What are the Practices used by Software Practitioners on Technical Debt Payment? Results From an International Family of Surveys

Boris Pérez
University of Los Andes
Bogotá, Colombia
Francisco de Paula S/der. University
Cúcuta, Colombia
br.perez41@uniandes.edu.co

Camilo Castellanos
University of Los Andes
Bogotá, Colombia
cc.castellanos87@uniandes.edu.co

Darío Correal
University of Los Andes
Bogotá, Colombia
dcorreal@uniandes.edu.co

Nicolli Rios
Federal University of Bahia
Salvador, Brazil
nicollirioss@gmail.com

Sávio Freire
Federal University of Bahia
Salvador, Brazil
Federal Institute of Ceará
Morada Nova, Brazil
savio.freire@ifce.edu.br

Rodrigo Spínola
Salvador University
Salvador, Brazil
rodrigo.spinola@unifacs.br

Carolyn Seaman
University of Maryland
Baltimore County, United States
cseaman@umbc.edu

ABSTRACT

Context: Technical debt (TD) is a metaphor used to describe technical decisions that can give the company a benefit in the short term but possibly hurting the overall quality of the software in the long term. Objective: This study aims to characterize the current state of practices related to TD payment from the point of view of software practitioners. Method: We used a survey research method to collect and analyze - both quantitatively and qualitatively - a corpus of responses from a survey of 432 software practitioners from Colombia, Chile, Brazil, and the United States, as a part of the InsighTD project. Results: We were able to identify that refactoring (24.3%) was the main practice related to TD payment, along with improving testing (6.2%) and improve design (5.8%). Also, we identify that small-sized systems and big-sized systems, along with young systems (less than one year) tend to use more refactoring. As a part of these results, we also could identify that some practices do not eliminate the debt by itself, but support a favorable scenario for TD payment or prevention. Additionally, after comparing the three major TD types cited (code debt, test debt and design debt) we could discover an important similarity of TD payment practices between code debt and design debt. Lastly, we identified that no matter the cause leading to TD occurrence, refactoring remained the most common practice. Conclusion: Definition of practices related to TD payment is an essential activity for software development teams. Developing healthy software systems that can be maintained in the future requires that companies find the right approaches for TD payment.

TechDebt '20, October 8–9, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7960-1/20/05…$15.00
https://doi.org/10.1145/3387906.3388632

CCS CONCEPTS

• **General and reference → Surveys and overviews**; **Empirical studies**; • **Software and its engineering → Software design engineering**; **Software design techniques**.

KEYWORDS

technical debt, technical debt management, payment practices, technical debt causes, family of surveys, insightd

ACM Reference Format:
Boris Pérez, Camilo Castellanos, Darío Correal, Nicolli Rios, Sávio Freire, Rodrigo Spínola, and Carolyn Seaman. 2020. What are the Practices used by Software Practitioners on Technical Debt Payment? Results From an International Family of Surveys. In *International Conference on Technical Debt (TechDebt '20), October 8–9, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3387906.3388632

1 INTRODUCTION

Technical debt (TD) is a metaphor used in software development to describe technical decisions that can give the company a benefit in the short term [9], but possibly hurting the overall quality of the software and the productivity of the development team in the long term [25]. This kind of technical decisions would increase complexity to the code base, making further development hard

and time-consuming [34]. Software companies are forced to work in tight schedules and deadlines to release software to customers in faster cycles, increasing the pressure for development teams to deliver working features to their customers [33].

The easiest way to deal with this problem could be paying the known TD before issues start to show. TD payment refers to resolving or mitigating TD in a software system through technical strategies such as reengineering and refactoring [10]. Differences in payment-related practices are tied to answering what, how and whether to pay the debt [2].

Despite the attention surrounding TD by both the industry and academia, there is still a lack of empirical evidence about payment-related practices used in real-life software development teams [10, 19]. It becomes relevant to gather evidence on how software teams are paying (or support the payment of) the debt injected in their software systems, and if there are activities related to supporting this activity.

Given the aforementioned requirements, the goal of this study is to characterize the current state of the practice of TD payment as described by software practitionersMore specifically, we focus on the comparison of the most used practices on TD payment related to the size of the systems, the age of the systems and the most cited types of TD. Also, we investigate whether or not the causes for the injection of TD have any association with payment practices. To this end, we performed an industrial survey with 432 software practitioners from Brazil, Chile, Colombia, and the United States. Data collection and analysis were done following the protocol developed inside the InsighTD project [23]. Although significant analysis has already been conducted over the available InsighTD data [21–23], much still remains to be studied. In particular, the data has yet to be analyzed with regards to practices on TD payment.

The contributions of this work are twofold. First, this study presents a list of the top five practices on TD payment (refactoring being the most cited) and its proportion considering both the age and the size of the systems. A numerical comparison of similarity between the lists of payment practices for the three major TD types (design debt, test debt and code debt) is provided in this study. And second, an analysis on how causes leading to TD occurrence by software teams can be associated to the TD payment practices identified as a part of this study.

This work has implications for both practitioners and researchers. As a report of current practices of TD payment in software projects, practitioners can use the identified list of TD payment-related practices as an additional instrument to support their decisions on which practices better fit their needs. Also, software practitioners could review the causes that lead to TD occurrence, mapping their situations to these causes, and then identify what payment practice could work in their software projects. For researchers, the findings provide a grounded view of the software industry needs with respect to TD payment. Results can guide new research efforts aligned with the demands and current context of TD payment experienced by industry practitioners.

The rest of the paper is organized as follows: in Section 2 we present a description of the InsighTD project history. In Section 3, we present the survey design, whose results are presented in Section 4. The obtained results are discussed in Section 5, along with useful implications for researchers and practitioners. Section 6 presents the comparison to previous work. Finally, in Section 7, we present threats to validity, and in Section 8 we conclude the paper.

2 INSIGHTD PROJECT

InsighTD project is a globally distributed family of industrial surveys initiated in 2017. Its goal is to gather relevant data about the state of practice of TD, and to improve the understanding of TD management. Surveys are used to gather data about causes leading to TD, the effects of its existence, payment practices, and prevention and monitoring strategies. To date, researchers from 11 countries (Brazil, Chile, Colombia, Costa Rica, Finland, India, Italy, Norway, Saudi Arabia, Serbia, and the United States) have joined the project. Data collections from Brazil, Chile, Colombia, and the United States are complete.

Several studies have been performed since 2017. First one [23] discussed the InsighTD project creation, the survey design process and some preliminary results for Brazil only about the top 10 causes and effects of TD. Following works, also in Brazil, focused on the causes and effects of TD in agile software projects [21] and on the use of cross-company probabilistic cause-effect diagrams to represent information about the TD causes and effects [22].

Replication in other countries have presented initial findings. Pérez et al. [15] focused on understand the familiarity with the TD concept and how practitioners react to the presence of debt in the Chilean software industry. In [7], Freire et al. focused on exploring, from the point of view of software practitioners, preventive actions that can be used to curb the occurrence of TD and the impediments that hamper the use of those actions. And Rios et al. in [20] investigated how software practitioners perceive the occurrence of documentation debt in their projects. This study used data from replications of Brazil, Chile, Colombia and the United States, and from an interview-based case study performed by researcher partners who were not involved in the InsighTD project.

Thus, although significant analysis has already been conducted over the available InsighTD data, much still remains to be studied. In particular, the data has yet to be analyzed with regards to practices on TD payment.

3 STUDY DESIGN

The research method depends on the research questions, therefore, a survey research method is used to identify industry practices related to TD payment [4]. This study follows the activities of a survey process proposed by Pfleeger and Kitchenham [17]: i) Set research objectives and questions, ii) Plan and schedule the survey, iii) Repository of resources, iv) Design the survey, v) Prepare the data collection instrument, vi) Validate the instrument, vii) Select participants, viii) Administer the instrument, ix) Analyze the data, and x) Report the results.

3.1 Survey Design

This section presents the research objectives and the research questions supporting the survey, along with its planning, availability and design.

3.1.1 Research Objective and Research Questions. The research was designed with the goal of characterizing the current state of the practice of TD payment. More specifically our goal can be

formalized by following the Goal-Question-Metric approach [3]: Identify and analyze (purpose) the main practices (issue) used to pay off (or support the payment of) technical debt in software system projects (object) from software practitioners (viewpoint).

Based on our research goal, we derived the following two research questions guiding our study and the reporting of the results:
RQ1: What are the main practices used by software development teams to pay off the debt in their projects?

By answering this research question we aim to identify the main practices used to pay off (or support the payment of) the debt injected in software system projects. Variables such as age and size of the systems, and major TD types were used to identify similarities or differences among the payment practices used by software development teams.
RQ2: Is it possible to establish an association between main causes leading to TD occurrence and main practices used on TD payment by software development teams?

By answering this research question we aim to uncover whether it is possible to establish a relationship between causes of TD injection and the main practices used on TD payment by software teams. The motivation for this question is to find the most appropriate form of practice depending on the cause for which the debt was injected. This will allow software teams to have a reference for actions to take to pay off the debt.

3.1.2 Plan and Schedule of the Survey. In this survey, data gathering was done using an online questionnaire to increase the number of possible participants. Invitations were sent to software practitioners through LinkedIn, industry-affiliated member groups, mailing lists, and industry partners, as invitation channels. Invitations sent followed a defined protocol: first, an initial invitation explaining the purpose of the survey and people involved, and second, a reminder sometime later (one month). Survey was anonymous, so reminders were sent to everyone.

3.1.3 Repository of Resources. As previous mentioned, this research used an online survey. Google Forms was used as the platform to design it and distribute it. Responses were automatically managed by Google, ensuring that no errors would be introduced during the recording of the responses. Surveys can be configured to be available anytime.

3.1.4 Design the Survey. The goal of this survey is to gain a deeper understanding of the main practices used by software teams to pay off the TD in their projects. This survey can be classified as both exploratory and descriptive research because it was preplanned and structured, it is focused on the discovery of insights, and the information collected can be statistically inferred over a population [27]. Due to the online nature of the survey, researchers did not intervene while participants filled up the survey. Despite the fact that this survey asks participants about a specific past experience related to a software system project, it can not be considered a cross-sectional study [8] because respondents used different points in time.

3.2 Survey Instrument Design

This section presents the data collection instrument, its validation, and the selection of participants.

3.2.1 Prepare the Data Collection Instrument. Developing the right set of questions is the most important part of a questionnaire. Survey questions were defined within the InsighTD replication package. The survey instrument was made up of 28 questions previously described in [23]. Table 1 presents the subset of the survey's questions related to the context of this work. Close-ended questions are single choice.

Table 1: Survey questions

RQ	No.	Question	Type
-	Q1	What is the size of your company?	
-	Q2	In which country you are currently working?	
-	Q3	What is the size of the system being developed in this project? (LOC)	
-	Q4	What is the total number of people of this project?	Closed
-	Q5	What is the age of this system up to now or to when your involvement ended?	Closed
-	Q6	To which project role are you assigned in this project?	
-	Q7	How do you rate your experience in this role?	
-	Q8	Which of the following most closely describes the development process model you follow on this project?	Closed
-	Q13	Please give an example of TD that had a significant impact on the project that you have chosen to tell us about:	Open
RQ2	Q16	What was the immediate, or precipitating, cause of the example of TD you just described?	Open
RQ2	Q17	What other cause or factor contributed to the immediate cause you described above?	Open
RQ2	Q18	What other motives or reasons or causes contributed either directly or indirectly to the occurrence of the TD example?	Open
RQ1	Q26	Has the debt item been paid off (eliminated) from the project?	Closed
RQ1	Q27	If yes, how? If not, why?	Open

Questions Q1 to Q8 characterize the survey participants and their contexts. Question 13 asks participants to describe an example of a TD item that occurred in their software project (this example is used as a basis for answering TD payment questions) and indicate the representativeness level of this example, respectively. Finally, Q26 and Q27 ask participants whether or not the TD item (from Q13) was eliminated and how. InsighTD was aware that could be possible to expect subjects not being extremely familiar with the TD concepts. This scenario was mitigated including a TD definition adapted from McConnell [13].

3.2.2 Validate the Instrument. As presented by Rios et al. in [23], the validation phase comprised three main steps: internal validation, external validation, and pilot study. The internal and external validations intend to ensure that the survey questions are clearly interpretable and sufficiently complete to answer the research questions. Six months were required to complete this stage. For the pilot study, the goal was to observe if the questionnaire was well understood by a small number of participants who represent the target population of the study.

This survey, as a part of InsighTD, was first designed in English. However, its application was done accordingly to the language of the participant countries. In the US, the survey was applied in English. In Colombia and Chile, the survey was applied in Spanish. In Brazil, the survey was applied in Portuguese. Translation of questions required a second validation phase focused on reviewing whether the questions were consistent and faithful to the English questions. This phase also executed an internal validation, an external validation, and a pilot study. This was done by the members of the teams and with the help of external reviewers.

3.2.3 Selection of Participants. Since we intend to investigate the state of the practice, the questionnaire was answered only by software practitioners. We intend to search for professionals that work on a variety of roles in a software project. To reach the target population we utilized the social media platform LinkedIn along with industry-affiliated member groups, mailing lists, and industry partners, as invitation channels. In the case of LinkedIn, this social network gave us direct access to a large number of professionals with whom we did not have previous connection. Vast majority of participants were first invited to connect, and after their acceptance, the invitation to participate in the survey was sent. For example, in Colombia, 1,500 invitations to connect and then to participate were sent through LinkedIn. In Chile, almost 1,900 invitations were sent. Additionally, LinkedIn allowed us to use keywords to search for professionals with specific expertise. The following keywords were considered: configuration analyst, configuration manager, developer, process analyst, product owner, programmer, project manager, requirements analyst, software architect, software engineer, system analyst, test analyst, test manager, and tester.

In total, 432 practitioners answered the survey: 107 from Brazil (24.8%), 92 from Chile (21.3%), 134 from Colombia (31.0%), and 99 from the United States (22.9%). Participants of the survey were informed that the results of the study will only be published in an aggregated form.

3.3 Data Analysis

The survey instrument is composed of a mix of closed and open questions. Thus, we need to rely on a variety of procedures for data analysis. For the analysis of the answers to closed questions, we first relied on descriptive statistics to get a better understanding of the data. For the nominal data, we calculated the share of participants choosing each option.

Answers for open-ended questions were codified using a coding schema provided with the InsighTD replication package. In answers given to Q27, we initially applied manual-open coding resulting in a set of codes. We only used codes where answers to Q26 were positive. The process was performed iteratively revising and unifying codes at each cycle of analysis until reaching the state of saturation, i.e., a point where no new codes were identified. At the end of the analysis, we obtained a stable list of codes along with their citation frequency. The following steps were performed: (i) Two researchers analyzed each participant's answer, (ii) payment practices codes were identified following the InsighTD's protocol, (iii) a comparison of the payment practices codes was performed between researchers' analysis to get the final list of codes of payment practices, (iv) final payment practices were grouped into categories and, (v) the number of times that each payment practice/category cited was counted.

At least three researchers conducted the coding in each of the four InsighTD replications. Each researcher could assume one of the following roles: (i) code identifier is responsible for extracting the existing codes in the answers, (ii) code reviewer is responsible for reviewing all extracted codes, and (iii) referee is responsible for resolving disagreements in codes identified by the code identifier and code reviewer.

4 RESULTS

This section presents the initial results on practices used for TD payment. This section is organized as follows: Section 4.1 describes the demographics of the participants. Section 4.2 presents the results about the main practices related to TD payment (RQ1). Section 4.3 presents the association found between causes and payment-related practices (RQ2).

4.1 Characterization of the Participants

In total, we have 432 answers from participants in Brazil (107), Chile (92), Colombia (134), and the United States (99). Participants are well distributed among small (27.9%), medium (38.2%), and large (33.9%) companies. Most participants tend to work in teams of 5-9 people (30.8%), followed by teams of 10-20 people (24.7%). Regarding the age of the system developed in the project, most indicated age 2 to 5 years (34.6%). There are also a significant number of systems represented from 1 to 2 years (22.5%). Related to the role within the development process, most reported working as Developers (41.8%), followed by Software Architects (15.7%), and Project Manager (15.1%).

Thus, in general, although it is not possible to guarantee that the participants represent all the professionals in the software industry from Brazil, Chile, Colombia, and the United States, participants are characterized by representing a broad and diverse audience reaching different functions and participant experience levels, different sizes of organizations and projects of different ages and team sizes.

4.2 Main Practices Related to TD Payment (RQ1)

The RQ1 covered the identification of main practices related to TD payment: "What are the main practices used by software development teams to pay off the debt in their projects?". In total, as informed by Q26-27, we found 47 practices used by software development teams to pay off (or support the payment of) debt in their projects. It is worth mentioning that the debt paid back only refers to explicit debt that practitioners were aware of. The five most commonly cited practices related to TD payment used by

software practitioners are visualized in Figure 1. These practices represent 46.5% of the set of all identified practices. These results show that *refactoring* is the most cited practice used on TD payment (55 citations), and it is almost four times greater than the second most cited practice: *improve testing* (14 citations). *Improve design* is in third place, *Adoption of good practices* in fourth place, and *repayment activities* in fifth place. Another practice, not listed here, is *improve documentation*. This is an important activity because TD is not usually documented as a consequence of having time pressure or deadlines [16].

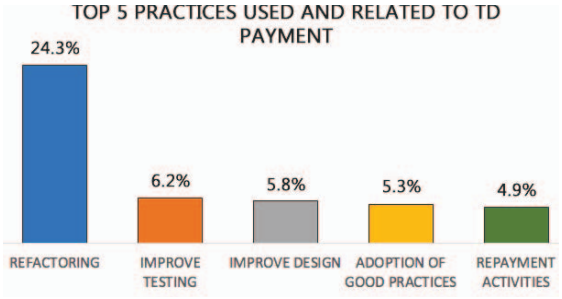

Figure 1: Top Five Most Cited Practices on TD Payment

By going further into the analysis of the whole set of identified practices related to the payment of TD items, we realize that some practices do not allow the elimination of TD items. For instance, the practice *adoption of good practices* contributes to create a favorable scenario for eliminating future TD items, but it does not eliminate the current item by itself. Other practices such as *improve testing* can be seen as preventative practices. Thus, TD payment-related practices encompass practices associated with TD payment, prevention, and also the creation of a favorable scenario for paying off debt items. A practice that could be hard to understand without some context is *repayment activities*. This practice is related to the specific action of pay TD but without including additional information about the specific payment process carried out that could help in the codification process. Majority of actions performed under this code could be related to *refactoring*, but again, without a deeper explanation of the specific repayment action, it become difficult to establish this relation. Some notable quotes from the survey for this last practice are: "Stop activities to pay the debt off", "Repayment Campaings" and "Pay off TD items."

As part of this research, it is important to better understand how these practices are used by different configurations of software systems. Figure 2 present five categories related to the size (LOC) of the software system used by respondents as an example of a TD case (Q13) in their companies. Each category include the percentage of cases where TD was paid. The category *10+ MLOC* have the highest rate of debt paid off (65.2%), followed by *1 - 10 MLOC* with 50% of cases paid off. At first sight, it is possible to establish an association between the size of the software systems and the amount of cited cases of *refactoring*. Small systems (less than 10 KLOC) and huge systems (more than 10 MLOC) tend to use more refactoring as a TD payment practice. On the other hand, medium size systems (100 KLOC to 1 MLOC) have the lower number of citations of *refactoring*. This kind of systems are more focused than the others on stop all development activity and pay the debt off (repayment activities). Also, medium size systems have the highest rate of citations of *improve testing*. This goes against systems with less than 10 thousand lines of code where *improve testing* practice was not even quoted.

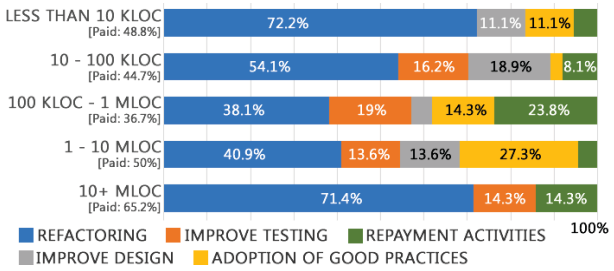

Figure 2: Top five practices on TD payment by system size

Figure 3 present five categories related to the age (years) of the software system used by respondents in Q13. Each category include the percentage of cases where TD was paid. The category *2 - 5 years* have the highest rate of debt paid off (50.4%), followed by *less than 1 year* with 46.3% of debt paid off. From Figure 3 it is possible to establish an association between the age of the software systems and the amount of cited cases of *refactoring*. Software systems with *less than 1 year* (70.6%) of development tend to use more refactoring than software systems with *more than 10 years* (29.4%) of development. The decrease of *refactoring* could be proportionally related to the presence of *repayment activities*. *Repayment activities* are only quoted in systems from 2 to more than 10 years. *Improve testing* have a strong presence in systems of *5 - 10 years* with 23.5% of citations.

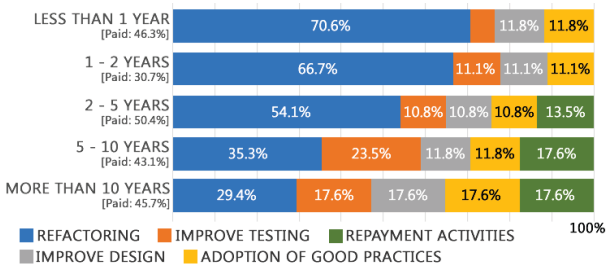

Figure 3: Top five practices on TD payment by system age

Finally, and considering that the objective of this study is to better understand the TD payment practices used by software practitioners, it was decided to compare TD payment practices according to the main types of TD described by the respondents. TD types were codified based on the example provided in question 13. The three main types of TD, as cited by the participants, are: design debt (17.9%), test debt (17.1%) and code debt (16.4%). The goal was to measure quantitatively how similar the practices used among the TD types are. To this end, we used a similarity measure for indefinite rankings called rank-biased overlap (RBO) [29]. RBO was

used because supports top-weighted ranked lists, ranked list could be incomplete and the decision to truncate the ranking at any depth is arbitrary. RBO is defined as follows:

$$RBO(S,T,p) = (1-p)\sum_{d=1}^{\infty} p^{d-1} \cdot A_d$$

where S and T are the ranked lists; p is the probability of looking for overlap at depth $d+1$ after having examined element at d. The smaller the p value, the more top-weighted the metric. A_d is the agreement between S and T at depth d, i.e. the proportion of S and T that are overlapped. Specifically, we use RBO_{EXT} as a single score to extrapolate from the visible lists, assuming that the degree of agreement seen up to depth k is continued indefinitely:

$$RBO_{EXT}(S,T,p,k) = \frac{X_k}{k} \cdot p^k + \frac{1-p}{p}\sum_{d=1}^{k} \frac{X^d}{d} \cdot p^d$$

where k is the length of the prefix; and X_k is the intersection length of both ranks at depth k.

Figure 4 depicts the RBO comparison among the three lists of practices related to TD payment (one line for each pair of td types). By increasing the p value, this comparison aims to explore how similar these practices are per group at the top of their rankings and at the bottom of their rankings. Comparison went from $p = 0.5$ (top 2 elements approx.) to $p = 0.97$ (top 33 elements approx.).

For $p = 0.5$, the Code Debt-Design Debt pair showed the highest similarity (RBO=0.73). This means that first practices in both lists are quite similar. On the other hand, for $p = 0.5$, the Code Debt-Test Debt pair exhibited the lowest similarity (RBO=0.2). The pair Test Debt-Design Debt begin with RBO=0.21 at $p = 0.5$ and increase to RBO=0.59 at $p = 0.97$. This behavior means that first practices in both lists are not similar, and then they become similar as more elements are compared. The Code Debt-Design Debt pair keeps the highest similarity when all practices of both types of TD are compared, RBO=0.66 at $p = 0.97$.

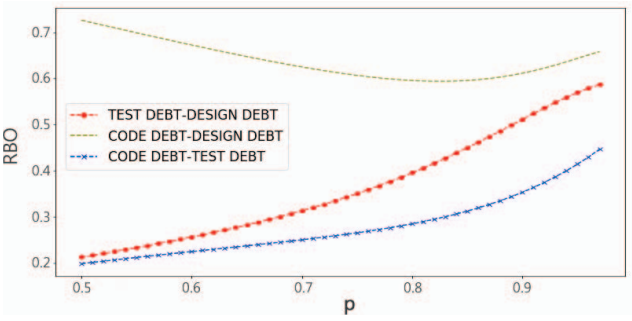

Figure 4: RBO of practices related to TD payment ranks per TD type

Refactoring, *improve testing* and *improve design* are the most selected practices used to pay off (or support the payment of) technical debt in software projects. *Adoption of good practices* along with *repayment activities* are the most used practices that contribute to creating a favorable scenario for eliminating TD items. *Refactoring* is most used by small and huge software systems, and by software systems with less than 1 year of development. *Refactoring* usage tend to decrease accordingly to the age of the system. Finally, list of TD payment practices related to code debt and design debt are the most similar among others types de TD.

4.3 Association Between TD Causes and TD Payment-Related Practices (RQ2)

To answer the RQ2 "Is it possible to establish an association between main causes leading to TD occurrence and main practices used on TD payment by software development teams?", we took the ten most cited causes leading to TD occurrence together with an associated heat map against the most cited TD payment-related practices (Figure 5). We intended to identify what practices on TD payment are used to pay (or support the payment of) the debt injected by certain situations (causes) as described by software teams. The five most-cited payment-related practices, as presented in Figure 1, are highlighted by their corresponding color.

In total, as informed by the participants in Q16-18, we found 111 causes that lead development teams to incur in technical debt in their projects. The first ten main causes shown in Figure 5 represent 41% of all causes cited. Each cause is complemented by their percentage of debt paid. *Deadline* is the most cited caused of TD, followed by *inappropriate planning* and *not effective project management*. Only one cause was paid off above 50%: *Non-Adoption of Good Practices*. This cause could be easily paid off considering the number of tools used for this purpose. Antipatterns, smells, bad coding practices are some examples of non-adoption of good practices [28]. The practice *Adoption of Good Practices* is not the most cited practice for this cause, and it makes sense because this practice could be more like a preventative practice envisioned to avoid future cases of TD. *Refactoring* is the main payment practice used to pay off the debt when the cause is *Non-Adoption of Good Practices*. Some respondents were explicit about actions taken, for example: removing duplicated code, resolving code smells or splitting god objects.

In Figure 5 we can see that *refactoring* is the most selected payment practice regardless the cause of the debt described by the participants. Only two causes escape from this practice: *lack of a well defined process*, for which there is no most selected practice; and *not effective project management*, for which the most selected practices are *improve testing* and *refactoring*. Usually, limitations or constraints on time, cost and resources of a project are consequences of bad project management. These limitations could also affect testing tasks, reducing the time and resources available to run all the tests. The "Risk-Based Testing" approach could be use by software teams to meet the time schedule with expected quality, when the testing timelines are very tight and there are not enough resources to carry out sufficient testing. In this approach, the team has clearly planned and identified the test scenarios based on the "Project Risk" criteria. Hence, it is essential to strike a balance in

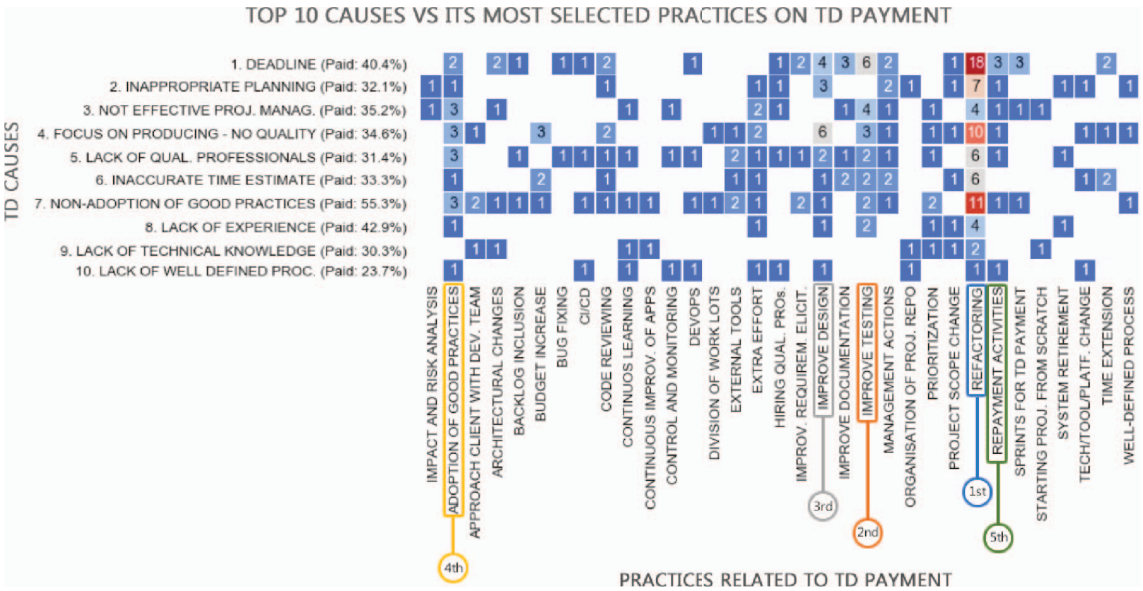

Figure 5: The ten most cited causes vs their practices related to TD payment

optimizing the testing in order to get the maximum benefit with the reasonable effort of testing [1, 26].

Improve testing and *improve design* are the second and third most cited practices when the cause was *deadline*. *Improve design* was the second most used practice when the cause was *focusing on producing more without quality*. This could be related to the fact that changes at top level need to be done in order to produce better software.

> Refactoring is the main payment practice of development teams to pay off the debt no matter what caused it to be injected. The only two causes using a different practice was *not effective project management* and *lack of a well defined process*.

5 DISCUSSION

In this section, we discuss the findings of this study by interpreting the obtained results and analyzing the implications of the results for researchers and practitioners who are working on TD.

5.1 Review of the research questions

This study acknowledges that practices described for TD payment were cited based on a single case, from the point of view of the respondent, without having a full grasp of the business context and described in retrospect to experiences. As our survey was anonymous, we could not ask the respondents to deeper explain their answers, but just analyze the variability on the values for the question.

5.1.1 RQ1: What are the main practices used by software development teams to pay off the debt in their projects? This study found 47 TD payment-related practices, where the five most cited practices were *refactoring*, *improve testing*, *improve design*, *adoption of good practices* and *repayment activities*.

Refactoring was almost four times greater than the second most cited practice. Refactoring is a technique for improving the design of an existing codebase, design, or architecture of a software system without altering its external behaviors [6, 10]. In the vast amount of answers, refactoring was used by software teams in terms of code refactoring. Refactoring of designs or architectures was coded under other practices such as: *improve design* and *architectural changes*. Given the big amount of answers for *refactoring*, it also should be good to have a more specific codification scheme. For example, some answers could be associated to *rewriting*, however, the lack of context and a deeper explanation by the respondent, dismiss this possibility. Rewriting means rewrite the code that contains TD [10], so, it means that the external behavior could change.

It is interesting to note that *improve testing* is the second most cited practice. It is important for software teams to improve how tests are designed and executed than actually improve the architecture or design of the software system. This is specially true in systems with 100 KLOC to 1 MLOC, or in systems between 5 to 10 years, as presented in Figures 2 and 3. Also, some types of testing, such as white box testing or mutation testing, could be more related to the identification of TD, so it could be understood as a better way to avoid the injection of TD in the software system.

Improve design is in third place. According to Ernst et al. in [5], architecture debt is the most common source of TD, therefore, practices to pay off the debt would be more related to changes in design or architecture. It is not clear whether the debt injected into the architecture and paid off in the code, is later reflected in the artifacts related to the architecture. For example, lack of modularity is associated with architecture debt, however, in order to pay it, it must be done in the code. This could be also related to documentation debt, when documents start to become outdated with respect to the implemented code [16].

From the point of view of the lines of code of software systems size, refactoring have a strong presence in small systems (less than 10 KLOC) and big systems (10+ MLOC). In the case of small systems, it could be explained by the rapid development of small pieces of software systems, where there is no time to document or even to improve the system design, and every improve is done in the code. In the case of big systems, having a lot of work done through the years, it becomes hard to maintain them, and therefore, a lot of changes and fixes are done in the code. This is complemented with repayment activities, where teams stop their daily activities to perform improvements in the code. The medium-sized systems, focus a little more in improving how tests are performed and in to get some time to fix debt items.

From the point of view of the lines of code of software systems size, it is possible to acknowledge a proportional ratio between refactoring and the age of the system. The older the system, the less refactoring will be used. In the case of systems with more than 10 years, maintainability and changes in the code could be a challenge, therefore, different strategies could be used by the software teams, such as *improve testing* or *improve design*. Software systems with less than a year (70.6%) of development tend to use more refactoring than software systems with more than 10 years (29.4%) of development.

5.1.2 RQ2: Is it possible to establish an association between main causes leading to TD occurrence and main practices used on TD payment by software development teams? Results showed that eight out of 10 causes leading to TD occurrence use refactoring as the main payment practice. *Deadline*, as the main cause, directly affects the decisions that ended in the code. Companies related to software development have an increasing pressure to deliver software systems as soon as possible [2, 12]. Therefore, if the code works, it goes to production.

Causes such as *lack of qualified professionals* do not seem directly related to the code. This lack of qualification could be in every step in the software development process such as: requirement elicitation, requirements analysis, architecture design, and tests. In the context of this survey, answers were explicit about the kind of professionals: "Little experience of the analyst", "Inexperienced team. Bad definition of the requirement or history" and "Lack of personnel with theoretical and technical skills necessary for development".

Also, it is possible to understand some of the causes as a consequence of something else. For example, *deadline* and *pressure* can be a consequence of having a bad project management (*not effective project management*, *inappropriate planning* and *inaccurate time estimate*).

In the end, whether or not the problem is at a management level or at an architectural level, everything leads to the code, so, it could be expected that the place to first pay the debt be the code. Following this idea, it could be possible to understand why *refactoring* was selected no matter what cause injected it. Following practices could be changing/fixing things up to the management.

5.2 Implications to researchers and practitioners

There are several practices related to TD payment that have been used by software development teams in Brazil, Chile, Colombia, and the United States. However, these payment-related practices by themselves are not enough. It is necessary to analyze the differences among them in order to understand the nature of the required changes (improves) and the resources needed. For example, refactoring, as a TD payment practice, says too little about how it should be implemented by software teams. As stated by Li, Avgeriou, and Liang in [10], refactoring means making changes to the code, design, or architecture of a software system. As can be seen, there are at least three places to implement the refactoring.

Despite the type of payment-related practice used by software teams, it is also necessary to define the frequency of the payment practice. A software team can either choose to repay TD continuously, occasionally, or not at all [33]. Some teams can reduce their TD by paying a certain percentage every month, while some teams can focus on new features, and therefore leaving TD reduction to minimum [18]. In [11], Martini and Bosch suggest that TD should be repaid using partial payments, reducing the risk associated with the payment of the debt.

Another aspect that needs to be considered is the cost related to the debt. How much would it cost to pay off the debt (principal)? vs. How much does it cost to maintain the debt (interest)? According to Martini and Bosch [11], it could be more profitable to delay the payment, i.e. continue paying interest. The last aspect to be considered is the team's capacity to perform the refactoring [33]. As can be seen, it is not just about to pay the debt, it is about to analyze several aspects before taking any decision.

Software practitioners can benefit from the results of this study by using the list of the most cited TD payment-related practices as an additional instrument to support initial efforts to pay off the debt injected in their software projects. It is important to first understand the differences among these practices, and then, begin by resolving the aspects presented at the beginning of this section. Also, for having responded RQ2, software practitioners could review the causes associated to the presence of TD and map their situation to these causes, and then identify what payment practice could work in their software projects. Finally, along with the payment practices, it is important to start thinking about how to prevent the injection of TD in their projects. Several of the practices found in this study are related to prevention strategies, such as: *Adoption of Good Practices*, *Improvement in Requirement Elicitation* and *Adequate Impact and Risk Analysis*. Project management needs to get involved in the software development process, in order to reduce the presence of TD in software projects. Many of the causes of TD can be related to the management of the project.

For researchers, results support future research by providing insights into software practitioners' perspectives on TD payment practices. Finally, the global family of surveys allows not only researchers to reproduce the results and their interpretation, but also practitioners to evaluate their own TD situation against overall industrial trends.

As a final remark, it is important to note that these results do not set any advantages or disadvantages related to a payment practice. It will depend on the software team the success of the implementation of any practice. The software team can pay off the debt under demand, or because they want to improve the software.

6 RELATED WORK

Although there are previous works that point out some causes and effects of TD [5, 31, 32], their sample sizes tend to be quite small, or limited to a small number of organizations. Besides, a predefined list of TD causes constrained participants to fit their experience into that list. Further, half of the relevant studies focused on architectural TD, just one type of debt, and payment strategies have not been sufficiently studied.

The survey presented in [5] reports the results obtained from 1,831 software practitioners. They studied the common understanding of the TD term, how much of technical debt is architectural nature, and the management practices and tools used to deal with TD. In [31], Yli et al. investigated the causes and effects of TD by interviewing 12 persons in a Finnish software company. Their results reported that TD is mainly the result of intentional decisions to reach deadlines, and customer satisfaction was the main reason for taking TD in the short-term, but it turned to economic and quality issues in the long-term. The analysis published in [32] is also applied in a Finnish software company, and it showed that decisions to resolve a technical issue are often intentional and forced by time-to-market requirements, but the negative consequences of those decisions are not always considered, or they are underestimated.

Finally, although initial analysis has already been conducted over the available InsighTD data, much still remains to be studied. In particular, the data has yet to be analyzed with regards to what practices related to TD payment have been conducted by software development teams, and how these practices change among countries. Pacheco et al. [14] reported another InsighTD replication in Costa Rica, where 156 software professionals reported that TD was the product of preventable situations, TD was monitored for slightly more than half of the cases, and TD was not paid in most cases.

To summarize, this work becomes a part of the InsighTD project, but, unlike the previous replications, we aim to study the payment related practices used in software projects, and how the causes and the payment practices can be associated.

7 THREATS TO VALIDITY

There are threats to validity in this work that we attempt to mitigate and remove when possible. According to Runeson and Höst [24], the main threats regarding this work are: external validity, internal validity, construct validity and reliability.

External validity. This study focused on the perception of TD from the point of view of 432 software practitioners from Brazil, Chile, Colombia and The United States. Respondents are well distributed based on their skills (project role, role experience), software projects participation (system age, system size, number of people involved) and company size. Thus, we reduce this threat by achieving diversity of participants who answered the survey. However, considering the small number of participants per country, results can not be generalized. This means that it is possible that results might differ in other countries or even in other software projects within the participant countries in this study. Also, the anonymous basis of the survey implied that more than one respondent could refer to the same TD case example (Q13), therefore, reducing the possibility of characterizing 432 different software projects. Another threat to validity of this study is the results get altered by participants' reactions to being studied (Hawthorne effect). This threat was mitigated by both the online and anonymous basis of the survey.

Internal validity. Maturation is the main threat to internal validity of this study. It implies that the participants can react differently as time passes, in this case, if the survey is too long [30]. The fact that all participants answered the whole questionnaire is a signal that this threat was not raised. Related to selection bias, LinkedIn platform was used to enlarge the respondents' sample by inviting participants from a variety of roles and cities within a country. Vast majority of participants were first invited to connect, and after their acceptance, the invitation to participate in the survey was sent. For example, in Colombia, 1,500 invitations to connect and then to participate were sent through LinkedIn.

Construct validity. To prevent hypothesis guessing and evaluation apprehension [30], we explain in the invitation to the survey the goal of the study and request that interviewees reply to questions by relying on their own background. Also, to make sure that the respondents were considering the correct interpretation of what TD is, we have reported, before the questions, the definition of TD according to McConnell [13]. Moreover, the InsighTD questionnaire has two questions (not listed in Table 1) that allow us to perceive if respondents have a correct understanding about TD: "In your words, how would you define TD?" and "Please give an example of TD that had a significant impact on the project that you have chosen to tell us about". By using the answers provided for both questions, we can observe if practitioners are able to provide valid examples of debt items, indicating that they are thinking correctly about the subject and that the answers given to the other survey questions are based on valid example. If the provided example is not valid, we do not consider the answers from that respondent. However, no definitions of TD payment-related practices were provided before questions. Therefore, respondents may have mistaken the differences between refactoring and rewriting, and moreover, differences among code refactoring, design refactoring or architecture refactoring. Not all refactoring is at code level, as stated in the literature [6, 10].

Reliability. To mitigate this threat and to avoid potential coding process dependencies on the researcher's subjective criteria, the coding activity was performed individually by two researchers, and then, discussed until an agreement was reached. A referee was required in some cases to help to resolve disagreements in codes identified. Also, because all interpretations are tentative ones, it is not possible to support causality, but only report trends and general beliefs in the state-of-practice. The complete process was explained in Section 3.3.

8 CONCLUSIONS

The contributions of this work are twofold. First, this study presents a list of the top five practices on TD payment (refactoring being the most cited) and its proportion considering both the age and the size of the systems. A numerical comparison of similarity between the lists of payment practices for the three major TD types (design debt, test debt and code debt) is provided in this study. And second, an

analysis on how causes leading to TD occurrence by software teams can be associated to the TD payment practices identified as a part of this study. In total, 432 practitioners answered the survey allowing us to investigate practices used on TD payment on software projects in Brazil, Chile, Colombia, and the United States.

We found that *refactoring* is the main practice used to pay off the debt, followed by *improve testing* and *improve design*. TD payment-related practices encompass practices associated with TD payment, prevention, and also the creation of a favorable scenario for paying off debt items. We also could identify that refactoring is more used in younger systems (between less than a year and 2 years) and in both small (less than 10 KLOC) and big systems (more than 10 MLOC). The smallest percentage of refactoring usage was found in systems with more than 10 years (29.4%). Related to the comparison of the major TD types found as a part of this study (test debt, code debt and design debt), we could identify that the list of TD payment practices of code debt and design debt, have the highest similarity, in comparison with code debt-test debt and test debt-design debt. Finally, from the top 10 of causes (*deadline* is the first one) leading to TD occurrence, *refactoring* was the most cited practice used.

The next steps of this research include: (i) a drill down of the responses on repayment activities and design improvement, (ii) include more variables and a joining of them to identify possible patterns of TD payment practices, (iii) running other possible analyses including others reactions to TD, such as monitoring practices and prevention actions. Also, include the replication data from other countries such as: Finland, Saudi Arabia, Serbia, and Costa Rica.

REFERENCES

[1] Ståle Amland. 2000. Risk-based testing:: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software* 53, 3 (2000), 287–295.

[2] Terese Besker, Antonio Martini, and Jan Bosch. 2018. Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software* 135 (2018), 1–16.

[3] Gianluigi Caldiera and Dieter Rombach. 1994. The goal question metric approach. *Encyclopedia of software engineering* (1994), 528–532.

[4] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. *Selecting Empirical Methods for Software Engineering Research.* Springer London, London, 285–311.

[5] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. 2015. Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 50–60. https://doi.org/10.1145/2786805.2786848

[6] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[7] Sávio Freire, Manoel Mendonça, Davide Falessi, Carolyn Seaman, Clemente Izurieta, and Rodrigo Oliveira Spínola. 2020. Actions and Impediments for Technical Debt Prevention: Results from a Global Family of Industrial Surveys. In *To appear in the Proceedings of the 35th ACM/SIGAPP Symposium On Applied Computing*. ACM.

[8] Barbara A. Kitchenham and Shari Lawrence Pfleeger. 2002. Principles of Survey Research Part 2: Designing a Survey. *SIGSOFT Softw. Eng. Notes* 27, 1 (Jan. 2002), 18–20. https://doi.org/10.1145/566493.566495

[9] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *Ieee software* 29, 6 (2012), 18–21.

[10] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220.

[11] Antonio Martini, Jan Bosch, and Michel Chaudron. 2015. Investigating Architectural Technical Debt Accumulation and Refactoring over Time. *Inf. Softw. Technol.* 67, C (Nov. 2015), 237–253. https://doi.org/10.1016/j.infsof.2015.07.005

[12] Antonio Martini, Erik Sikander, and Niel Madlani. 2018. A semi-automated framework for the identification and estimation of Architectural Technical Debt: A comparative case-study on the modularization of a software component. *Information and Software Technology* 93 (2018), 264–279.

[13] Steve McConnell. 2007. Technical Debt. http://www.construx.com/10x_Software_Development/Technical_Debt/

[14] Alexia Pacheco, Gabriela Marín-Raventós, and Gustavo López. 2019. Technical Debt in Costa Rica: An InsighTD Survey Replication. In *International Conference on Product-Focused Software Process Improvement*. Springer, 236–243.

[15] Boris Pérez, Juan Pablo Brito, Hernán Astudillo, Darío Correal, Nicolli Rios, Rodrigo Oliveira Spínola, Manoel Mendonça, and Carolyn Seaman. 2019. Familiarity, Causes and Reactions of Software Practitioners to the Presence of Technical Debt: A Replicated Study in the Chilean Software Industry. In *38th International Conference of the Chilean Computer Science Society*. IEEE.

[16] Boris Pérez, Darío Correal, and Hernán Astudillo. 2019. A Proposed Model-Driven Approach to Manage Architectural Technical Debt Life Cycle. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 73–77. https://doi.org/10.1109/TechDebt.2019.00025

[17] Shari Lawrence Pfleeger and Barbara A. Kitchenham. 2001. Principles of Survey Research: Part 1: Turning Lemons into Lemonade. *SIGSOFT Softw. Eng. Notes* 26, 6 (Nov. 2001), 16–18. https://doi.org/10.1145/505532.505535

[18] Ken Power. 2013. Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options. In *2013 4th International Workshop on Managing Technical Debt (MTD)*. IEEE, 28–31.

[19] Nicolli Rios, Manoel Gomes de Mendonça Neto, and Rodrigo Oliveira Spínola. 2018. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology* 102 (2018), 117–145.

[20] Nicolli Rios, Leonardo Mendes, Cristina Cerdeiral, Ana Patrícia F. Magalhães, Boris Pérez, Darío Correal, Hernán Astudillo, Carolyn Seaman, Clemente Izurieta, Gleison Santos, and Rodrigo Oliveira Spínola. 2020. Hearing the Voice of Software Practitioners on Causes, Effects, and Practices to Deal with Documentation Debt. In *26th Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer.

[21] Nicolli Rios, Manoel G Mendonça, Carolyn Seaman, and Rodrigo O Spinola. 2019. Causes and Effects of the Presence of Technical Debt in Agile Software Projects. (2019).

[22] Nicolli Rios, Rodrigo Oliveira Spínola, Manoel G de Mendonça Neto, and Carolyn Seaman. 2019. Supporting analysis of technical debt causes and effects with cross-company probabilistic cause-effect diagrams. In *Proceedings of the Second International Conference on Technical Debt*. IEEE Press, 3–12.

[23] Nicolli Rios, Rodrigo Oliveira Spínola, Manoel Mendonça, and Carolyn Seaman. 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*. ACM, 39.

[24] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14, 2 (2009), 131.

[25] Carolyn Seaman and Yuepu Guo. 2011. Measuring and monitoring technical debt. In *Advances in Computers*. Vol. 82. Elsevier, 25–46.

[26] Gayathri Subrahmanyam and Swati Seela. 2019. The Ultimate Guide To Risk Based Testing: Risk Management In Software Testing. https://www.softwaretestinghelp.com/risk-management-during-test-planning-risk-based-testing/

[27] FluidSurveys Team. 2014. *3 Types of Survey Research, When to Use Them, and How they Can Benefit Your Organization!* http://fluidsurveys.com/university/3-types-survey-research-use-can-benefit-organization/

[28] Roberto Verdecchia. 2018. Architectural Technical Debt Identification: Moving Forward. In *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 43–44.

[29] William Webber, Alistair Moffat, and Justin Zobel. 2010. A Similarity Measure for Indefinite Rankings. *ACM Trans. Inf. Syst.* 28, 4, Article 20 (Nov. 2010), 38 pages. https://doi.org/10.1145/1852102.1852106

[30] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer Science & Business Media.

[31] Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander. 2014. The sources and approaches to management of technical debt: a case study of two product lines in a middle-size finnish software company. In *International Conference on Product-Focused Software Process Improvement*. Springer, 93–107.

[32] Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander. 2015. The benefits and consequences of workarounds in software development projects. In *International Conference of Software Business*. Springer, 1–16.

[33] Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander. 2016. How do software development teams manage technical debt?–An empirical study. *Journal of Systems and Software* 120 (2016), 195–218.

[34] Jesse Yli-Huumo, Tommi Rissanen, Andrey Maglyas, Kari Smolander, and Liisa-Maija Sainio. 2015. The relationship between business model experimentation and technical debt. In *International Conference of Software Business*. Springer, 17–29.