th Annual Conference on , vol., no., pp.144-153, 11-12 April 2011, doi: 10.1049/ic.2011.0018.

[12] Nagappan, N., Ball, T., and Zeller, A. "Mining metrics to predict component failures," in Proceedings of the 28th international conference on Software engineering, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 452–461.

[13] Riaz, M., Mendes, E., and Tempero, E. 2009. "A systematic review of software maintainability prediction and metrics," in 3rd International Symposium on Empirical Software Engineering and Measurement, 2009, pp. 367-377.

[14] Zazworka, N., Seaman, C., Shull, F.: "Prioritizing Design Debt Investment Opportunities" In Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11). ACM, New York, NY, USA, 39-42.