POLITECNICO DI TORINO
Repository ISTITUZIONALE

Understanding automated and human-based technical debt identification approaches-a two-phase study

Original
Understanding automated and human-based technical debt identification approaches-a two-phase study / Spínola, Rodrigo O.; Zazworka, Nico; Vetro, Antonio; Shull, Forrest; Seaman, Carolyn. - In: JOURNAL OF THE BRAZILIAN COMPUTER SOCIETY. - ISSN 0104-6500. - STAMPA. - 25:1(2019). [10.1186/s13173-019-0087-5]

Availability:
This version is available at: 11583/2735916 since: 2023-04-27T13:59:13Z

Publisher:
SpringerOpen

Published
DOI:10.1186/s13173-019-0087-5

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)

10 May 2024

Spínola et al. Journal of the Brazilian Computer Society (2019) 25:5
https://doi.org/10.1186/s13173-019-0087-5

Journal of the
Brazilian Computer Society

RESEARCH Open Access

Understanding automated and human-based technical debt identification approaches-a two-phase study

Rodrigo O. Spínola[1*], Nico Zazworka[2], Antonio Vetro[3], Forrest Shull[4] and Carolyn Seaman[5]

Abstract

Context: The technical debt (TD) concept inspires the development of useful methods and tools that support TD identification and management. However, there is a lack of evidence on how different TD identification tools could be complementary and, also, how human-based identification compares with them.

Objective: To understand how to effectively elicit TD from humans, to investigate several types of tools for TD identification, and to understand the developers' point of view about TD indicators and items reported by tools.

Method: We asked developers to identify TD items from a real software project. We also collected the output of three tools to automatically identify TD and compared the results in terms of their locations in the source code. Then, we collected developers' opinions on the identification process through a focus group.

Results: Aggregation seems to be an appropriate way to combine TD reported by developers. The tools used cannot help in identifying many important TD types, so involving humans is necessary. Developers reported that the tools would help them to identify TD faster or more accurately and that project priorities and current development activities are important to be considered together, along with the values of principal and interest, when deciding to pay off a debt.

Conclusion: This work contributes to the TD landscape, which depicts an understanding between different TD types and how they are best discovered.

Keywords: Technical debt, Automated technical debt identification, Human-based technical debt identification

Introduction

The technical debt (TD) concept brings a new perspective on how software development tasks are discussed and managed. It 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 [7]. According to Avgeriou et al. [5], TD 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. TD presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.

It is common for a software project to incur TD during the development process since small amounts of debt can increase productivity [43]. However, its presence brings risks to the project. Effects of TD can be noticed in different stages of software development due to different types of debt. Low quality, delivery delay, low maintainability, rework, and financial loss are among the top 10 most commonly impactful effects of TD [36, 37].

Based on a familiar vocabulary from the financial domain, the TD concept facilitates discussion among practitioners and researchers and has potential to become a truly universal language for communicating technical tradeoffs. However, the TD management is more than just facilitating communication; it is comprised of a whole set of tools and techniques. These tools must

* Correspondence: rodrigo.spinola@unifacs.br
[1]Graduate Program in Systems and Computer, Salvador University, State University of Bahia, Salvador, Bahia, Brazil
Full list of author information is available at the end of the article

Springer Open
© The Author(s). 2019 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

provide, among other things, facilities for TD identification, an essential step for making TD manageable and explicit, which leads to creating a TD "list" that allows better control of the debt situation. Currently, TD identification approaches can be broadly categorized into two groups:

- *Automated tools* use a metric or analyze an artifact in a way that defines indicators of TD in software projects. Most of these tools have been developed to detect potential debt in the source code;
- *Manual approaches* identify TD items by asking developers or other stakeholders about the presence of debt or by making a manual search for debt in artifacts from the project.

Automated tools are less time consuming and their application is scalable. However, manual approaches are hypothesized to have two advantages 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) that is difficult or even impossible to glean from analysis tools.

Both approaches are relevant and have been investigated in different studies ([14, 23, 27, 38, 51, 52]; Morgenthaler [30, 34]). 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 (Schumcher et al. [14, 38]). Other studies have shown that code comment analysis can also be used to identify several types of debt such as design, defect, and requirement debt [11, 25]. More specifically, Schumacher et al. [38] showed that metric-based detection approaches perform well compared to human classification and that their use decreases the effort spent on manual code inspections.

Despite the fact that a number of studies have been conducted regarding TD identification, there is a lack of evidence on how different tools could be complementary and, also, how human-based identification compares with them. The work presented in this paper intends to investigate this area by performing a two-phase study, involving several complementary research methods, considering both automated and human-based TD identification approaches. The goal of the first phase of the study was to compare human elicitation of TD to automated TD identification. We studied three automated approaches (code smells, automated static analysis issues, and collection of code size and complexity metrics), and how their output (in terms of the location in the codebase they pointed to as having potential TD items) compares to TD that is elicited from humans. The goal of the second phase was to understand the differences in the outcomes of the two approaches to TD identification and gain an understanding of when and how they can best be used and/or combined. We also hoped, in Phase II, to glean from developers some understanding on findings that were observed from the first phase but that were hard to understand without contextual information.

The results from the first phase were previously published [51]. Here, we summarize the previously published results from Phase I, present the results from Phase II, and then draw conclusions based on the entire set of results. Thus, this paper includes new analysis of data that was collected in addition to that presented in the previous paper, in particular the focus group, plus the integration of the two data sets. Figure 1 highlights the results from Phase I that were published in [51] (in yellow), the steps from Phase I that were reanalyzed based on the whole set of data that was collected, and the steps of the work that are specific to this paper (in blue). Thus, this work has three main contributions for the area:

1. *Better understanding of how to elicit TD from humans*: We assessed a TD template [40] that can be used to capture, store, and communicate essential information about TD to support decision-making about debt payment. Our results give some insight into the dynamics of eliciting TD from a team of developers, helping us to answer such questions as: How did the developer find the TD items? Were identified TD items related to code ownership?
2. *Comparison among several types of tool support for TD identification*: Despite the fact that automated 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. Thus, 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.
3. *Developers' point of view about TD indicators and TD items reported by tools*: We asked questions regarding the connection between a TD indicator (e.g., as reported by a tool) and a TD item (i.e.,

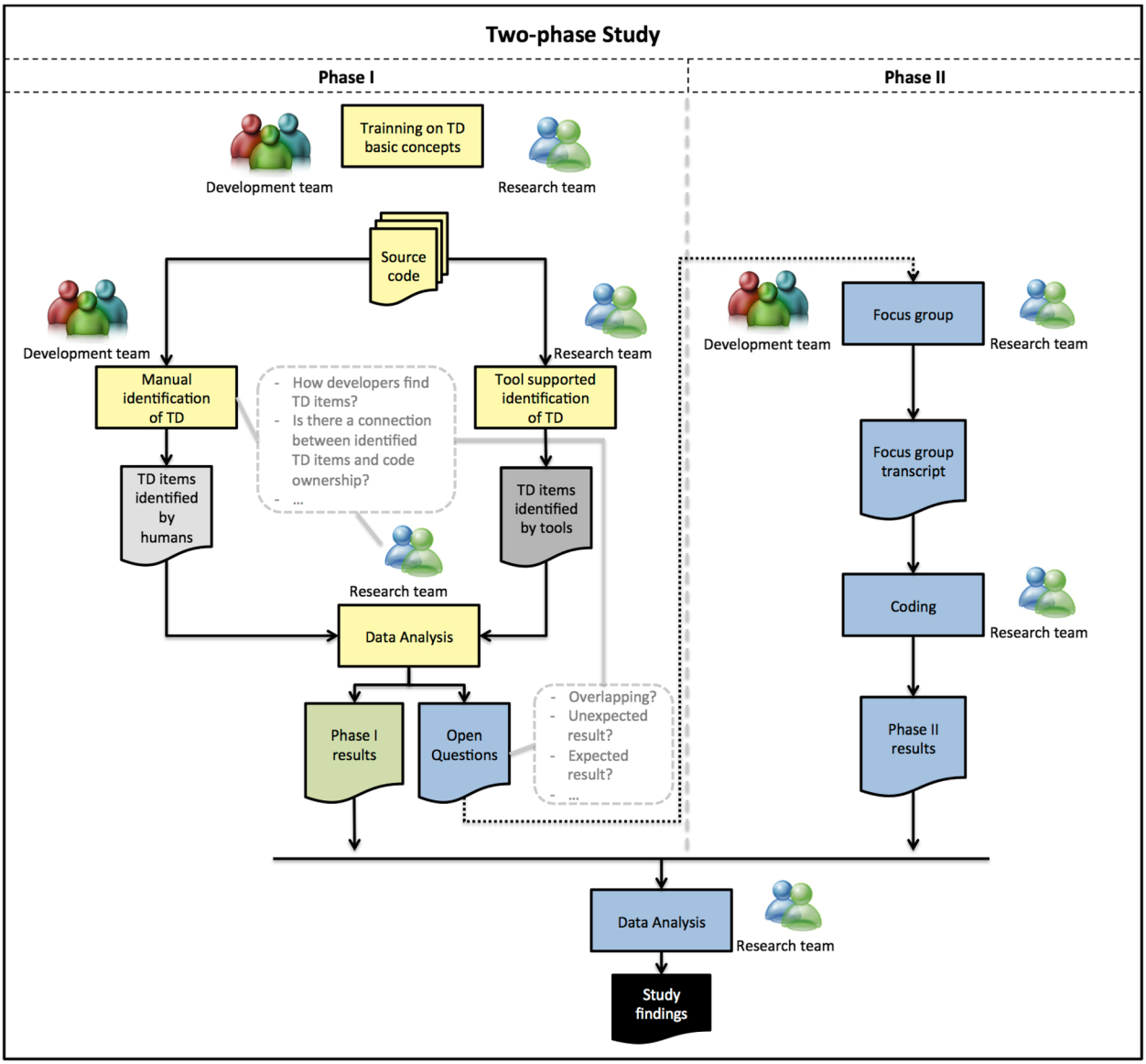

Fig. 1 Research strategy

something a developer would find worthwhile to pay off), the value of performing preventive maintenance to remove some of the issues found by the tools, and level of surprise about the results reported by the tools.

We identified that the main resources used by the development team members to identify TD items is their own knowledge on the project, augmented by an artifact closely related to their own specific development tasks. Different people, who work with different artifacts and have different roles on the project, appear to report completely different TD items. Thus, we have further shown that different stakeholders know about different debt in their project, indicating that TD identification should include a range of project team members. We also found that automated TD identification tools, to some extent, point to different potential debt than people do. The three different automated approaches did well in pointing to source code files with defect debt and also could point to some instances of design debt. On the other side, many developer-identified TD items could not have been found by the tools or metrics since the artifacts in which they were located are not included in the static code analysis. The type of debt that represented the greatest agreement between the human-elicited TD items and the automatically detected TD was defect debt.

The remainder of the paper is organized into six more sections. Section 2 describes some background concepts related to TD and also related work. Next, section 3 presents the design of the present study. Section 4 reports the results of the quantitative and qualitative analyses of the study data. Next, section 5 discusses the results of the study. After, the threats to validity are presented in section 6. Finally, section 7 presents some final remarks on this work.

Background and related work

This section begins with an overview of relevant background on technical debt management. In particular, we present a specific view of TD management that motivates our data collection instrument. Second, we review past work on TD identification approaches in particular.

Background

TD is seen as an important part of software management [43]. According to Guo et al. [15], the management of TD can center on a TD list. This list contains TD items (in the following simply referred to as items) that represent tasks that were left undone, but that run a risk of causing future problems if not completed. Each item is described by a set of properties (see Table 1).

The principal on the debt refers to the cost to eliminate the debt (i.e., the effort required to complete the task). Depending on the type of TD, this can translate into different kinds of activities, such as updating outdated documentation, refactoring code that is hard to maintain, or defining new test cases to improve their coverage. 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 [40] proposed to initially estimate the principal on a rough ordinal scale from *low* to *medium* to *high*, which allows enough understanding to contribute to 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 amount* is the potential penalty in terms of increased effort and decreased productivity that will have to be paid in the future as a result of not completing these tasks in the present, including the extra cost of paying off the debt later, as compared to earlier. Sometimes, historical data can help in estimating the interest amount. Like TD principal, data on past defects, effort, and changes can be useful. Also, like TD principal, an initial estimate of *high*, *medium*, or *low* can be used for initial prioritization decisions [40];
2. 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 [40]. The probability part of the definition of interest is necessary because TD will not always bring negative impacts on future project activities. For example, the higher the probability that the artifact that contains the debt will undergo maintenance, the higher the probability that the interest will negatively impact the project, and vice versa. Interest probability can also be estimated using, e.g., historical usage and defect data. Again, estimation can use a simple *high*, *medium*, and *low* scale until numeric estimates become necessary. 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

Table 1 The TD template [40]

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/Do not Know

consider longer periods of time, e.g., over the next year or 5 years.

The process of managing TD usually 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. This estimation (often using a scale of high/medium/low as in this example) is difficult and most often relies on subjective, but educated, guesses from developers. However, there are ways to use historical data to make this estimation more accurate. For example, interest probability, in most cases, is really the probability that a given module will be touched in future maintenance work. This probability can be estimated by calculating, based on historical data, the frequency with which the module has been touched in past releases. Once each TD item's interest and principal have been estimated, using whatever methods are available, the debt items are monitored and decisions can be made on when and what debt items should be paid or deferred.

Recently, three mapping studies addressed the management of TD as part of their scope [1, 4, 22]. Ampatzoglou et al. [4] investigated the management of TD from a financial perspective. Their goal was to understand how financial aspects are defined in the context of TD and how they relate to the underlying software engineering concepts. As a result, the authors found that the most common financial terms that are used in TD literature are principal and interest. Besides, the financial approaches that have been more frequently applied for TD management are real options [3], portfolio management [41], cost/benefit analysis [41], and value-based analysis [44].

In another secondary study in the area of TD management, Li et al. [22] presented an overview on the current state of research on TD management. The authors identified eight TD management activities (identification, measurement, prioritization, prevention, monitoring, repayment, documentation, communication) and 29 tools for TD-related activities (among the 29 tools, only four are tools dedicated to managing TD). Most tools support code and design TD management, while few tools support managing other types of TD. Also according to the authors, these TD activities received significantly different levels of attention, with TD repayment, identification, and measurement receiving the most attention and TD representation/documentation the least attention.

More recently, Alves et al. [1] identified several TD management strategies. However, only five strategies (Portfolio Approach [41], Cost-Benefit Analysis [41], Analytic Hierarchy Process [41], Calculation of TD Principal [16], and Marking of dependencies and Code Issues [9]) were cited in more than one paper. Thus, most of them still require further investigation and empirical evaluation. Further, few empirical studies have been performed in real settings. This is an indicator that, for some areas, we still do not fully understand all the costs or benefits of the proposed TD management strategies.

Finally, Rios et al. [36, 37] identified, through a tertiary study, a list of situations in which debt items can be found in software projects, and organized a map representing the state of the art of activities, strategies and tools to support the TD management. According to the authors, there are a number of gaps that need to be addressed when dealing with TD management. The existing limitations such as lack of tools/strategies to support some activities and the lack of comprehensive solutions that consider a management process for TD as a whole can make TD management difficult to perform.

Our work focuses on TD identification (discussed in the next section), which is common to all approaches to managing TD in the literature.

Technical debt identification

In this study, we are focused on the first step in TD management: TD Identification. We can use different strategies, both automated and human-based, to find TD items for each TD type. Two automated strategies that have been proposed and studied 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. In addition to code smells and issues, in this case 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.

The concept of code smells was first introduced by Fowler and Beck (as Bad Smells) [12] and describes patterns in object-oriented code that are less than ideal, e.g., that violate the rules of good object-oriented design, and should be refactored. Code smells are a type of design debt, because they are believed to slow down development, in particular changes and enhancements to affected code, when not removed early. Fowler and Beck originally suggested that developers identify code smells by performing code reviews during development, i.e., continuous refactoring. Marinescu [26] was the first to see the opportunity to automatically detect code smells by using metric-based rules that can be checked via a tool. He proposed rules (detection strategies) for identifying a set of 11 code smells. The precision and recall for Marinescu's classifiers have been studied for the most often studied code smell, god classes, and found to be high (precision 71%, recall 100%) [38].

Several studies have looked into the relationship between code smells (and related phenomena) and change

and defect proneness. Most of these have focused on one smell in particular, god classes. Most such studies have found that god classes are associated with higher change proneness and defect proneness [20, 21, 33, 38, 40], although there is some evidence that this association disappears when the number of defects and changes are normalized by size [33]. Other examples include a study by D'Ambros et al. [10] that identified that introducing a design flaw in a class is likely to generate bugs that affect the class. However, this does not hold for all design flaws in all software systems. Further, the authors also identified that there is no design flaw addition that has a consistently high correlation with bugs in all the systems. In another study on the relationship between code smells and software maintainability, Yamashita and Moonen [47] investigated the interactions among 12 code smells and analyzed how those interactions relate to maintenance problems. This study found empirical evidence that certain inter-smell relations were associated with problems during maintenance and also that some inter-smell relations manifested across coupled artifacts. More recently, Soltanifar et al. [42] have used data science and analytics techniques on software data to build defect prediction models. As result, they found that code smells are a good indicator of defect proneness of the software product. This literature is representative of the work that has established at least a correlational relationship between code smells and maintainability and has motivated the use of code smell detection as a form of TD identification.

Other types of TD identification that appear frequently in the literature are automated static analysis (ASA) and the use of code metrics. ASA refers to a set of source code analysis techniques that consist of extracting information about a program from its source or source code-based artifacts using automatic tools [6]. ASA tools look for issues in terms of violations of recommended programming practices and potential anomalies that might cause faults or might degrade some dimension of software quality (e.g., maintainability, efficiency). ASA issues are generally at a finer-grained level (e.g. at the source code line) than code smells (e.g. at the method or class level), and previous work indicates that there is little overlap between the two automated approaches when used to identify TD [52]. 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 [17, 45].

Gat and Heintz [13] identified TD in a customer system using both dynamic (i.e., unit testing and code coverage) and static (computing rule conformance, code complexity, duplication of code, design properties) program analysis techniques. Nugroho et al. [32] also used a combination of static analysis and code metrics to identify TD. They assigned levels of different metrics and indicators to risk categories to quantify the amount of interest owed in the form of estimated maintainability cost. A CAST report [8] focuses on the density of static analysis issues on security, performance, robustness, and changeability of the code. The authors built a pricing model assuming that only a percentage of the issues are actually being fixed. Sonar (http://www.sonarsource.org/) is an open source toolkit that has gained in popularity. It also uses static measurements against various source code metrics and attributes to assess the level of TD in a code base.

In a recent mapping study, Alves et al. [1] investigated the types of TD, how TD items can be identified through indicators of their existence in projects, and the strategies that have been developed for the management of this debt. Moreover, they assessed the degree of maturity of the existing proposals through an analysis of the empirical evaluations that have been carried out. TD indicators allow the discovery of TD items when analyzing the different artifacts created during the development of a software project. The authors observed that some types, such as design, already have a fair number of indicators and, on the other hand, indicators were not identified for some types of debt (process, infrastructure, people, and usability debt). In total, the study mapped 45 different TD indicators, and the results show that the most cited and analyzed TD indicator is code smell, and the type of code smell that has been most investigated is god class. According to the authors of the study, an explanation for this is that god classes are conceptually easy to understand and are up to 13 times more likely to be affected by defects and up to seven times more change-prone than their non-smelly counterparts, which makes them a good candidate when starting to detect TD from the source code.

Few studies have explored the task of manually identifying TD. Potdar and Shihab [34] manually analyzed code comments to identify text patterns and TD items. They read more than 101 K code comments and showed that 2.4–31.0% of the files in a project contain self-admitted TD. The most used text patterns that indicated the presence of TD were (i) "is there a problem" with 36 instances, (ii) "hack" with 17 instances, and (iii) "fixme" with 761 instances. Next, Maldonado and Shihab [25] evolved the work of Potdar and Shihab [34] proposing four simple filtering heuristics to eliminate comments that are not likely to contain technical debt. For that, they read 33 K code comments from source code of five open source projects. According to the authors, the most common type of self-admitted TD is design debt (between 42% and 84% of the classified comments). Related to this work, but performed in an automatic way,

Farias et al. [11] proposed the CVM-TD. CVM-TD is a contextualized structure of terms, implemented in a tool named eXcomment, which focuses on using word classes and code tags to provide a TD vocabulary, aiming to support the detection of different types of debt through code comment analysis.

Despite the fact that a number of studies have been conducted regarding TD identification, there is a lack of evidence on how different tools could be complementary and, also, how human-based identification compares with them. This study intends to shed some light on this area.

Research design

In this section, we describe the methods and techniques we used to investigate our research questions. Both phases took place in the same industrial context, which we describe in the next subsection, and used the same goals, described in the "Goal and research" section. Then, we describe the data collection and analysis activities in Phase I and Phase II, respectively.

Figure 1 illustrates our research strategy. The two phases of the study are complementary and contribute in different ways to the study findings. While in the first phase, the goal was to understand the human elicitation of TD and compare it to automated TD identification (see "Phase I" section), in the second phase the goal was to understand those differences, and gain an understanding of when each set of approaches is appropriate and how they can best be combined (see "Phase II" Section). For instance, at the end of the first phase, we had several open questions regarding how developers found TD items, if there is a connection between identified TD items and code ownership, if the results provided by tools were expected or unexpected, and others. Phase II was designed to answer as many of those open questions as possible. At the end, the results from both phases were analyzed together.

Context

The two-phase study was conducted at KaliSoftware, a small software development company located in Rio de Janeiro, Brazil, that developed primarily web applications written in Java and based on the MVC framework. The project we studied consisted of a small application of 25 K non-commented lines of code. It was a database-driven web application for the sea transportation domain. It had undergone a full product lifecycle (elicitation, design, implementation, deployment, and maintenance). The project team was composed of five professionals: two developers, one maintainer, one tester, and one project manager who also played the role of the requirements analyst. Table 2 shows the level of experience of the participants in terms of years working in the software development area.

Table 2 Characterization of the participants

Participant	Level of experience in years
Project Manager	10
Developer	2
Tester	2
Developer	1.5
Maintainer	1.5

Goal and research questions

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

- RQ1—Do the TD identification tools find the TD items that were reported by the members of the development team?
- RQ2—How much overlap is there between the TD items reported by different members of the development team?
- RQ3—Is the TD item template a feasible and effective tool for eliciting TD items from members of development teams?

Data collection and analysis

Below, we outline the data collection and analysis techniques used in Phase I and Phase II. More details about the Phase I research design can be found in [51].

Phase I

As can be seen in Fig. 1, one of authors trained the development team in TD basic concepts via Skype. This training included an opportunity for Q&A. As the development team's native language was Portuguese, all the material was prepared and then translated to Portuguese before presentation. Initially, we explained some basic concepts related to TD. The definition of TD we used was [29]: *Technical Debt refers to delayed technical work that is incurred when technical short cuts are taken, but that creates a technical context in which the same work will cost more to do later than it would cost to do now*. To explain the attributes of a TD item (principal, interest, interest probability) and the TD template used to report a TD item, we used the concepts and definitions presented in the "Background" section of this work.

During the training, we wanted to give some examples of TD items to make sure that the team members understood the concept well. However, we also wanted to

avoid biasing them towards, later, reporting only TD items that were similar to the examples we presented. Our solution was to present only non-software-related TD items as examples (for instance, "TD items" related to house or car repair). The types of TD presented during the training (design, documentation, defect, and testing) were defined and described as follows:

- Design debt: any kind of anomaly or imperfection that can be identified by examining source code that leads to decreased maintainability if not remedied;
- Testing debt: refers to issues found in testing activities that can affect the quality of those activities. Examples of this type of debt are tests that were planned but not executed;
- Documentation debt: refers to the problems found in software project documentation and can be identified by looking for missing, inadequate, or incomplete documentation of any type;
- Defect debt: known defects that are not yet fixed.

We indicated that those types are just examples of debt that could affect software projects. Thus, the participants were free to consider any other type of debt during the study. 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 automatic TD identification, 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. These tools were chosen primarily because of our previous experience in using them in studies of technical debt [14, 18, 38, 45, 48–50, 52], as well as their ability to find common smells and issues. They were, at the time, the best tools available for our purposes. 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), and a priority from 1 (highest) to 3 (lowest). We also selected and computed for each file the following structural metrics: Lines of Code, McCabe's Cyclomatic Complexity, Density of Comments, and Sum of Maximum Nesting of all Methods in a Class. Lines of Code and McCabe's Cyclomatic Complexity are widely used in the literature on defect and maintainability prediction (e.g., [31, 35]). 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. Application of all these tools, metrics, and indicators resulted in a very large amount of data and an enormous number of potential TD items. Therefore, in order to present relevant results, we devised and applied some filtering strategies to the tool outputs. Thus, the results presented in the "Results" section do not include all of the metrics and indicators mentioned here. The filtering mechanisms are also explained in the "Results" section.

In parallel to automatic identification, the development team (project manager, developers, maintainers, and testers) was 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 below) and provide information about the difficulty of documenting debt items (questions 2 to 5 below). 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 in Table 1 to answer question 1.

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 may be differently perceived in life critical software applications. Other known TD types are design debt, documentation debt, and testing debt, all of which are defined above as they were defined for the respondents. Other types of debt have been identified and proposed in the literature [2], but at the time this study was performed, these were the

types primarily referenced, and so were the ones used in the survey. However, respondents were also free to invent other types of debt when these four did not cover a particular situation, and some of the respondents did that.
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.
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 field indicates, if the debt is code-related, the file, class, or component that needs to be modified to eliminate the debt. For other types of debt, the relevant artifact or document is specified. This information is important to understand impact on the product, relationships between items, and ripple effects in source code when repaying the debt. For our study, it also allowed us to match up the elicited TD items with the output of the automated tools.

After manual and automatic identification of TD, we performed data analysis with a comparison of the two sets of results. More details and preliminary results from Phase I were presented at the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE 2013) [51].

Phase II

Phase II of the study involved a focus group with developers to help gain a deeper understanding of the findings from Phase I. After reviewing the Phase I results, we developed a structure for the focus group based on questions about specific findings. First, questions CQ1 and CQ2 were used to gain an overall perception of how the project team members worked to identify TD items on the project:

- CQ1—How did you (developer) find the TD items?
- CQ2—Were identified TD items related to code ownership?

Next, we chose several specific instances in the results where the same location in the source code was found to have a high level of some TD indicator (as evidenced by one of the automated tools) and was indicated in a TD item reported by a developer. Question CQ3 helped us to determine if these instances of overlapping were meaningful:

- CQ3—Do you (developer) think there is a connection between this indicator and the TD item?

Questions CQ4 and CQ5 were asked about specific instances in the results where issues were identified by automated approaches in a particular file, but no TD items in those files were present on anyone's reported TD list:

- CQ4—Does this file have any TD?
- CQ5—Would it be worthwhile to do some preventive maintenance to remove some of the issues found by the tools?

Finally, questions CQ6 and CQ7 were asked to shed light on cases where a TD item was flagged by developers in a particular location but the tool-identified indicators did not signal a problem:

- CQ6—Are you (developer) surprised that the automated tools did not find any problems with this file?
- CQ7—Would you have expected to see more problems found by the tools? Why or why not?

Due to the distance and language, this phase also involved a company contact, who was part of the research team (first author) and was in charge of data collection, conducted the focus group, and performed all communication between the project team and the research team. All communication with the project team was in Portuguese, while the research team communicated in English. We conducted the focus group meeting over Skype, audio-recorded the entire session, and transcribed it for analysis. The transcription of the focus group was then translated to English. The next step was to code [39] the transcription in order to find evidence to help the authors understand the differences found in the results from Phase I. The coding was also performed by the first author, who generated and followed the coding schema defined in Table 3.

We coded the focus group data in vivo, using codes that emerged from the data and thus were, in some sense, related to the formulated questions. For this, we used a combination of pen, paper, and Microsoft Excel. We formulated findings by synthesizing the coded segments pertaining to categories that had substantial data weight. That is, the findings were grounded in the data rather than formulated ahead of time and validated through the data. At the end, the results of both Phase I and Phase II were put together and we drew conclusions based on the entire set of results.

Table 3 Coding schema

Code	Description
TD Item Description	Question and answer that helps explain a specific TD item.
Specific Question	A question about a specific TD item.
Generic Question	A question that is not specific for any TD item.
New Question	A question that was not planned before the meeting.
Confirmation Question	A question asking for confirmation from all participants.
Metric Correlation	Answers related to the correlation between TD items and metrics.
Code Smells Correlation	Answers related to the correlation between TD items and code smells.
Find bug Correlation	Answers related to the correlation between TD items and FindBugs issues.
Note	Moderator notes about participant actions.
Answer	Answers to new or generic questions.
Technical Debt Overlap	Answers addressing the lack of overlap between the identified TD Items.
Technical Debt Identification	Answers about the manual TD identification process.
Indicators and Technical Debt Identification	Answers about the correlation between technical debt and metrics, code smells, and FindBugs issues.
Benefits of TD Management	Comments about the benefits of TD management.
Benefits of TD Documentation	Comments about the benefits of TD identification.
TD Identification without code reading	Comments about TD identification without reading the source code.
TD Identification with code reading	Comments about TD identification by reading the source code.
Indicators and Problems in the Project	Comments about the correlation between problems in the source code and metrics, code smells, and FindBugs issues.
Code Inspection Feasibility	Comments about the feasibility of inspecting the source code to look for TD items.

Results

The tools we used generated a large number of issues pointing to potential TD, with indicators occurring in nearly every file in the code base. Therefore, we had to pre-filter the results in some way to allow a meaningful comparison with the manually elicited TD items. 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 each of the source code metrics, 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 realize that this filtering approach somewhat biases our results by only showing the automated tool results that performed best in terms of matching up with manually reported TD items. 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. Thus, we can reject the notion that none of the automated indicators (metrics, ASA issues, and code smells) are good at finding TD, but we are still not at the point where we understand which indicators generally predict TD best, or under which circumstances they predict best, especially given the high number of false positives typically included in the results from such tools.

Results in Fig. 2 show how the 21 TD items (presented in Appendix) identified by the software team, each represented as a colored box, were distributed over project roles and types of debt. As the legend indicates, each box has three faces, corresponding to principal (front), interest probability (right side), and interest amount (top). 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. 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.

Figure 3 shows the results of automated identification approaches (FindBugs, Code Smells, Metrics) compared to the items reported by the development team. Each box in Fig. 3 corresponds to one of the

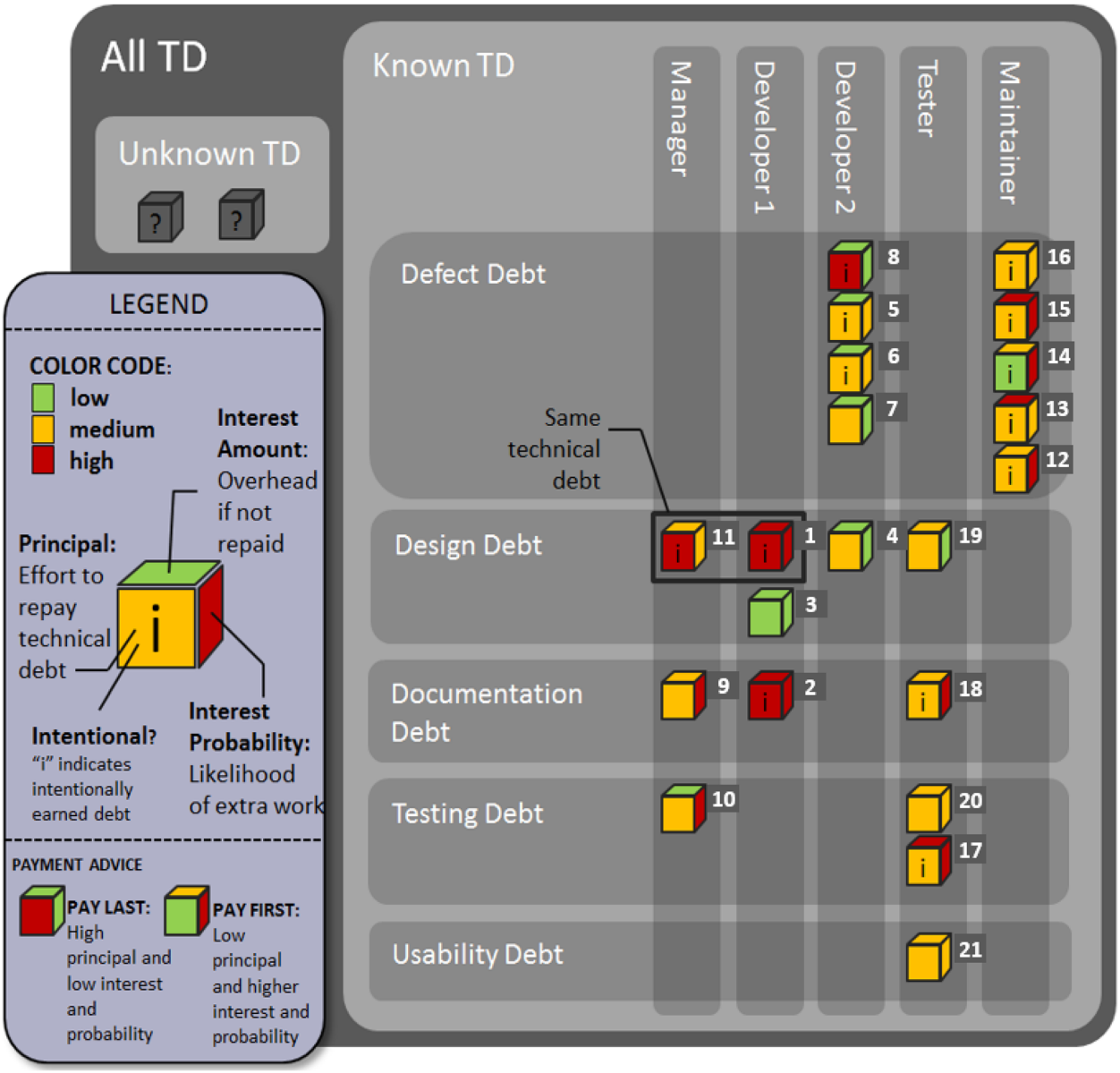

Fig. 2 Results of the human elicitation of TD items (adapted from [51])

boxes (elicited TD items) shown in Fig. 2. An "s" on the front face of a box shows that the TD item was located in the source code by the subject who reported it (as opposed to some other artifact besides code). The overlapping shaded areas in Fig. 3 depict the overlaps between the TD items reported by the human subjects (shown in Fig. 2), and the TD items found by the top three automated indicators (MAX Nesting, FindBugs P1, and Intensive Coupling). We can observe that there are overlaps in only two types of debt, defect, and design debt. For example, the shaded area labeled "Defect Debt" in Fig. 3 contains the nine 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 Fig. 2. Figure 3 shows that, of these nine defect debt items, seven were also found by all three automated indicators. A possible explanation for this concentration of overlaps on only those two types of debt is that they are closely related to the code (and then, could be reached by the tools) and there are participants whose roles are also directly related to code tasks (developers, tester, and maintainer). The other identified instances of debt in this study (documentation,

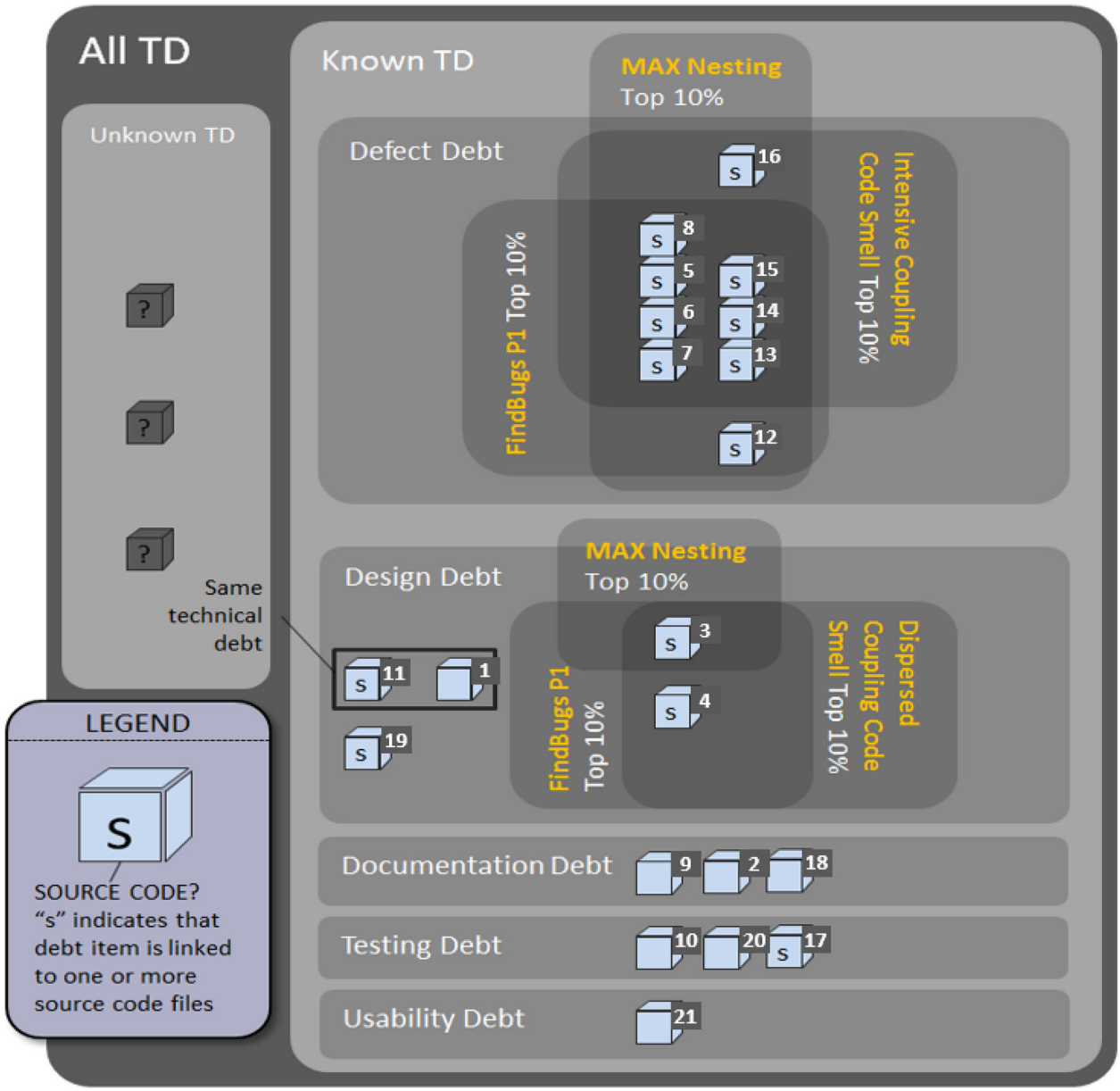

Fig. 3 Results of the tools compared to human elicitation (adapted from [51])

testing and usability) could not be identified by the used tools.

We synthesize these results below, along with insights gained from the Phase II focus group, to address our research questions.

Do the TD identification tools find the TD items that were reported by the members of the development team? (RQ1)

We observe in Fig. 3 that the three top automated approaches do about equally as well as manual elicitation in identifying defect debt. In fact, these three 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) identified 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 item 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 others were related to other development artifacts (e.g., requirements documents and test plans).

In order to have a better understanding of how developers proceeded to find TD and what they think about the indicators considered by the tools, our first focus group question to the development team was *How did you (developer) find the TD items?*

The main source for identifying TD items was clearly the participants' own knowledge on the project, without external aids (i.e., written sources). The following excerpts were typical: "Developer 1: I already had some knowledge of these debts in my mind," "Developer 2: I reported those debts that I remembered," "Maintainer: I have knowledge of what was done differently from what the customer wanted," and "Tester: I used my own experience with testing activities." These types of responses were reported regardless of the role assumed by the participant in the development process, i.e., by developers, maintainers, and testers.

In addition, the data showed that each participant used a "resource" associated with their responsibility in the development process to support the identification of TD items. Both developers reported analyzing or searching the source code for TD items. The maintainer reported analyzing tickets, and the tester relied on information from "testing activities when several problems are identified."

Thus, we identified that the main resources used by the development team members to identify TD items is their own knowledge on the project, augmented by an artifact closely related to their own specific development tasks. Specifically, developers relied on source code, the maintainer on tickets, and the tester on results of test activities.

This qualitative data from the focus group helps explain in part why there was so little overlap between the developer-elicited TD items and the items revealed by automatic tools and thus contributes to addressing RQ1. Since part of the development team did not work directly with the source code, and since they used resources (knowledge, experience, artifacts) closely related to their job tasks to identify debt in the project, it follows that much of the debt reported by the development team would not be associated with the code and thus could not be identified by the tools used in this study.

To further understand the difference between developer-reported and tool-elicited TD, we asked developers about the value of the automated TD indicators we used (metrics, code smells and FindBug issues) . The answer was overall positive. When asked, "Do you think these tools and metrics would help you to identify TD faster or more accurately?," all participants responded in the affirmative, using phrases such as "useful" and "valid." However, interestingly, they also used terms that indicated that the use of tools is not a complete solution, e.g., "a starting point to identify TD, " "they can indicate a path to follow," and "for code, they are good indicators."

The focus group data also indicates that metrics, code smells, and FindBugs issues, beyond being valid indicators of debt, are also considered good indicators of general quality problems in the source code of the project. For example, when asked "If you didn't have time to report the TD items, do you think that using these metrics (that point to some larger classes) could help you to identify possible problems in the project?," the maintainer answered that "Looking at a project as a whole, yes." Participants also answered positively to a similar question about code smells, indicating that the smells, besides being useful as pointers to debt, are also useful as indicators of quality problems in general and could be used to guide quality improvement efforts.

As a very specific example, there were two distinct classes that we asked the participants to comment on, because the developers had reported TD items in them, but the automated indicators did not reveal any issues. We asked the participants if they were surprised by this result, and the two developers answered no, because the major problems with these two classes had to do with performance, and so would not be expected to be highlighted by the tools.

As another example, when asked about a class in which the automated tools indicated the presence of debt, but for which the participants did not report any TD items (with the prompt "Does this file have any TD?"), the participants answered that they did not know but that the problems reported by the indicators actually exist in the file: "I don't know. But this class is really quite complex since it involves several use cases and has file handling …. Besides, there is a lot of IF conditions that must be analyzed and, yes, there is a high probability of errors in this class." Again, participants felt that metrics, code smells and FindBugs issues are good

indicators of problems in the system classes, whether or not those problems could be classified as "debt."

When asked about the usefulness of the automated indicators to help plan maintenance activities, the answers given by one of the developers also reinforce this view about automated indicators and source code problems in general:

- Q: "Would it be worthwhile to do some preventive maintenance to remove some of the issues found by the tools?"
- A: "Yes. For example, I would have some trouble understanding this even though I developed it myself. It would be good to refactor this class, for example, to decrease the size of the methods … in order to facilitate future maintenance activities."
- Q: "If you had only these data to indicate potential problems in the project, would it be useful?"
- A: "Yes. For example, I had an idea because I developed this class, but the other developers would not have any idea about this class because they didn't work on it. Thus, they would not have an idea if there is or is not a problem there."
- Q: "In addition, having knowledge about code smells, could it be used to improve the quality of the project?"
- A: "Yes. For example, large classes that have been growing and we do not have time to refactor the project although we would like to. More specifically, there is a very large class that we need to adjust but we do not have time right now."

Therefore, indicators are useful to support the identification of TD items and are considered a good starting point for this activity. On the other hand, participants of the study also indicated that they are not enough:

- "Developer 1: but these indicators are more associated with refactoring to facilitate future maintenance of the project";
- "Developer 1: Therefore, the tools are not enough (for example, we reported items that were not identified by indicators)";
- "Developer 2: After that, you need something more to find more specific TD items like the ones related to requirements";
- "Maintainer: However, the indicators are not enough for some technical debt related to functional aspects, standardization or even system requirements."
- "Maintainer: So, the indicators are valid but not enough."

Summarizing, these results address RQ1 in the following way:

- Code smells, FindBug issues, and code metrics can support the identification of defect and design debt in this project, but not other types of debt that were reported by developers.
- These automated indicators are considered good indicators of technical debt in the source code of the project, at least as a starting point.
- They are also considered useful as indicators of general quality problems and thus could be used to guide quality improvement efforts during maintenance.

However, the indicators we used are not sufficient because they are not able to identify other types of technical debt items that are not directly associated to the code, for example, problems with requirements, business rules that have not been fully implemented, and usability issues. It should also be noted that the focus group participants were looking at the results from our pre-filtering of the automated indicators, so they were only looking at the top-rated indicator results. The tools returned flagged many more potential TD instances than were examined in the focus group. Thus, we can reject the notion that none of the automated indicators is useful for TD identification, but we cannot make claims about the usefulness of the overall approach of automatic TD identification, nor can we comment on whether the same three indicators we found to be the best performers would also perform well in other contexts.

How much overlap is there between the TD items reported by different members of the development team? (RQ2)

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 team member. By analyzing the focus group transcript, it is possible to identify what can lead to non-overlapping TD items when they were being identified.

As mentioned before, when we asked the developer team how they found TD items, they reported that the main resource used was their own knowledge on the project. In addition to their own knowledge, what they used to support the description of the TD items was an artifact related to the specific tasks each team member was engaged in. Because each participant worked on different parts of the project and each TD item was reported based on their memories and experiences on the project, we can assume that the reported TD items

would be associated with their daily activities and would be different from each other. In this project, two of the participants worked as developers, one as maintainer and one as tester. Thus, in addition to working on specific parts of the product, each participant had a different perspective when analyzing and reporting TD items. All of this contributed to reduce the overlap when identifying TD items.

In addition, when asked directly about the reasons that led them to report different technical debt items, participants reported that "Everyone was focused on a different part of the project;" thus, "Each developer reported based on what they worked on the project." Lastly, participants indicated that there was no contact between them while performing the TD identification activity ("There was no contact between us in answer to the questionnaire.").

These findings provide initial evidence that indicates that:

- If the participants responsible for identifying TD items work in different parts of the project, the items reported by each participant tend to be associated with the parts of the project in which that participant works;
- Identified TD items are related to code/artifact ownership.

Thus, to increase coverage when manually identifying TD items, we can infer that it is important that the people identifying the TD are ones who work on different parts of the project and/or have different roles in the development process.

Is the TD item template a feasible and effective tool for eliciting TD items from members of development teams? (RQ3)

As reported in [51], the five study subjects reported that it took between 50 min and 2 h to identify and document the TD items (average of 19 min 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. This research question was only addressed in Phase I, i.e., we did not ask questions about the use of the template during the focus group.

Discussion

The findings presented in the previous section have several implications. We found that automated TD identification tools (to some extent) point to different potential debt than people do. Further, many developer-identified TD items could not have been found by the tools or metrics since the artifacts in which they were located are not included in the static code analysis. We did not limit our participants to reporting only TD items that could be found by the tools we used, which in some sense might have been a "fairer" comparison. But this type of "fairness" was not our goal. We were interested in finding which types of TD developers felt were most important and what role tools could plan in finding those types of TD. The fact that developers reported some code-related debt, as well as TD items that were not code-related, suggests that a focus on source code as the single source of TD is too narrow. Future studies might consider including or proposing tools for other kinds of development artifacts affected by TD. At the time of this study, few such tools were known to us, but several have since been proposed (see, e.g., http://www.hejf.de/).

Our findings also indicate that different people, who work with different artifacts and have different roles on the project, appear to report completely different TD items. This result would have been stronger if we had had multiple participants in the same role, to see if people in the same roles would have reported the same TD items or would have agreed on the importance of different TD items (a limitation of this study that could be addressed in future work). However, this finding is still interesting and implies that, if a project wants a comprehensive view of all the TD present, identification must involve people representing all different development roles, as well as automated tools. This also has implications for prioritization and decision-making about TD. It is likely that different development roles have very different ideas about which TD items are most important to address in the near term, making consensus difficult to reach. This makes it important to have objective criteria to help in prioritizing task and in making decisions about what debt to pay off and when. Estimation of principal and interest could be part of such an approach, but we found that developers found it especially difficult to make these estimations, even at a very coarse-grained (high/medium/low) level. Thus, further work in quantifying TD is crucial if TD decision-making is to be facilitated.

On the other hand, it may be that most projects do not need to consider a comprehensive view of all types of TD. In this study, we asked the development team to consider all types of debt, but in many cases, a project has specific types of debt they want to focus on.

Projects often have identifiable "pain points" [24] that clearly imply specific types of TD that are most relevant to that project. In such cases, our findings imply that involving only specific tools, or specific people, would be sufficient to identify the relevant debt.

These observations about the relationship between code (or artifact) ownership and identifying TD also imply another role for automated detection approaches. In cases where a wide cross-section of developers is not available for TD identification tasks, automated tools might serve as a guide for those developers who are engaged in TD detection, by helping them identify places in the code (with which they might not be familiar) that should be examined for potential TD.

The type of debt that represented the greatest agreement between the human-elicited TD items and the automatically detected TD was defect debt. Remember that defect debt represents known defects that have been allowed to persist beyond when they could have been fixed. In this study, much of the defect debt reported was actually not-fully-implemented requirements, i.e., implementations that do not fully meet the requirements and thus are logged as defects. The tools we used to analyze the source code were not intended to find defects, per se, but are meant, in theory, to find anomalies in the code that could, in the future, affect maintainability and lead to changes that introduce defects. Thus, at some level, it is surprising that automated source code analysis pointed to code that had latent defects and unfinished implementations of requirements. One possible explanation is that such pieces of code represent "quick and dirty" implementations, where time constraints led to (at the same time) code that is poorly structured (thus triggering ASA issues, code smells, and high complexity scores) and incomplete (thus leading developers to tag it as defect debt). In this way, the issues found by the automated tools could be said to coexist with defects, but not to cause them. Instead, the issues and defects are both caused by a third factor, time constraints. Another possible explanation is that poorly structured code does, in fact, cause defects in a more direct way. Thus, the poor structure (as highlighted by the automated tools) would have preceded the defects, which were introduced during changes that were made more difficult and error-prone by the low quality of the code.

Although not a subject, we explored in the focus group, some interesting observations can be made about the participants' estimations of principal and interest, represented by the colors of the TD items in Fig. 3. This color coding indicates that 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.

The color coding in Fig. 3 also 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 [49].

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. Our goal in this work (as with all case studies) was not generalizable results, but insights derived from a deep dive of a single case on how two classes of identification approaches relate.

An important construct validity threat derives from the fact that we do not know if either the human-elicited TD items or the TD indicated by the tools we applied, constituted "real" technical debt, i.e., that would lead to future maintenance costs. Thus, we had no "ground truth" against which different types of TD identification can usefully be compared. 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, our results are limited to comparisons between the two modes of TD identification, but not evaluation of either as being more "correct."

Another assumption that we have made that may also lead to a construct validity 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, 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.

An internal validity threat (i.e., one that threatens the validity of the relationships we believe we have

discovered) arises from the dual role of our company contact (first author) as researcher and member of the organization. Dr. Spínola was a member of the research team, but was also at the time a co-owner of Kali Software, principally engaged in customer service. Although he did not have direct involvement in operations (i.e., in software development), there may have been some bias in the opinions the participants chose to share with him.

In addition, we reported in the "Results" section that we had to pre-filter the issues from automated tools by applying a ranking strategy: for each indicator, we examined the top 10% of the sorted list and determined how many source files in that 10% corresponded to TD items reported by developers, selecting then the top three performing indicators (i.e., FindBugs P1 issues, the MAX nesting metric, and Intensive Coupling code smell). This filtering mechanism gives rise to an external validity threat, since we cannot claim that the top performing indicators used in this study would also be the best ones in any given case. Thus, it made sense to look first at the top performing indicators.

Finally, the focus group was performed in Portuguese, transcribed in Portuguese and, after that, translated to English. So, language constraints and cultural idioms may have had an influence in the understanding of the statements, thus also posing an internal validity threat. To deal with this, the first author is Brazilian and worked carefully to avoid or minimize any bias or misunderstanding in the translation process from Portuguese to English and during the execution of the focus group. A related reliability threat is the fact that coding was performed by only one author, with oversight but not second coding, from another of the authors.

Conclusions

The TD concept 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 the results of a two-phase study, in which we compared the results of applying manual and automated techniques for identifying TD and gained further insight into the similarities and differences through a focus group with the development team members. We 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, if the goal is to gain a comprehensive understanding of all types of debt in a project. 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 some instances of design debt.

A limitation of the performed study is related to the development team involved in the study. Given the fact that multiple developers were not involved in the manual identification of TD on the same part of the project, it is not clear how reliable the reported results of a single developer can be. Aggregation sounds like a reasonable approach but some intersection of results from different developers working on the same part of the project may be necessary to improve the accuracy of the results. Besides, the study pointed out that developers tend to report different results than tools but it was not possible to investigate to which extent this behavior is due to the fact that the study did not use tools that find TD items in non-code artifacts, for example in requirements.

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)?
- How can TD principal and interest be better and more objectively quantified in order to facilitate decisions and prioritization of debt items, in the face of potentially conflicting priorities among stakeholders?

A more comprehensive study (using a larger set of tools, such the ones presented in [9, 19, 28, 46]) would be interesting in order to better grasp the overlap and differences between manual and automatic identification approaches. Thus, we also encourage practitioners to use the proposed template in their projects and to share results and experiences. 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.

Appendix

Table 4 TD items identified by the development team

ID	Type	Location	Description	Principal	Interest	Interest Probability	Intentional?
1	Design	Lingada.java, ItemGuarnicao.java, FuncionarioAccess.java, ItemTreinamento.java, RestricaoFuncionario.java	These classes have the attribute Employer Name, but this data is stored in an external database. Thus, it is not possible to access the employer information as we can do with the other information stored in the system database.	High	High	High	Yes
2	Documentation	MALO_SGI Especificacao_Requisitos.doc	The Module of Allocation does not have the requirements specification document.	High	High	High	Yes
3	Design	AlocacaoController.java alocacar(...) method	This method checks the data about employers in Access database and, after, sort the data according to a set of criteria. However, the method is currently very large and need to refactored in small methods.	Low	Low	Low	No
4	Design	customer inclusion (file ClienteController.java) company inclusion (file EmpresaController.java)	By entering a zip code in the inclusion of a company or customer, the system returns some districts as cities, but those districts do not exist on IBGE's table. This fact can cause errors when an invoice is created.	Medium	Low	Low	No
5	Design	Files FaturaController.java and NFe.java	I need to make a verification with the activity name when I need to identify a service type or bill. This information is fixed in the code, and can bring errors when some update is performed or the data in database have incorrect names.	Medium	Low	Medium	Yes
6	Design	File DespesaController.java	There is a need to locate the cost codes according to their numbering. However, this information is fixed in the code and any update or error on it can bring bad side effects to the system.	Medium	Low	Medium	Yes
7	Design	File AutoTracJob.java	High consumption of memory causing stack overflow. This method needs to run every 2 min, but because of this problem it was only running once a day to keep the data updated.	Medium	Low	Low	No
8	Design	File Parser.java	In the import of "Machine Daily," it is necessary to associate the "Machine Daily 1" to "Machine Daily 2" (this mapping is one-to-one), but sometimes, we can have more than one Machine Daily 2. In this case, the system can only store the "machine daily 2" that arrived first, the second "machine daily 2" is lost.	High	Low	Low	Yes
9	Documentation	Use cases of the Module of Billing	Requests for change by the customer have been treated, but the description of use cases and their business rules have not been updated.	Medium	Medium	High	No
10	Test	Several functionalities	Deliveries of system functionalities for acceptance testing before system testing, due to customer pressure who wants to see the changes that have been addressed (weekly deploys).	Medium	Low	High	No
11	Design	Many files of Tranship System and many files of an external system that is integrated to Tranship.	Each system has its own database (both MySQL), but they have some entities in common, so some data are repeated. Besides, sometimes one table is more complete in one of the systems. The integration work was only performed for the employees table.	High	High	High	Yes

Table 4 TD items identified by the development team *(Continued)*

ID	Type	Location	Description	Principal	Interest	Interest Probability	Intentional?
12	Defect	cadatrarContratoApoioMaritimo method of Module of Operations.	The contract can have a symbol or "adjustment percentual." These points were not considered on the Module of Operations.	Medium	Medium	High	Yes
13	Defect	excluirEmbarcacao method of Module Administrative	A ship, even excluded, can be part of the division of the cost center associated with it.	Medium	High	Medium	Yes
14	Defect	gerarRelatorioEstado method of Module of Operations	A query to the database does not return old data that are still valid when the search date is the current date.	Low	Medium	High	Yes
15	Defect	calcularValorServico method of Module of Operations	The calculation of the Maritime Support Services value is incorrect due to a change occurred in how this calculation is made.	Medium	High	High	Yes
16	Defect	gerarRelatorioLocalizacao method of Module of Operations	This functionality is almost all incomplete.	Medium	Medium	Medium	Yes
17	Test	Module of Operations	It is a very complex module because there are dependencies between modules in the system, and this requires careful attention during testing execution.	Medium	High	High	Yes
18	Documentation	Requirements specification and use case description for the Administrative and Alocation Modules	The documentation must be always updated and the requirements gathering must be in accordance with customer needs. Frequent changes in these modules have been causing a lot of rework.	Medium	Medium	High	Yes
19	Design	Cliente class	In the Cliente form, sometimes due to lack of time when implementing any specific method in the system, the system loses quality.	Medium	Medium	Low	No
20	Test	Forms	The lack of test plans can bring problems after system deployment.	Medium	Medium	Medium	No
21	Usability	All project forms	The system should have a user interface "standard."	Medium	Medium	Medium	No

Acknowledgements
Not applicable.

Authors' contributions
ROS and NZ planned and executed the study, and analyzed and interpreted its results. AV analyzed and interpreted the study results. FS and CS planned the study and validated the obtained results. ROS, NZ, and CS were the major contributors in writing the manuscript. All authors read and approved the final manuscript.

Authors' information
Rodrigo Oliveira Spínola is a Professor of Computer Science at the Salvador University where he leads the Technical Debt Research Team, Visiting Professor at State University of Bahia, and Senior Scientist at Fraunhofer Project Center for Systems and Software Engineering at UFBA. Prof. Spínola holds a D.Sc. and a M.Sc. in System Engineering and Computer Science from the Federal University of Rio de Janeiro. From 2011 to 2012, he was a visiting researcher at the University of Maryland Baltimore County and the Fraunhofer Center for Experimental Software Engineering. His main research interests are on technical debt and empirical software engineering.
Nico Zazworka is a software project manager at Elsevier Information Systems. His main research interest is to investigate the applicability and performance of software development processes and to identify when and why developers stray away from them and how tools can support developers better. He has previously worked at Fraunhofer to study this question. Besides, he is interested in research that aims at identifying and visualizing code smells, code decay, and other technical debt items through software repositories. He received his Ph.D. and M.S. in Computer Science from University of Maryland, College Park in 2010 and 2009, respectively, and his B.S. in Computer Science from University of Applied Sciences in Mannheim, Germany.
Antonio Vetrò has been Director of Research at the Nexa Center for Internet and Society at Politecnico di Torino (Italy) from November 2015 to August 2017. He's currently Research Programs Manager at ORS. Formerly, he has been research fellow in the Software and System Engineering Department at Technische Universität München (Germany) and junior scientist at Fraunhofer Center for Experimental Software Engineering (MD, USA). He holds a Ph.D. in Information and System Engineering from Politecnico di Torino (Italy). He is specialized in empirical methodologies and statistical analyses. After working for a few years on methodologies to improve the quality of software and data, he recently steered his research on how to transfer technological innovations to industry and public institutions.
Forrest Shull is assistant director for empirical research at Carnegie Mellon University's Software Engineering Institute (SEI). His role is to lead work with US government agencies, national labs, industry, and academic institutions to advance the use of empirically grounded information in software engineering and cybersecurity. Prior to SEI, Shull was at the Fraunhofer Center for Experimental Software Engineering. He has been a lead researcher on projects for the US Department of Defense, NASA's Office of Safety and Mission Assurance, DARPA, NSF, and commercial companies. Shull served as editor in chief of IEEE Software from 2011 to 2014. He currently serves as associate editor of IEEE Transactions on Software Engineering. Since 2015 Shull has been a member of the Computer Society Board of Governors

and twice served on the Executive Committee. He received his Ph.D. in 1998 from the University of Maryland College Park.
Carolyn Seaman is an Associate Professor of Information Systems at the University of Maryland Baltimore County (UMBC). She is also the Interim Director of the Center for Women in Technology, also at UMBC. Her research consists mainly of empirical studies of software engineering, with particular emphases on maintenance, organizational structure, communication, measurement, and technical debt. She also investigates qualitative research methods in software engineering, as well as computing pedagogy. She holds a Ph.D. in Computer Science from the University of Maryland, College Park, a MS from Georgia Tech, and a BA from the College of Wooster (Ohio).

Funding
Dr. Spínola's contribution to this was supported by CNPq Universal 2014 grant #458261/2014-9 and #201440/2011-3. The participation of Seaman and Zazworka in this work is supported by the US National Science Foundation, award #0916699.

Availability of data and materials
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Graduate Program in Systems and Computer, Salvador University, State University of Bahia, Salvador, Bahia, Brazil. [2]Elsevier, Frankfurt, Germany. [3]Nexa Center for Internet & Society, DAUIN, Politecnico di Torino, Torino, Italy. [4]Carnegie Mellon University, Software Engineering Institute, Arlington, VA, USA. [5]Department of Information Systems, University of Maryland Baltimore County, Baltimore, MD, USA.

Received: 24 January 2019 Accepted: 22 May 2019
Published online: 08 June 2019

References

1. Alves NSR, Mendes TS, Mendonça MG, Spínola RO, Shull F, Seaman C (2016) Identification and management of technical debt: a systematic mapping study. Inf Softw Technol 70:100–121. https://doi.org/10.1016/j.infsof.2015.10.008 ISSN 0950-5849
2. Alves NSR, Ribeiro LF, Caires V, Mendes TS, Spínola RO (2014) Towards an ontology of terms on technical debt. In: The Sixth International Workshop on Managing Technical Debt, Victoria, British Columbia. IEEE Computer Society, Washington, pp 1–7. https://doi.org/10.1109/MTD.2014.9
3. Alzaghoul E, Bahsoon R (2013) "CloudMTD: using real options to manage technical debt in cloud-based service selection", 4th International Workshop on Managing Technical Debt (MTD '13), IEEE Computer Society, pp. In: 55–62, 18–26 May. USA, San Francisco
4. Ampatzoglou A, Chatzigeorgiou A, Avgeriou P (2015) The financial aspect of managing technical debt: a systematic literature review, information and software technology, volume 64, April, pages, pp 52–73. https://doi.org/10.1016/j.infsof.2015.04.001 ISSN 0950-5849
5. Avgeriou P, Kruchten P, Ozkaya I, Seaman C (2016) Managing technical debt in software engineering (Dagstuhl Seminar 16162). In: Dagstuhl Reports (Vol. 6, No. 4). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik
6. Binkley D (2007) Source code analysis: a road map. In: Future of Software Engineering, pp 104–119
7. Brown N, Cai Y, Guo Y, Kazman R, Kim M, Kruchten P, Lim E, MacCormack A, Nord R, Ozkaya I, Sangwan R, Seaman C, Sullivan K, Zazworka N (2010) Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP workshop on Future of Software Engineering Research (FoSER '10). ACM, New York, pp 47–52
8. CAST (2010) Cast worldwide application software quality study: summary of key findings. In: Tech. Rep 2010
9. Curtis B, Sappidi J, and Szynkarski A (2012). Estimating the principal of an application's technical debt in IEEE Software,29(6):34–42.
10. D'Ambros M, Bacchelli A, Lanza M (2010) On the impact of design flaws on software defects. In: Proceedings of the 10th international conference on quality software (QSIC '10). IEEE Computer Society, Washington, pp 23–31. https://doi.org/10.1109/QSIC.2010.58
11. Farias MAF, Silva AB, Mendonça MG, Spínola RO (2015) A contextualized vocabulary model for identifying technical debt on code comments. In: 7th international workshop on managing technical debt 2015, pp 25–32
12. Fowler M, Beck K (1999) Refactoring: improving the design of existing code. Addison Wesley
13. Gat I, Heintz JD (2011) From assessment to reduction: how cutter consortium helps rein in millions of dollars in technical debt. In: Proceedings of the 2nd workshop on managing technical debt. ACM, New York, pp 24–26. https://doi.org/10.1145/-1985362.1985368.
14. 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, pp 167–170
15. Guo Y, Spínola RO, Seaman C (2014) Exploring the costs of technical debt management - a case study. Empir Softw Eng 1:1–24
16. Holvitie J, Leppanen V (2013) DebtFlag: technical debt management with a development environment integrated tool. In: 4th International Workshop on Managing Technical Debt (MTD), pp 20–27
17. Hovemeyer D, Pugh W (2004) Finding bugs is easy. SIGPLAN Not 39(12):92–106
18. Izurieta C, Vetro' A, Zazworka N, Cai Y, Seaman C, Shull F (2012) Organizing the technical debt landscape. IEEE ACM MTD 3rd international workshop on managing technical debt. In: Association with the 34th international conference on software engineering ICSE, Zurich, IEEE Press, Piscataway, pp 2–9
19. Kazman R, Cai Y, Mo R, Feng Q, Xiao L, Haziyev S, Fedak V, Shapochka A (2015) A case study in locating the architectural roots of technical debt. In: Proceedings of the 37th international conference on software engineering - volume 2 (ICSE '15), Vol. 2. IEEE Press, Piscataway, pp 179–188
20. Khomh F, Penta MD, Gueheneuc YG (2009) An exploratory study of the impact of code smells on software change-proneness. In: Proceedings of the 16th working conference on reverse engineering (WCRE '09). IEEE Computer Society, Washington, pp 75–84
21. Li W, Shatnawi R (2007) An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J Syst Softw 80(7):1120–1128
22. Li Z, Avgeriou P, Liang P (2015) A systematic mapping study on technical debt and its management. J Syst Softw 101:193–220. https://doi.org/10.1016/j.jss.2014.12.027
23. Li Z, Liang P, Avgeriou P, Guelfi N, Ampatzoglou A (2014) An empirical investigation of modularity metrics for indicating architectural technical debt. In: Proceedings of the 10th international ACM Sigsoft conference on quality of software architectures. ACM, New York, pp 119–128
24. Lim E, Taksande N, Seaman C (2012) A balancing act: what software practitioners have to say about technical debt. IEEE Softw 29(6):22–27
25. Maldonado ES, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: 7th international workshop on managing technical debt, pp 9–15
26. Marinescu R (2004) Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE international conference on software maintenance (September 11–14, 2004). ICSM. IEEE Computer Society, Washington, pp 350–359
27. Marinescu R (2012) Assessing technical debt by identifying design flaws in software systems. IBM J Res Dev 56(5):9:1–9:13
28. Martini A, Bosch J (2016) An empirically developed method to aid decisions on architectural technical debt refactoring: AnaConDebt. In: IEEE/ACM 38th international conference on software engineering companion (ICSE-C), Austin, ACM, New York, pp 31–40
29. McConnell S (2007). "Technical debt," 10x Software Development Blog, (Nov 2007). Construx Conversations. URL= https://www.construx.com/resources/managing-technical-debt/
30. Morgenthaler J, Gridnev M, Sauciuc R, Bhansali S (2012) Searching for build debt: experiences managing technical debt at Google. In: Third International Workshop on Managing Technical Debt (MTD), pp 1–6
31. Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proceedings of the 28th international conference on software engineering. ACM, New York, pp 452–461
32. Nugroho A, Visser J, Kuipers T (2011) An empirical model of technical debt and interest. In: Proceedings of the 2nd workshop on managing technical debt (pp. 1–8), ser. MTD'11. ACM, New York. https://doi.org/10.1145/-1985362.1985364.
33. Olbrich SM, Cruzes DS, Sjoberg DIK (2010) Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In: The Proceedings of the International Conference on Software Maintenance. ICSM, Timisoara, pp 1–10

34. Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: IEEE International Conference on Software Maintenance and Evolution (ICSME) 2014, pp 91–100
35. Riaz M, Mendes E, Tempero E (2009) A systematic review of software maintainability prediction and metrics. In: 3rd international symposium on empirical software engineering and measurement, pp 367–377
36. Rios N, Mendonça MG, Spínola RO (2018a) A tertiary study on technical debt: types, management strategies, research trends, and base information for practitioners. Inf Softw Technol 102:117–145
37. Rios N, Spínola RO, Mendonça M, and Seaman C. 2018. The most common causes and effects of technical debt: first results from a global family of industrial surveys. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18). ACM, New York, Article 39, 10 p. https://doi.org/10.1145/3239235.3268917
38. Schumacher J, Zazworka N, Shull F, Seaman C, Shaw M (2010) Building empirical support for automated code smell detection. In: Proceedings of the ACM-IEEE international symposium on empirical software engineering and measurement (ESEM '10). ACM, New York Article 8, 10 pages
39. Seaman C (1999) Qualitative methods in empirical studies of software engineering. IEEE Trans Softw Eng 25(4):557–572
40. Seaman C, Guo Y (2011) Measuring and monitoring technical debt. In: Advances in Computers, vol 82, pp 25–46
41. Seaman C, Guo Y, Zazworka N, Shull F, Izurieta C, Cai Y, Vetro A (2012) Using technical debt data in decision making: potential decision approaches. In: The Third International Workshop on Managing Technical Debt (MTD), pp 45–48
42. Soltanifar B, Akbarinasaji S, Caglayan B, Bener AB, Filiz A, Kramer BM (2016) Software analytics in practice: a defect prediction model using code smells. In: Desai E (ed) Proceedings of the 20th International Database Engineering & Applications Symposium (IDEAS '16). ACM, New York, pp 148–155. https://doi.org/10.1145/2938503.2938553
43. Spínola RO, Zazworka N, Vetro' A, Seaman C, Shull F (2013) Investigating technical debt folklore: shedding some light on technical debt opinion. In: Proceedings of the 4th workshop on managing technical debt (MTD '13). https://doi.org/10.1109/MTD.2013.6608671
44. Stochel MG, Wawrowski MR, Rabiej M (2012) Value-based technical debt model and its application. In: 7th International Conference on Software Engineering Advances (ICSEA' 12). XPert Publishing Service, Lisbon, pp 205–212 18–23 November
45. Vetro' A, Morisio M, Torchiano M (2011) An empirical validation of FindBugs issues related to defects. In: 15th annual conference on Evaluation & Assessment in software engineering (EASE) 2011, pp 144–153. https://doi.org/10.1049/ic.2011.0018 11-12 April
46. Xiao L, Cai Y, Kazman R, Mo R, Feng Q (2016) Identifying and quantifying architectural debt. In: Proceedings of the 38th international conference on software engineering (ICSE '16). ACM, New York, pp 488–498
47. Yamashita A, Moonen L (2013) Exploring the impact of inter-smell relations on software maintainability: an empirical study. In: Proceedings of the International Conference on Software Engineering (ICSE). IEEE Press, Piscataway, pp 682–691
48. Zazworka N, Ackermann C (2010) CodeVizard: a tool to aid the analysis of software evolution. In: Proceedings of the ACM-IEEE international symposium on empirical software engineering and measurement (ESEM). ACM, New York Article 63, 1 pages (Poster
49. Zazworka N, Seaman C, Shull F (2011b) Prioritizing design debt investment opportunities. In: Proceedings of the 2nd Workshop on Managing Technical Debt (MTD). ACM, New York, pp 39–42
50. Zazworka N, Shaw MA, Shull F, Seaman C (2011a) Investigating the impact of design debt on software quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt (MTD). ACM, New York, pp 17–23
51. Zazworka N, Spínola RO, Vetro' A, Shull F, Seaman C (2013) A case study on effectively identifying technical debt. In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE). ACM, New York, pp 42–47. https://doi.org/10.1145/2460999.2461005
52. Zazworka N, Vetro' A, Izurieta C, Wong S, Cai Y, Seaman C, Shull F (2014) Comparing four approaches for technical debt identification. Softw Qual J 22(3):403–426 Springer

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com