Information and Software Technology 140 (2021) 106692

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Technical debt payment and prevention through the lenses of software architects

Boris Pérez [a,b,*], Camilo Castellanos [a], Darío Correal [a], Nicolli Rios [c], Sávio Freire [d,e], Rodrigo Spínola [f], Carolyn Seaman [g], Clemente Izurieta [h,i]

[a] Universidad de los Andes, Bogotá, Colombia
[b] Univ. Francisco de Paula Santander, Cúcuta, Colombia
[c] Federal University of Rio de Janeiro, Rio de Janeiro, Brazil
[d] Federal University of Bahia, Salvador, Brazil
[e] Federal Institute of Ceará, Morada Nova, Brazil
[f] Salvador University, Salvador, Brazil
[g] University of Maryland, Baltimore MD, United States
[h] Montana State University, Bozeman MT, United States
[i] Idaho National Laboratories, Bozeman MT, United States

ARTICLE INFO

Keywords:
Technical debt
Technical debt management
InsighTD
Payment practices
Preventive actions
Software architects

ABSTRACT

Context: Architectural decisions are considered one of the most common sources of technical debt (TD). Thus, it is necessary to understand how TD is perceived by software architects, particularly, the practices supporting the elimination of debt items from projects, and the practices used to reduce the chances of TD occurrence.
Objective: This paper investigates the most commonly used practices to pay off TD and to prevent debt occurrence in software projects from the architect's point of view.
Method: We used the available data from InsighTD, which is a globally distributed family of industrial surveys on the causes, effects, and management of TD. We analyze responses from a corpus of 72 software architects from Brazil, Chile, Colombia, and the United States.
Results: Results showed that refactoring (30.2%) was the main practice related to TD payment, followed by design improvements (14.0%). Refactoring, design improvements, and test improvements are the most cited payment practices among cases of code, design and test debt. Concerning the TD preventive practices, we find that having a well-defined architecture and design is the most cited practice (13.6%), followed by having a well-defined scope and requirements. This last practice is the most cited one for expert software architects. Finally, when comparing preventive practices among the three major roles derived from the survey (software architects, engineer roles, and management roles), we found that none of the roles shared the most cited practice, meaning that each role had its worries and focus on different strategies to reduce TD's presence in the software.
Conclusion: The lists of TD payment and prevention practices can guide software teams by having a catalog of practices to keep debt controlled or reduced.

1. Introduction

Tight schedules and deadlines are common conditions faced by software companies when delivery of software in faster cycles is required. These conditions increase the pressure for the development teams to deliver working features to their customers [1]. Additionally, the onset of continuous integration approaches as well as DevOps has contributed to this problem [2]. Technical debt (TD) is a metaphor used by the software community to describe technical decisions that can give companies a benefit in the short term [3] but possibly hurt the overall quality of the software and the productivity of the development team in the long term [4]. Intentional TD injection during software development is a common practice for software teams because it can help to achieve the project's goals sooner or more cheaply. However, this TD could pose risks to projects if it becomes difficult to manage [5], such as financial and technical problems linked to software maintenance and evolution costs [6].

* Corresponding author at: Universidad de los Andes, Bogotá, Colombia.
E-mail address: br.perez41@uniandes.edu.co (B. Pérez).

https://doi.org/10.1016/j.infsof.2021.106692
Received 29 December 2020; Received in revised form 21 May 2021; Accepted 18 July 2021
Available online 22 July 2021

According to Ernst et al. [7], architectural decisions are the most important source of TD. Therefore, it becomes crucial to understand how TD is managed by those who make architectural decisions, i.e., software architects: the practices used to pay off the debt introduced in software projects, and the practices performed by them to avoid or reduce TD occurrence in software projects. Investigating payment and preventive practices solely from the perspective of software architects is a compelling argument because they are responsible for critical decisions that affect the longevity of software products. These two practices (payment and prevention) are important because they are related to each other: debt prevention can be better and cheaper for the development team than incurring debt and paying it off later [8]. In other words, TD prevention also supports other TD management activities by reducing not only payment activities but also monitoring or identification activities [1]. Prevention supports the implementation of the optimal solution right from the beginning without potential interest payments. Also, catching TD early by architects facilitates implementation phases for practitioners. TD preventive actions can support the development team to minimize the occurrence of debt [9]. Therefore, TD prevention is worth additional consideration [8]. These two practices (payment and prevention) are of special interest to the software community because knowing the current practices adopted by practitioners can provide initial guidance for software teams on how to prevent or pay off debt items.

Despite the attention surrounding TD by both industry and academia, there is a lack of empirical evidence about both the payment-related and preventive practices used by software architects in real-life software projects [10,11]. There are, however, two studies close to ours: a study focused on TD payment practices [12], and a study focused on TD preventive practices [9]. The former [12] discusses an analysis of the most cited TD payment practices considering the size and age of systems, followed by an analysis of how TD causes can be associated with TD practices. The analysis includes 432 professionals from Brazil, Chile, Colombia, and the United States. The latter [9] discusses an analysis of the preventive actions that can be used to avoid the occurrence of TD and the impediments that hamper the application of these actions. This analysis includes 207 professionals from Brazil and the United States. Both studies focused on the point of view of all software practitioners' roles including developers, testers, project managers, software architects, among others. Our study focused only on the perspective of software architects. Also, differences between populations lead to varying results among these studies.

The goal of this study is to investigate the practices performed on TD payment and on TD prevention from the point of view of software architects in real-life software systems projects. It is important to note that TD payment/preventive practices were reported without knowing how much debt was paid off nor the level of success of the practice, therefore, this study does not make any assumptions on how optimal a practice is. This study focused on payment-related practices of the debt that practitioners were aware of, and not on whether the debt was intentionally introduced. This study uses data from Brazil, Chile, Colombia, and the United States, collected by the InsighTD Project, which is a globally distributed family of industrial surveys on the causes and effects of TD [13]. A total of 72 software architects from the countries mentioned responded to the survey. This study analyzes this data through qualitative and quantitative strategies: first, we characterize the study participants, and then, qualitatively analyze the payment and preventive practices cited by them.

Results show that *refactoring* and *improve design* (19 citations altogether) are the most cited TD payment practices used by software teams, as reported by software architects. These practices were expected to be at the top of the list [1,10,14,15], considering that both practices are intertwined. Lists of payment practices among design, code, and test debt tend to be more similar as more practices are included. We find that 6 out of 7 payment practices are shared between code and design debt, and only three practices are shared among all three TD types: *refactoring*, *improve design*, and *improve testing*. Finally, software architects reported the most common causes of TD occurrence, and when they were mapped against the payment practices we found that *refactoring* is the most used practice for TD payment in 5 out of 9 most cited TD causes.

Related to the TD preventive practices, we found that *well-defined architecture/design* and *well-defined scope/requirements* were the most performed practices. Software architects understand the relationship between architectural design and TD. Also, we found that *well-defined scope/requirements* was the most performed preventive practice for expert group, *code evaluation/standardization* for proficient group and, *well-defined architecture/design* for both competent and beginner groups. Finally, when comparing the preventive practices among software architects, engineer roles, and management roles, we found that all three roles share more than 50% of the performed preventive practices. This is expected considering how close they are to the development process. However, all three roles are different in their first 5 most cited practices.

Software practitioners can benefit from the results of this study by using the list of practices related to TD payment used in industry to support initial efforts to understand their debt (by considering the most cited TD causes) and to pay it off in their software projects. Also, software practitioners could review the list of TD preventive practices (commensurate with their level or expertise) and use it as a guide to improve their software development process. The global family of surveys allows practitioners to evaluate their own TD situation against overall industrial trends. For researchers, our results support future research by providing insights into software architects' perspectives on practices related to TD payment and TD prevention.

The contributions of this work are two-fold. First, an analysis of the most used TD payment-related practices (refactoring being the most cited) is presented. In addition, a comparison of the similarity of the payment practices according to the three major TD types cases found in our study is presented, together with a heatmap of the most cited practices related to TD payment against the most cited TD causes reported by software architects. Second, an analysis of the most performed TD preventive practices is presented, together with an analysis of these practices according to the experience of the architects, and in comparison with management roles and engineering roles.

The rest of the paper is structured as follows: Section 2 presents a description of the InsighTD project history altogether with a review of TD concept. Section 3 presents the survey design. In Section 4, we present the results of our analysis. Discussion of our results are presented in Section 5. Section 6 presents a review of studies similar to this one. And finally, in Section 7, we present threats to validity, and conclude the paper in Section 8.

2. Background

2.1. InsighTD project

InsighTD is a globally distributed family of industrial surveys initiated in 2017 and focused on organizing an open and generalizable set of empirical data on the state of practice and industry trends in the TD area. The InsighTD's survey includes questions covering (i) the characterization of the participants and their respective organizations, (ii) the understanding of the TD concept, (iii) the identification of possible causes and their possible effects on software projects, and (iv) how software development teams react (e.g. monitor, pay off, or define preventive actions) to debt items in their projects. 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. The rationale behind country-level replication as the scope is twofold: i. Organizing the work and making the dissemination of the survey wider by utilizing the local industry contacts of a large set of researchers, and ii. Investigating whether differences in local development practices could influence how participants experience the TD concept.

Currently, several studies have been conducted to analyze this information, such as a discussion on the top 10 causes and effects of TD [13], causes and effects of TD in agile software projects [16], reaction to the presence of debt in the Chilean software industry [17], documentation debt in software projects [18], among others [9,19,20].

2.2. Technical debt (TD)

The most recent definition of TD can be found in the Dagstuhl seminar report 16162 [21]: “In software-intensive systems, technical debt 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. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability”. As stated by McConnell in [22], internal quality attributes are characteristics that a user of the software product is not aware of, for example, maintainability, flexibility, portability, reusability, readability, and testability.

TD is composed of a debt, interest and principal. Debt is the term used to describe the gap between the existing state of a software and some “ideal and optimised” state [23]. Interest is related to the extra effort in maintaining the system due to the presence of TD. And principal is related to the effort required to address (refactoring) the TD and lead the system to an optimal level of quality [24,25].

In the systematic mapping study by Li et al. in [10], they classified TD in 10 different types as follow: design, architecture, code, documentation, test, defect, requirement, infrastructure, build, and versioning debt. In [15], Alves et al. identified four more TD types: people, process, service, and usability debt. Kruchten et al. in [3] stated that architecture, documentation and testing can add significantly to the debt and thus are part of the TD landscape. Also, Holvitie et al. in [26] found that inadequate architecture and inadequate documentation were the most frequently causes of TD. According to Ernst et al. in [7], architectural decisions are the most important source of TD.

3. Research method

This section presents the research questions posed in this work, and discusses its data collection (survey research method) and data analysis procedures.

3.1. Research questions

The goal of this study is to identify and analyze the practices performed to pay off TD and to prevent TD occurrence from the point of view of software architects in real-life software systems projects. Based on this goal, we derived the following Research Questions (listed below) guiding our study, the reporting of the results, and the knowledge contribution to the TD community. In the list, we also present the mapping between the RQs and the questionnaire Questions (Q) described in Section 3.2.

RQ1: From a software architect’s point of view, what are the practices for TD payment used by software development teams? This RQ explores the practices used by software teams to pay off or to support the payment of TD items that practitioners were aware of, regardless of the intentionality of the injection. This research question is not related to a practice used by the software architects to pay off the TD, but to an action taken by the software development team at any specific time, to pay off the TD. It is important to mention that this question focuses on the practice used, without considering the amount of TD paid off, or whether the practice was successful or not. Questions 26 and 27 are used to answer this RQ.

RQ1.1: Is it possible to find similarities of the TD payment practices according to the TD type associated with them? This RQ focuses on discovering if the type of TD (Section 2.2) has an impact on the practice used to pay off TD. This is important because software teams can define a set of practices according to the type of TD facing at that moment. Q13 and Q27 are used to answer this RQ.

RQ1.2: Is it possible to establish an association between main causes leading to TD occurrence and main practices related to TD payment? This RQ focuses on identifying any existing relationship between causes leading to TD and how software teams paid off these TD cases. Grouping TD payment practices based on what cause the TD injection can support software teams to adopt a set of most used practices of eliminating debt items. Q16 to Q18, and Q27 are used to answer this RQ.

RQ2: From a software architect’s point of view, what practices have been performed to prevent TD occurrence? This RQ explores the main practices performed by software architects to prevent TD occurrence in software projects. These practices are based in the personal opinion and experience of the respondents. Q28 is used to answer this RQ.

RQ2.1: Does the software architect’s level of expertise influence the preventive practices performed by them? This RQ seeks to discover if there are preventive practices that are common regardless of the level of experience of the architects. Q7 and Q28 are used to answer this RQ.

RQ2.2: Are preventive practices performed by software architects different from the ones performed by other software development roles? This RQ was formulated to better understand how TD prevention is perceived by software architects, in comparison with other roles. We decided to compare the list of preventive practices performed by them against the ones performed by management roles and engineer roles. This analysis allows us to understand if it is possible to establish a set of preventive practices accepted by all roles, or if each role presents its own set of preventive practices. Q6 and Q28 are used to answer this RQ.

3.2. Data collection

The data was collected in the context of the InsighTD project, using an online questionnaire (Google Forms). This allowed us to reach a greater number of participants. Invitations were sent online only to software practitioners. The sending of the questionnaire followed a defined invitation procedure[1]: an invitation explaining the purpose of the survey and the people involved was sent to participants. A reminder was sent to participants one month later. Due to the anonymous nature of the survey, reminders were sent to everyone. This protocol was similar in all replications.

This survey can be classified as both exploratory and descriptive research. It is exploratory research because it focuses on the discovery of ideas and insights [27]. Open-ended questions are commonly used in this kind of research. It is descriptive research because it is preplanned and structured in design and the information collected can be statistically inferred on a population [27]. Respondents were asked to describe a specific past experience related to their participation in a software system project [28]. This past experience can be placed at different points of time, therefore, this survey cannot be considered

[1] https://github.com/borisrperezg/TDPaymentAndPreventivePractices-Protocol.

Table 1
Survey questions.

No.	Question	Type
Q1	What is the size of your company?	Closed
Q2	In which country you are currently working?	Closed
Q3	What is the size of the system being developed in that project? (LOC)	Closed
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?	Closed
Q7	How do you rate your experience in this role?	Closed
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
Q16	What was the immediate, or precipitating, cause of the example of TD you just described?	Open
Q17	What other cause or factor contributed to the immediate cause you described above?	Open
Q18	What other motives or reasons or causes contributed either directly or indirectly to the occurrence of the TD example?	Open
Q26	Has the debt item been paid off (eliminated) from the project?	Closed
Q27	If yes, how? If not, why?	Open
Q28	Considering your personal experience with TD management, what actions have you performed to prevent its occurrence?	Open

cross-sectional. Also, considering the online nature of the survey, respondents could answer the survey at any time, avoiding researchers to intervene while participants filled up the survey.

The InsighTD questionnaire consists of 28 questions [13]. Table 1 presents the subset of the survey's questions related to the context of this work. Questions Q1 to Q8 capture the characterization questions. For Q6, the options available are: business analyst, dba/data analyst, developer, process analyst, project leader/project manager, requirements analyst, software architect, and test manager/tester. Also, option Other is included to allow participants to enter a different role. Related to Q7, participants defined their level of experience in their role among the following options: Novice (Minimal or "textbook" knowledge without connecting it to practice), Beginner (Working knowledge of key aspects of practice), Competent (Good working and background knowledge of area of practice), Proficient (Depth of understanding of discipline and area of practice), and Expert (Authoritative knowledge of discipline and deep tacit understanding across area of practice).

Closed questions in Table 1 refer to multiple-choice closed questions. This type of question can only be answered by selecting from a limited number of options. Some of the closed questions include a free text option (e.g., other) so that the participants can express their opinion more appropriately. This set of closed-ended questions is mainly used for the characterization questions (Q1 to Q8). Question 26 is a yes/no closed-ended type question. Available options for each of these questions are reported in Fig. 1 (Section 4.1). The set of available options for the questions was selected based on the experience of the Core Team (CT) of InsighTD and the steps described in the validation phase presented in [13].

Question 13 asks participants to describe a case of a TD item that had a significant impact on their software project (this case is used as a basis for answering follow-on questions). In this question, the impact could be measured in labor hours required to pay off the debt, or the impact on maintainability, or possible delays for future releases. Any measure that the participant considered relevant to describe the impact of the TD item.

Also, it is important to note that it could be possible to expect subjects not to be familiar with the TD concept. To mitigate this scenario, participants were asked about their familiarity with the concept of Technical Debt (Q9). Then, a TD definition adapted from McConnell [29] was included before Q11. Question Q11 asks participants to indicate how close was this definition to the understanding of TD of the participant (5-point Likert scale), and also if there were parts of this definition that the participant disagrees with (Q12). These three questions (Q9, Q11 and, Q12) are not listed in Table 1.

Participants describe a case of a TD item in Q13. This case of a TD item was then classified according to the TD types (code debt, design debt, etc.) presented in Section 2.2. This classification is required in order to answer our RQ1.1 (Is it possible to find similarities of the TD payment practices according to the TD type associated with them?). Each of the example cases given by the respondents followed a mapping process between the overall description and TD indicators from Alves et al. [15]. For example, "Update pending documentation" was an answer for Q13, and it has a direct mapping with the TD indicator *outdated documentation*. This TD indicator is associated with documentation debt, therefore, this example case was classified as documentation debt. This classification process was done independently for each replication, and could have been done by one or all of the researchers in each replication.

Questions Q16 to Q18 ask participants to describe the causes leading to the occurrence of TD in the case presented in Q13. Questions Q26 and Q27 ask participants to answer if the TD item (from Q13) was eliminated and how. Question 27 asks the participant to answer how the debt was paid off, however not further instructions were given to the respondent, such as the amount of debt actually paid off or by whom the payment practice was performed. Also, respondents did not include any information beyond the practice used. Finally, Q28 asks participants about their personal opinion about practices performed by them to prevent TD in their software systems.

3.2.1. *Instrument validation*

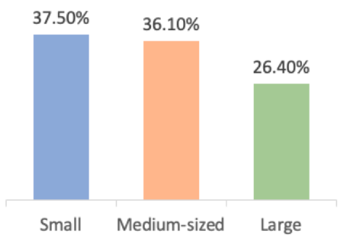
Three main steps were required to validate this survey, as presented in [13]: internal validation, external validation, and pilot study [13]. Internal and external validations were completed in a period of six months. Both validations were done to ensure that questions were clearly interpretable and sufficiently complete to answer the original research questions [13]. A pilot study was performed to ensure that the questionnaire was well understood by a small number of participants representing the target population of the study. Additional validation activities were performed in countries where questions had to be translated to their corresponding native language.

3.2.2. *Selection of participants*

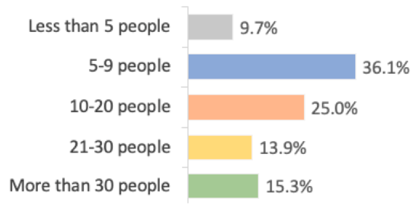
The selection of participants was done based on their role in the software industry. The following roles were considered: configuration analyst, configuration manager, developer, process analyst, product owner, programmer, project manager, requirements analyst, software architect, software engineer, system analyst, test analyst, test manager, and tester. Three main invitation channels were used to reach the target population (software practitioners): the social media platform LinkedIn, mailing lists, and industry partners. LinkedIn gave us direct access to a large number of professionals with whom we did not have previous contact. This process has been applied in all replications (Brazil, Chile, Colombia, and the United States).

3.3. *Data analysis*

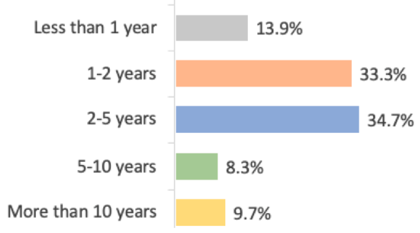
During the data analysis stage, only four countries had finished the data gathering stage: Brazil, Chile, Colombia, and the United States, and therefore, these countries were selected in this paper. Also, we only selected answers where participants chose *Software Architect* as their role in a software project (Q6). Finally, according to RQ1, this study



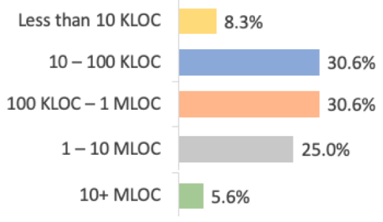
(a) Distribution per company size



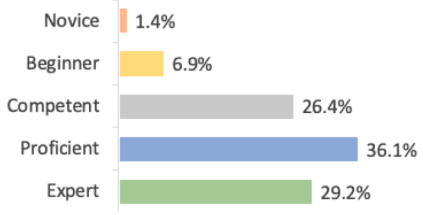
(b) Distribution per team size



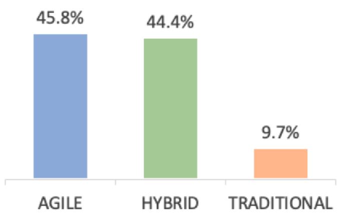
(c) Distribution per system age



(d) Distribution per system size



(e) Distribution per level of experience



(f) Distribution per process model

Fig. 1. Characterization of participants.

focused only on answers where participants reported that the TD was paid off (answered *yes* to Q26) , without considering the amount of TD paid off, or the level of success of the payment. Negative answers were not considered as a part of this study.

The survey instrument is composed of a mix of closed and open questions. For closed-ended questions, we used descriptive statistics to get a better understanding of the data. For open-ended questions, answers were codified using a code schema provided from the InsighTD project.[2] Manual open coding was initially applied resulting in a set of codes. The process was performed iteratively revising and unifying codes at each cycle of analysis until reaching 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. For example, two participants cited the following preventive practices in raw form: "... Educate the different participants of the project about the implications of TD. Education is key", and "... I've also tried to track the TD items, classify them, and estimate their TTF (time to fix)". We initially coded these two chunks with "raising awareness of the debt", and "implementation of a TD management strategy", respectively. Then, we could identify these two examples as different nomenclature for the same preventive practice. Finally, we unified the names of sets of preventive practices using the most commonly used term in that subset, which was "td awareness/management" in this example. After repeating these steps on the whole data set we had the final list of preventive practices.

The coding process was performed by at least three researchers in each of the four InsighTD replications. Each researcher could assume one or two of the following roles: (i) *code identifier*, responsible for reviewing the answers and extracting the corresponding codes, (ii) *code reviewer*, responsible for reviewing and joining all extracted codes, and (iii) *referee*, responsible for resolving disagreements in codes identified by the code identifier and code reviewer. For example, for the preventive practice "specialized training in development patterns", two researchers identified two different codes: "training" and "appropriate use of design pattern". In this case, the referee decided to focus on the final goal of the practice which is the appropriate use of design patterns, and therefore, the code "appropriate use of design pattern" was selected.

The data-gathering stage was done in 2017 for Brazil, 2019 for both Chile and Colombia, and 2018 for the United States. Some of these data may seem quite old, but practices and actions described by software architects are related to the what and not about the how. For example, refactoring (the *what*) could be done by several means (the *how*) such as using external tools to automating refactoring, or by doing some small changes in the code. Therefore, the practice could remain updated no matter how it was performed. In the same vein, a preventive action such as having well-defined requirements (the *what*) can be done in multiple ways (the *how*).

4. Results

This section presents the practices used to pay off TD, as reported by software architects, and the practices performed by software architects to prevent TD injection into software projects.

4.1. Characterization of the respondents

As a result of the joint effort of the participating countries, 427 responses were obtained. Of these responses, we found 72 software architects, distributed as follows: 10 (13.9%) from Brazil, 16 (22.2%) from Chile, 28 (38.9%) from Colombia, and 18 (25%) from the United States. Fig. 1 summarizes the distribution of the survey participants

[2] https://github.com/borisrperezg/TDPaymentAndPreventivePractices-Protocol.

Table 2
Most cited practices related to TD payment.

Practice related to TD payment	#CP	%CP
Refactoring	13	30.2%
Improve design	6	14.0%
Adoption of good practices	3	7.0%
Budget increase	3	7.0%
Improve testing	3	7.0%

per company size, team size, system age and size, level of experience, and process model. We notice that the participants are well distributed among the different company sizes, most of them working in teams composed of 5 to 9 practitioners following mostly hybrid or agile processes. Further, systems with sizes mostly from 10 KLOC to 100 KLOC and 100 KLOC to 1 MLOC, and aged between 1 and 2 years, and 2 and 5 years are the most commonly mentioned among the surveyed practitioners. Finally, software architects see themselves as proficient regarding their level of experience, indicating that, in general, the questionnaire was answered by professionals with experience in their functions.

We are aware that these participants do not represent all the software architects in the software industry from Brazil, Chile, Colombia, and the United States. However, participants are characterized by representing a broad and diverse audience reaching different levels of experience, different sizes of organizations, and projects of different ages and team sizes.

4.2. From a software architect's point of view, what are the practices related to TD payment used by software development teams? (RQ1)

To answer this RQ, we used question Q26 to select only answers where TD was paid off: 31 answers (43.1%). Then, we used the answers for question Q27 to extract the specific practice used to pay off the TD. We excluded answers where no payment practice was described. In the end, 27 valid responses from software architects were used to answer RQ1. It is worth mentioning that TD payment practices were reported without knowing how much debt was paid off nor the level of success of the practice, therefore, this study does not make any assumptions on how optimal a practice is.

We identified 16 TD payment-related practices. Some of the respondents described more than one practice, and therefore, they were coded and counted. For example, one respondent cited three practices: *improve design*, *refactoring*, and *code reviewing*.

Based on the 16 practices related to TD payment, we selected the top 5 most cited practices and present them in Table 2. Only practices cited more than two times are presented, and they represent 65.1% of the set of all identified practices. Table 2 presents the TD payment-related practices, the total number (i.e., count) of times the practice (#CP) was cited, and the percentage of #CP in relation to the total of all cited practices (%CP).

Refactoring, as presented in Table 2, is the most cited practice used to pay off TD (13 citations) by software teams, as reported by software architects. The reported practice was not an unexpected result [1,10,14,15]. In the context of this study, refactoring means to refactor the source code of a system. Some examples of answers given by the respondents are: "Engineers taking the time to eliminate it during new feature development or taking it upon themselves to refactor a messy bit of code", "Performing the necessary refactoring" and "Code restructuring (refactor)". As can be seen, answers about the payment practices used to pay the debt were straightforward.

The second TD payment practice is *improve design*. This practice is related to the changes in the system architecture. Changes in the architecture will be performed in the code later, so, *refactoring* and *improve design* are intertwined, and therefore, it is unsurprising that this practice (*improve design*) ranks as one of the first two practices listed to remove the debt. Some examples of answers given by the respondents are: "The identified tech debt has been resolved through updated designs and refactors" and "refactoring and change of architecture".

In [30], the authors found that some TD payment-related practices do not directly allow the elimination of TD items. For instance, the practice *adoption of good practices* is more related to preventive practices to avoid (reduce) TD occurrences but does not eliminate the item by itself. The authors [30] indicated the existence of four types of practices related to TD payment: payment practice, defining a favorable setting for TD payment, TD prevention, and TD prioritization. We also have such kinds of cases in our results. The following three practices in our rank are sorted alphabetically since they share the same number of citations (3 citations): *adoption of good practices*, *budget increase*, and *improve testing*. *Adoption of good practices* and *improve testing* are more related to preventive practices. On the other hand, *budget increase* could be a consequence of having too much TD or a requirement to pay TD off. Also, *budget increase* is the only practice related to the management level of software development. Quotes of these three practices were very explicit: "Benefits of a shared library cutting down on dev time duplicating or maintaining extra code", "The cost was implicitly assumed", and "Defining hours of testing by the development team".

Other practices not listed in Table 2 (with two citations) are: *code reviewing*, *extra effort*, *improve communication*, *incremental payment*, and *use of external tools*. *Extra effort* and *incremental payment* could be both related to refactoring. Extra effort was required to pay the debt, and an incremental payment was used to pay TD off. However, answers were not explicit about the payment practice itself. *Code reviewing*, *improve communication*, and *use of external tools* could be related to preventive practices to avoid or reduce TD occurrences.

4.3. Is it possible to find similarities of the TD payment practices according to the TD type associated with them? (RQ1.1)

This RQ focused on measuring quantitatively how similar the list TD payment practices are for each of the following types of debt: code debt, design debt, and test debt. To accomplish this task, we first investigate the type of TD based on the example case reported by respondents in Q13, and then, the list of practices (ranked by citations) according to each of the TD types was extracted. The identification of the type of TD based on the example case was presented in Section 3.2. The three main types of debt are code (32%), design (24%), and test debt (16%). Code debt refers to the problems found in the source code that can negatively affect the legibility of the code increasing maintainability efforts [15]. Design debt refers to violations, at the source-code level, of the principles of good object-oriented design (e.g. very large or tightly coupled classes) [31]. Test debt refers to shortcuts taken in testing, such as lack of tests (e.g., unit tests, integration tests, and acceptance tests) [10].

To quantitatively measure how similar the lists of TD payment practices are to each other, we used the rank-biased overlap (RBO) similarity measure [32]. RBO compares two ranked lists and returns a numeric value between zero and one to quantify their similarity. An RBO value of zero (0) indicates the lists are completely different, and an RBO of one (1) means completely identical or overlapping. RBO was used because it supports top-weighted ranked lists, the 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 p value used in RBO represents the number of elements to compare, for example, $p = 0.5$ corresponds to a comparison of the first two elements approximately, and $p = 0.97$ corresponds to a comparison

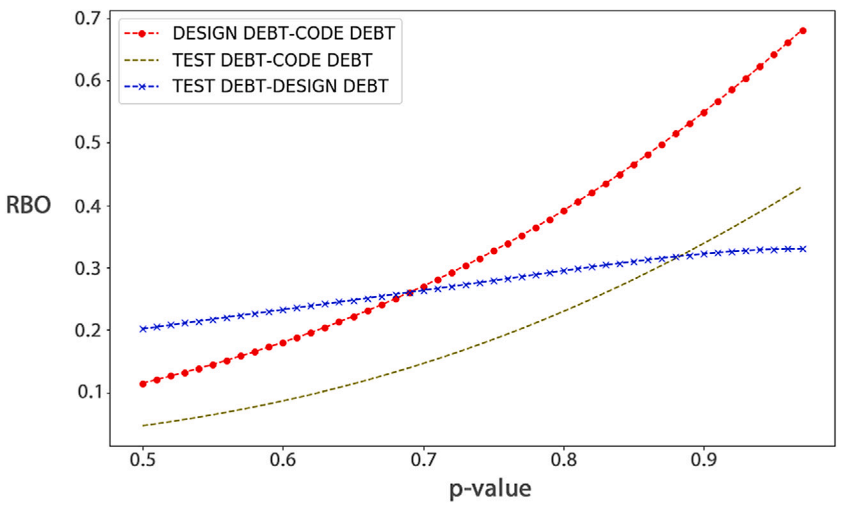


Fig. 2. RBO analysis of TD payment practices ranks per TD type.

Table 3
TD payment practices per TD type.

Payment practice	Code debt	Design debt	Test debt
Refactoring	8	1	1
External tools	2	–	–
Adoption of good practices	1	2	–
Code reviewing	1	1	–
Improve design	1	2	1
Improve testing	1	1	1
Incremental payment	1	1	–
Budget increase	–	1	–
Extra effort	–	1	–
Tech independent implement.	–	1	–
Backlog inclusion	–	–	1
To make defect free system	–	–	1

of all elements approximately. 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.

Fig. 2 depicts the pairwise RBO comparisons among the three lists of practices related to TD payment (one line for each pair of TD types). This figure presents a visual comparison of the similarity of practices. RBO comparison went from $p = 0.5$ to $p = 0.97$.

Overall, the RBO comparison indicates that the similarity level concentrated on the very initial practices of the ranks is only around 15%. This value increases as we consider more practices from all the ranks. However, the difference of perceptions in the very beginning of the ranks is very significant because these practices have the biggest citation values, which means that they have a large use as perceived by the software architects.

We can also notice that the payment practices from design debt and code debt tend to be more similar as more practices are included. At $p = 0.5$, the RBO value for the pair design debt-code debt was 0.11, and then, at $p = 0.97$, the RBO value increased to 0.68. We found 7 practices for cases of code debt, and 6 out of 7 practices are shared with design debt.

The pair design debt-test debt presents some degree of similarity. They both have *improve design* in second position, and also share *improve testing* and *refactoring*. For the test debt-code debt pair, it can be seen that they do not share practices at the first two elements, but then, as more practices are included, more similarity tends to show (RBO = 0.43).

The list of practices per TD type is presented in Table 3. This Table lists the TD payment practices and the number of times it was cited according to the TD type derived from Question 13.

As can be seen, it is possible to note that only three practices are shared among the three TD types: *refactoring*, *improve design*, and *improve testing*. There are also cases of particular practices per type of TD, for example, cases of code debt reported the practice *(use of) external tools*, and cases of test debt reported the practices *backlog inclusion* and *to make defect free system*. For design debt, there are three practices only described for this TD type: *budget increase*, *extra effort*, and *technology independent implementation*.

4.4. Is it possible to establish an association between main causes leading to TD occurrence and main practices related to TD payment? (RQ1.2)

To answer this RQ, we took the nine (out of 21) most cited causes leading to TD occurrence reported by software architects together with an associated heatmap against the most cited TD payment-related practices (Fig. 3). We took the causes leading to TD occurrence that were cited 4 or more times. This was done to prevent the heatmap from being scattered, considering that there are cases that were cited only once.

The selected nine causes for the analysis represent 73.6% of the total of causes identified. The list of causes was gathered joining answers from questions Q16, Q17, and Q18. Then, a codification process was performed to assign codes to the answers, as described in Section 3.3. The list of causes in Fig. 3 includes the number of times they were cited.

From Fig. 3, it is possible to see that *non-adoption of good practices* is the most cited cause leading to TD occurrence, followed by *inappropriate planning* and *lack of qualified professionals*. The first and third causes have a technical inkling, while the second one, has a management inkling. In total, there are four causes with management and five with technical inkling. So, from the point of view of software architects, technical and management issues are almost equally decisive for the occurrence of debt items.

Refactoring is the most selected practice used for TD payment in 5 out of 9 causes. *Non-adoption of good practices* had the highest number of citations for *refactoring* (9 citations), followed by *lack of qualified professionals* (5 citations). On the other hand, *not effective project management* had the lowest number of citations for *refactoring* (1 citation). Other practices cited for this cause are: *improve testing*, *incremental payment*, and *improve communication*. This means that dealing with problems at the management level requires a different set of practices to fix the problem.

Refactoring shares the number of citations for the following causes: *producing more with no quality*, *inadequate choice of technology/tool /platform*, *deadline*, and *not effective project management*. With respect to these first three causes, the other practices cited are: *improve design*

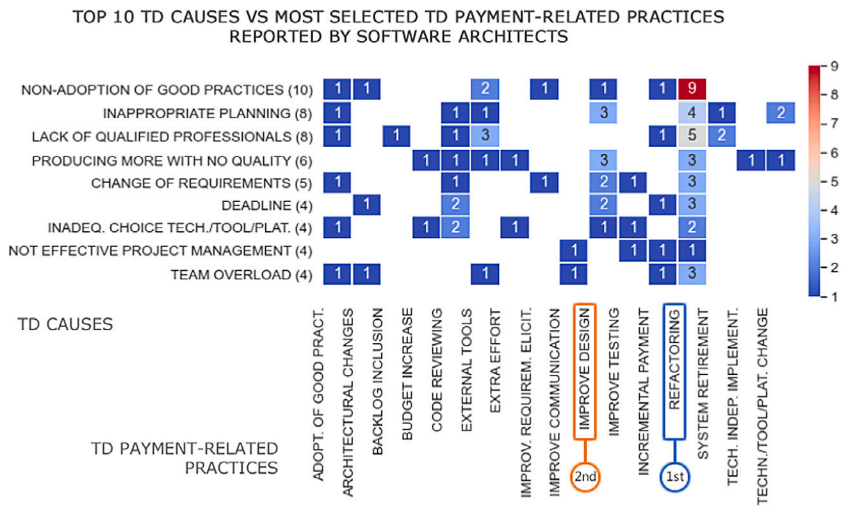


Fig. 3. Practices related to TD payment vs. the most cited causes.

Table 4
Top 9 practices to prevent TD occurrence.

Preventive practice	#	%
Well-defined architecture/design	14	13.6%
Well-defined scope/requirements	12	11.7%
Code evaluation/standardization	12	11.7%
TD awareness/management	10	9.7%
Adoption of good practices	9	8.7%
Better project management	8	7.8%
Improving tests/coverage	7	6.8%
Good communication on team	6	5.8%
Training (code review/refactoring)	4	3.9%

and *code reviewing*. *Improve design* is the second most cited practice for TD payment among all presented causes. This practice has the highest number of citations (after refactoring) for *inappropriate planning* and *producing more with no quality*.

Although we can observe some interesting behaviors in Fig. 3, overall, results allow us to conclude that *refactoring* is the main payment practice of development teams to pay off the debt no matter what caused it to be injected.

4.5. From a software architect's point of view, what practices have been performed to prevent TD occurrence? (RQ2)

To answer this RQ we used the answers for question Q28. Most participants cited multiple preventive practices when answering this question. For example, an experienced software architect answered "adoption of good practices and available documentation". In these cases, the researchers analyzed each practice separately and mapped them to their corresponding code.

After the coding process (described in Section 3.3), we identified 25 TD preventive practices, which sum up to 103 citations. Based on this list, we selected the practices with four or more citations and present them in Table 4. These 9 practices represent almost 80% of the set of all identified practices. Table 4 presents the TD preventive practice, the total number (i.e., count) of times the practice (#CP) was cited, and the percentage of #CP concerning the total of all cited practices (%CP). We can observe that *well-defined architecture/design*, *well-defined scope/requirements*, and *code evaluation/standardization* are the most cited preventive practices performed by software architects to minimize the occurrence of TD.

Well-defined architecture/design is the most frequently cited practice with 14 citations. In the context of this study, *well-defined architecture/design* is related to the appropriate use of architectural/design patterns, architecture review, and good practices in architecture design, as stated by the respondents. A good architecture would imply a reduction of TD in the system. Also, a well-defined architecture would depend on having enough understanding of the requirements and the business. This is the main input with which the architect must work. Therefore, the second most cited practice (*well-defined scope/requirements*) could be considered a precursor of good architecture and also, a real practice to prevent TD occurrence.

The third practice cited by the architects is *code evaluation/standardization*. This practice shares the same amount of citations as *well-defined scope/requirements* with 12 citations (11.7%). This practice can go hand in hand with the use of tools to perform a continuous inspection of the code quality to detect bugs, code smells, and security vulnerabilities.

TD awareness/management is the fourth most cited practice with 10 citations (9.7%). This practice encompasses the following sub-practices: *information about extra effort due to TD*, *use of tools for TD identification*, *awareness of TD*, *training on TD management*, and *implementation of a TD management strategy*. Being aware of the TD concept could support architects to be more cautious about whether or not to allow the debt to be injected.

Adoption of good practices with 9 citations (8.7%) is related to code development and covers the following sub-practices: *following a well-defined development standards* and *adoption of pair programming*. *Better project management* with 8 citations (7.8%) is the first non-technical practice in the list of practices presented in Table 4. This practice encompasses the following sub-practices: *well planned deadlines*, *project manager participation*, *adding time to estimate task*, among others. Having this practice in the sixth position indicates that software architects focus more on technical practices to prevent TD occurrence.

Improving tests/coverage with 7 citations (6.8%) is in seventh position. This practice is related to the improvement of the tests designed to check source code. This practice encompasses the following: *creation of automated tests* and *appropriate test coverage*. *Good communication on team* with 6 citations (5.8%) is the second non-technical practice to prevent TD occurrence. This practice is related with *good communication among stakeholders*, *improving of internal communication*, and *discussion about project improvements*. This practice implies having a role capable of communicating with the business side and with the technical side. Finally, *training (code review/refactoring)* with 4 citations (3.9%) is in ninth position. Training on code reviews could be useful to reduce TD occurrence. On the other hand, training on refactoring is more related to TD payment practices. However, both pieces of training were grouped.

The full list of preventive practices includes *quality control* (3 citations; 2.9%), *well-defined/available documentation* (3 citations; 2.9%), using continuous integration (2 citations; 1.9%), monitoring of the process (2 citations; 1.9%), and 11 more practices with 1 citation.

Table 5
Top 3 most cited practices for TD prevention by level of architectural experience.

Experience	Action for TD prevention	#CR	%CR
Expert	Well-defined scope/requirements	8	25.0%
	TD awareness/management	5	15.6%
	Better project management	4	12.5%
Proficient	Code evaluation/standardization	7	18.4%
	Well-defined architecture/design	5	13.2%
	TD awareness/management	5	13.2%
Competent	Well-defined architecture/design	4	19.0%
	Improving tests/coverage	3	14.3%
	Good communication on team	3	14.3%
Beginner	Well-defined architecture/design	2	18.2%
	Code evaluation/standardization	2	18.2%

4.6. *Does the software architect's level of expertise influence the preventive practices performed by them? (RQ2.1)*

To answer this RQ, we analyzed the relationship between the level of experience (via the answers to question Q7) and the preventive practices. Table 5 presents the most cited TD preventive practices as performed by expert, proficient, competent, and beginner software architects. The expert group consists of 18 architects, the proficient group consists of 22 architects, the competent group consists of 11 architects and the beginner group consists of 5 architects. The novice group was excluded because only one respondent was part of this group. Table 5 presents the name of TD preventive practice, the total number (i.e., count) of times the practice (#CP) was cited, and the percentage of #CP in relation to the total of all cited practices by group (%CP).

The first thing to note in Table 5 is the difference among all levels of experiences, each one having a different most cited TD preventive practice. The expert group has the most cited preventive practice among all groups: *well-defined scope/requirements* (8 citations). This group understands the importance of having clearly defined requirements to build a strong architecture. *Well-defined scope/requirements* was cited twice by the proficient group and once by both competent and beginner groups.

Well-defined architecture/design was cited by all groups. The expert group cited three times this practice. All groups understand that good architecture could prevent the presence of TD. However, only experts were clear about the importance of requirements. *TD awareness/management* was cited only by expert and proficient groups. These groups understand the importance of including a TD management strategy during the software development process.

Code evaluation/standardization was cited by all groups. The expert group cited this practice one time, in contrast to the proficient group which had seven citations for this practice. *Better project management* was cited four times by the expert group. Proficient and competent groups cited this practice twice. The beginner group cited it only once. Finally, *good communication on team* was cited three times by both expert and competent groups, and only once by the proficient group.

4.6.1. *RBO analysis*

We decided to use RBO to measure quantitatively how similar the TD preventive practices among the level of experience of software architects are. Fig. 4 depicts the pairwise RBO comparisons among the four lists of preventive practices (one line for each pair of levels of experience). Overall, although the RBO value tends upwards, the result of the analysis indicates that there are differences in how software architects with different levels of experience perceive the prevention of TD items.

Fig. 4 presents interesting results. TD preventive practices for Beginner–Proficient pair are the most similar among all groups. This similarity keeps the highest from $p = 0.5$ ($RBO = 0.79$) to $p = 0.97$ ($RBO = 0.69$). This means that the preventive practices performed by both groups are similar in the first two elements and keeps some similarity as more practices are included to compare. On the other hand, the Beginner–Expert pair exhibits the lowest similarity for $p = 0.5$ ($RBO = 0.045$). This means that the beginner group is closer to the proficient group but not to the expert group. However, the more practices are compared, the greater the similarity of this pair. At $p = 0.97$, the similarity increase to $RBO = 0.6$.

Another interesting result is the lack of similarity of the preventive practices performed by the expert group and the proficient group. When only two practices are compared the similarity is $RBO = 0.21$, and when all practices are compared, the similarity increase to $RBO = 0.58$. This means that when all practices are compared, only a little more than half of the practices are shared between these two groups.

Table 6
Consolidated top 5 practices to prevent TD occurrence by role.

Preventive practice	SA	EN	MA
Well-defined architecture/design	14 (1st)	19 (4th)	5
Well-defined scope/requirements	12 (2nd)	14	4
Code evaluation/standardization	12	15	3
TD awareness/management	10 (4th)	24 (3rd)	10 (2nd)
Adoption of good practices	9 (5th)	31 (1st)	6 (5th)
Better project management	8	28 (2nd)	11 (1st)
Good communication on team	6	18 (5th)	7 (4th)
Qualified professionals	2	1	8 (3rd)

4.7. *Are preventive practices performed by software architects different from the ones performed by other software development roles? (RQ2.2)*

To answer this RQ, we took the roles selected in Q6 and codified them into three big groups: management roles (manager, project manager, business analyst, QA manager, product owner, process analyst, TI manager), engineering roles (developer, tester, software engineer, database administrator, requirements analyst, infrastructure analyst, technical leader, data analyst), and software architects.

Table 6 presents the list of the 5 most cited TD preventive practices according to each of the three roles. It also shows the number of citations by each role and the relative position of the practice per role. The first 5 practices of the table belong to software architect (SA) roles and were extracted from Table 4 (Section 4.5). The following two practices (*better project management* and *good communication on team*) belongs to engineer roles (EN), and the last practice (*qualified professionals*) belongs to management roles (MA).

We can see that all three roles share this list of practices, but not in the same order. Even more, none of the practices share the same position in two or more roles, except by *adoption of good practices*, which share the 5th position in software architects and management roles.

Three of the top 5 practices are shared between software architects and engineer roles: *well-defined architecture/design*, *TD awareness/management*, and *adoption of good practices*. Only two practices are shared between software architects and management roles: *TD awareness/management* and *adoption of good practices*. However, things look different between engineer roles and management roles. Four practices are shared between these groups: *TD awareness/management*, *adoption of good practices*, *better project management*, and *good communication on team*.

Considering the number of citations of the preventive practices for all three roles, the practice *better project management* is the most cited one with 47 citations, closely followed by *adoption of good practices* (46 citations) and *TD awareness/management* (44 citations).

Although there are some similarities between the three groups, it is also possible to see specific practices for each role. For example, the practice *qualified professionals* seems to be important for management roles with eight citations (3rd position). However, this same practice was less cited by software architects (two citations) and technical roles

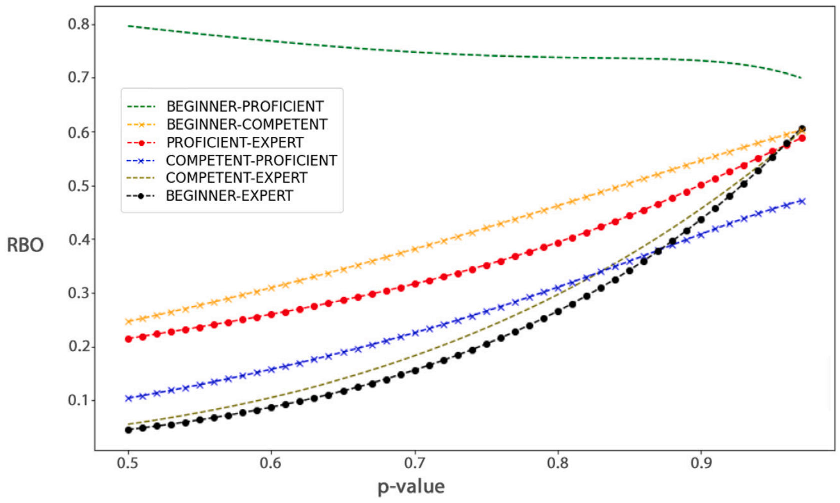


Fig. 4. RBO of practices related to TD prevention ranks per level of experience of software architects.

(one citation). On the other side, *code evaluation/standardization* was well cited by software architects and engineer roles and barely cited by management roles.

4.7.1. RBO analysis

Table 6 presented the top 5 most cited TD preventive practices per role, which seems similar for all three roles. From this list, we can conjecture that engineer roles and management roles are the most similar ones. However, there is a necessity to go deeper into this comparison by measuring quantitatively how similar all the cited TD preventive practices are among the three groups. Fig. 5 depicts the pairwise RBO comparisons among the three lists of TD preventive practices (one line for each pair of practices).

The first thing to note in Fig. 5 is that, at $p = 0.5$, the management–architect pair and architect–engineer pair had both an RBO value of 0.046 and 0.075, respectively. This means that the list of preventive practices from management roles and software architects has no common practices in their first two positions (approximately). The same happens for software architects and engineer roles. On the other hand, the management–engineer pair has an RBO value of 0.29 at $p = 0.5$, which is higher, but still small as a comparison value. This value occurs because the engineer and management roles share the practice *better project management* in their first two positions (see Table 6).

The second thing to note is that, at $p = 0.97$, the management–engineer pair and architect–engineer pair had both an RBO value of 0.64. This means that preventive practices from management roles and engineer roles share more than 50% of their practices. The same thing happens for software architects and engineer roles. management–architect had an RBO value of 0.56.

Overall, all three roles were different in their first 5 most cited practices. This means that every role has its worries and focuses on different strategies to reduce TD's presence in the software.

5. Discussion

This section discusses the results and presents their implications for both practitioners and researchers.

5.1. RQ1: From a software architect's point of view, what are the practices related to TD payment used by software development teams?

As presented in Section 4.2, *refactoring* is the most cited TD payment practice. As stated by Fowler [33], refactoring is a technique for improving the design of an existing code base through a series of small behavior-preserving transformations. On the other hand, a software team can opt to just write new code altogether, in other words, do the rewriting. This distinction is important because our survey artifact did not allow us to know if respondents were aware of the differences between these two practices to pay off TD. However, some respondents gave us some hints, for example: "Redoing the software". Other respondents were somehow ambiguous: "Through refactoring and APIs analysis". It was not clear which specific refactoring activities were performed during TD payment. It would be worthwhile to have a deeper codification scheme, however, the lack of deeper explanations by the respondents, dismiss this possibility.

According to Ernst et al. in [7], architectural decisions are the most common source of TD. The debt introduced early in the software (such as in the architecture), persists throughout the whole software lifecycle, becoming a major concern. Improving the architecture should be at least one of the first two practices to pay off the TD. *Improve design* ranks as one of the first practices (second) described by software architects as a practice implemented by software teams to remove the debt.

There are payment-related practices that would require a deeper analysis, for example, *budget increase*. Increasing the budget of the project will also mean that the software team would have to include new functionalities and not only pay off the debt. No client will pay for fixing code issues that the user will not see. It is important to remark that TD consequences are related to maintainability and evolvability. Also, increasing the budget will require practitioners to consider some tradeoffs: how much would it cost to pay off the debt? vs. how much does it cost to maintain the debt? According to Martini and Bosch [34], it could be more profitable to delay the refactoring, i.e. continue paying interest, but this will be a decision made by software teams.

After reviewing the list of TD payment practices (Table 2), some of the cited practices do not allow the elimination of TD items, such as *adoption of good practices* and *improve testing*. TD payment-related practices encompass practices associated with TD payment, prevention, and the creation of a favorable scenario for paying off debt items. This situation could be explained considering the context of the survey, where respondents could or could not, be part of the implementation of the payment practice. Therefore, they could go from a general (and maybe abstract) idea of how it was paid off, to a full understanding of the payment activity carried out.

5.1.1. RQ1.1: Is it possible to find similarities of the TD payment practices according to the TD type associated with them?

A comparison of the TD payment practices used in cases of code debt, test debt, and design debt was performed and presented in Fig. 2. Software architects, in the context of our study, have in mind more

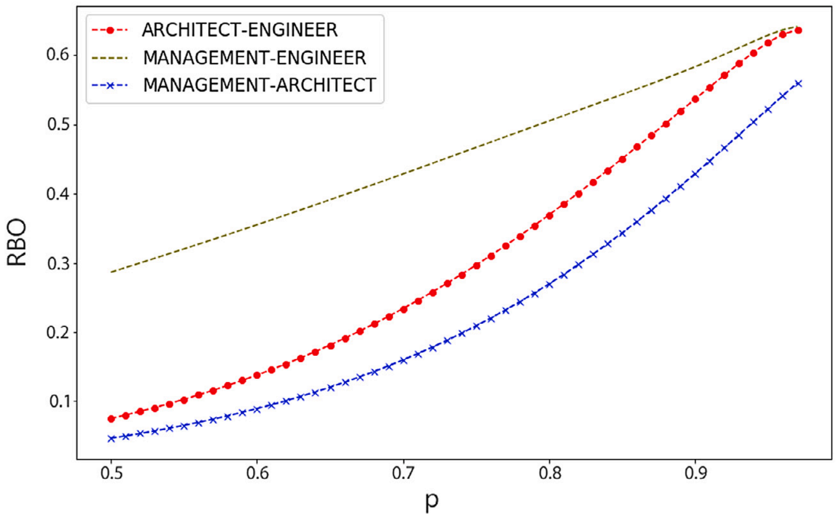


Fig. 5. RBO of TD preventive practices per software development role.

cases of code debt, than cases of design debt or test debt. This could be an indicator of what software architects consider as the root of TD injection in software projects.

This comparison showed that the practices used in the cases of code debt and design debt were the most similar. This could be considered evident given the close relationship between code and design during software development. The pair test debt-code debt shows an increasing similarity as many practices were compared. Unfortunately, there were only five cases of test debt and therefore, no more items could be included in the comparison. Code and test are related and it could be expected some increasing similarity between practices used to pay off TD in these two types of debts.

Also, based on Table 3, it is possible to note that only three practices are shared among the three TD types: *refactoring*, *improve design*, and *improve testing*, which are also in the top 5 TD payment practices (Table 2). So, there is a common agreement in what the main practices related to TD payment for all three TD types are.

5.1.2. RQ1.2: Is it possible to establish an association between main causes leading to TD occurrence and main practices related to TD payment?

Fig. 3 gave us the opportunity to understand what practices are used according to what caused the TD injection. For example, the cause *producing more without quality* directly affects the decisions that ended in the code. If the code works, it goes to production. Then, it is expected to have *refactoring* and *improve design* as the main practices used to pay off TD. This same happened for the cause *inappropriate planning*. Bad planning leads to deadlines, and deadlines lead to producing code without the expected quality. So, from this point of view, *producing more without quality* could be a consequence or effect of *inappropriate planning*.

Another cause, *inadequate choice of technology/tool/platform*, has *refactoring* and *code reviewing* as main payment practices. This could be more related to architectural decisions. If the team chooses some technology and then they decide that it was not the best choice, then the best choice could be a change in the platform or tool. However, this could be expensive in some cases, and therefore, small improvements in the code should be the only possible actions to take.

Refactoring is also the main practice to pay off the TD injected by *team overload* of work. It seems there is not much of a relationship between this cause and this payment practice. However, if the practitioner has a lot of work to do, they will try to work as fast as possible (maybe injecting all kind of debt). As a consequence, there will be a code without the expected quality.

Further research should be undertaken to investigate the types of refactoring and the process of carrying it out. Also, it is mandatory to go further in measuring the true amount of TD types in software projects, and also, which quality attributes are most likely to inject TD. This information could not be deduced from the answers in the survey.

5.2. RQ2: From a software architect's point of view, what practices have been performed to prevent TD occurrence?

TD preventive practices allow software practitioners to curb the presence of TD in their software systems. It is important to mention that Q28 asks participants to report the practices performed to prevent TD. From this question it is possible to establish that practices reported were success at some level because they were used to prevent TD injection. It is not clear, however, if each practice worked by itself or it was the joint effort of the reported practices.

From the point of view of software architects, the most cited practice is *well-defined architecture/design*, followed by *well-defined scope/requirements*, and *code evaluation/standardization*. Architectural decisions are the main source of TD, and therefore, a good architecture design would imply a reduction of TD in the system. However, this would only reduce deliberately injected TD. Inadvertently injected TD is inevitable. As a result, the TD trickles down in the development process reaching the code. It is also relevant to understand what makes architecture good.

The architect, from his perspective, could be sure that the architecture is correct. If detailed evaluations of the architecture could be carried out in all cases, then architects would be aware of existing TD items and deliberate trade-offs made during design sessions. Also, TD injected as a consequence of architectural decisions can also be introduced without anyone's fault. Sometimes, decisions made by architects are made in the context of a specific time, and it may have been the best decision at that point. Later, as new requirements and the environment surrounding the system changes, the decision made in the past now becomes a problem. Unfortunately, this is an open issue. In the practice, it is not pragmatic to constantly revisit past decisions when designing new software. If software architects were to do this, then the "revisits" actions would become a burden in it of itself.

Having a well-defined architecture looks more like a goal to reduce the presence of TD, than a practice to prevent TD occurrence. A well-defined architecture would depend on having enough understanding of the requirements and the business. This is the main input with which the architect must work. However, this is not feasible all the time. Software architects make decisions under conditions of time pressure, high stakes, uncertainty, and with too little information [35]. Software architects must work under these conditions, and iterate as they go along. Despite this, we acknowledged that the second most cited practice (*well-defined scope/requirements*) could be considered a precursor of good architecture and also, a real practice to prevent TD occurrence.

The first two TD preventive practices are strongly related to the software architecture stage. The third practice, (*code evaluation/standardization*), is related to the development stage. It is important to note that software architects acknowledge the importance of having clearly defined the software requirements in order to have a well-defined architecture. Then, as the code begins to be developed, the changes to take care of, in the code base to avoid TD injection, are apparent. Software architects are aware that TD injection can be prevented by focusing on the architecture and then on the source code.

This list of preventive practices is focused on technical practices. The only practice out of this trend is *better project management*. This is unexpected considering that 4 out of 9 causes of TD injection (Section 4.4) have a management inkling. Software architects acknowledge that the majority of TD items are actually related to technical aspects of software development, no matter external aspects such as management issues.

5.2.1. RQ2.1: Does the software architect's level of expertise influence the preventive practices performed by them?

In Section 5.2, it was stated that *well-defined scope/requirements* is a practice for TD prevention and cited 2nd in the list of the most cited TD preventive practices. This practice was recognized as the most important one (most citations) for the expert group (8 citations). This practice was also cited by proficient (2 citations), competent (1 citation), and beginner group (1 citation).

Well-defined architecture/design was acknowledged by all groups (expert: 3 citations). This is expected considering the role of respondents. All groups understand that good architecture could prevent the presence of TD. However, only experts were clear about the importance of requirements. *Code evaluation/standardization* was mainly cited by proficient (7 citations) and beginner (2 citations) groups. It was also cited by expert (1 citation) and competent (2 citations) groups. This result indicates that source code is still a concern for software architects as a means of preventing TD injection.

TD awareness/management was mainly and only cited by expert and proficient groups. This result marks a separation line between groups with a high level of experience and groups with a low level of experience. Lastly, this result brings an interesting opportunity to go further to understand how being aware of the TD concept could be a TD preventive practice. We believe that tracking TD items as part of the daily activities of a software team can make the team more proactive in terms of avoiding the occurrence of new debt items.

5.2.2. RQ2.2: Are preventive practices performed by software architects different from the ones performed by other software development roles?

At first sight (Table 6) it is possible to think that engineer and management roles have performed similar TD preventive practices. These two roles shared 4 out of their first 5 most cited practices. This is truly unexpected considering the differences between these two roles. This could be related to the general idea about TD and source code. But the shared practices are not only about technical concerns, they also cited preventive practices about management concerns and soft skills: *better project management* and *good communication on team*. So, at least, both roles understand the importance of having good management of the software projects and to have good communication channels. This latter could be done through the best meeting spaces and a good working environment.

Also, it was possible to note that none of the roles shared a most cited practice. Software architects cited *well-defined architecture/design*, engineer roles cited *adoption of good practices* and management roles cited *better project management*. All three roles share this list of practices, but not in the same order. Even more, none of the practices share the same position in two or more roles, except by *adoption of good practices*, which share the 5th position in software architects and management roles. This means that every role has its worries and focuses on different strategies to reduce TD presence in the software.

5.3. Implications to researchers and practitioners

Software practitioners can benefit from the results of this study by using the list of the most cited TD causes and practices related to TD payment used in industry to support initial efforts to understand their debt and to pay it off from their software projects. However, this list of payment practices by itself is not quite enough. It becomes necessary to analyze the differences among these practices in order to understand the nature of the required changes (improvements) and the resources needed, such as the frequency of the payment practice, the cost related to the debt, among others. In the end, the success of the implementation of any practice will depend on the software team.

Also, in response to RQ2, software practitioners could review the list of TD preventive practices as a guide to include TD prevention into their software development process, according to their level of experience. Or they can go deeper on practices provided by experts and find a way to include them in the development process. A lot of work could be required to have a full implementation of these practices, however, even a small change can have a impact on the development process.

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

6. Related work

There are studies related to payment practices described by software practitioners. In [1], Yli-Huumo et al. conducted an exploratory case study method to collect and analyze empirical data by performing semi-structured interviews to 25 software practitioners (11 software architects) from eight (8) software development teams in one large software company. Related to TD prevention, the vast majority of development teams used coding standards to prevent TD, along with code reviews, and the definition of done. These results were also found in [36] where software teams used coding standards/guides to prevent and reduce TD.

Several other studies focused on the software practitioners' involvement with TD through empirical methods such as interviews and questionnaires. In [7], Ernst et al. focused on the relationship of software architecture and TD, and the use of tools for TD management. In [8], Rios et al. focused on understanding causes and conditions to prevent TD injection. They found that preventive strategies depend on causes of TD. In [37], Codabux et al. focused on TD definition, characterization, consequences, benefits, and how it is communicated. They found that allocating developer time to address TD is the most common practice to manage the debt. In [38], Codabux and Williams reported automating manual tests and fixing defects in their own code as practices to reduce TD presence.

In [11], Rios et al. presented a tertiary study to investigate the current state of research on TD. They analyzed 13 secondary studies and reported a TD management landscape, including activities, strategies, and tools. In this landscape, they reported four TD management macro activities: prevention, identification, monitoring and payment. They found that no strategies and/or tools have been identified to support TD prevention activity. Finally, in [10], Li et al. performed a systematic mapping study to obtain a comprehensive understanding on the TD concept and an overview on the current state of research on TDM. In this study, they identified nine TD management activities.

For TD prevention, they established four approaches: development process improvement, architecture decision making support, lifecycle cost planning, and human factors analysis.

In the context of the InsighTD project, a lot of research is already done leveraging the project data. This work is part of InsighTD, but, unlike its previous works, we aim to study the practices related to TD payment and actions to prevent TD injection, from the software architect's point of view. Fig. 6 presents the timeline of studies done using the project data. As shown, there has been a lot of studies with different focuses or populations:

- TD concept. This subject is covered in questions Q9 to Q11 by asking about the familiarity with the TD concept and comparing his/her understanding of TD with the concept presented in McConell [29]. Studies focusing on this subject are: P1 [13], P4 [17], P8 [5] and P11 [39].
- TD causes and/or effects. These subjects are covered in questions Q16 to Q21 by asking about causes and effects related to the case presented in Q13. Studies focused on these subjects are: P1 [13], P2 [16], P3 [20], P4 [17], P7 [18], P8 [5] and P10 [40]
- TD prevention. This subject is covered in questions Q22 and Q28. Question 22 asks about preventive practices to be used in the example case presented in Q13. Question 28 ask about general practices performed by respondents to prevent TD injection. Currently, only one study has focused on this subject: P6 [9].
- TD Repayment. This subject is covered in questions Q26 and Q27 by asking participants about practices used in the case presented in Q13 to pay off the debt in their systems. Studies focusing on this subject are: P5 [30] and P9 [12].
- TD monitoring. This subject is covered in questions Q24 and Q25. Currently, no studies (within InsighTD context) have covered this subject.

There are several of these studies with which our work shares some similarities. Freire et al. [9] investigated, from the point of view of software practitioners, the preventive actions that can be used to curb the occurrence of TD and the impediments that hamper the use of those actions. This study used data from Brazil and the United States. They presented results about the team capacity to prevent the occurrence of TD, and the top 10 most cited preventive actions. This study includes the preventive practices that could be used to prevent the TD described in the example case (Question 13) of InsighTD survey. Results include preventive practices from question Q23 and question Q28. Our study includes only answers to Q28. This way, our study is not focused on preventive practices proposed for a specific case, but in the general preventive practices performed by software architects. This leads to differences in the practices employed, for example, in [9], the most cited practices used are: following project planning, adoption of good practices, well-defined requirements. In our study, the most cited practices are: well-defined architecture/design, well-defined scope/requirements and Code evaluation/standardization. These differences are related not only to the exclusion of Q23 but also to the population used. More recently, Pérez et al. [42] focused on TD causes as described by software architects from Colombia. Authors included a list of the top 7 most cited causes of TD injection, and an RBO analysis of the comparison of the causes cited by software architects, developers, and management.

Thus, although significant analysis has already been conducted, much still remains to be studied. In particular, a noticeably absent and important perspective is the one on TD payment and preventive practices from the architect's point of view. Decisions that affect software artifacts and TD are typically made earlier in the lifecycle, and software architects play a critical role in this space. Thus, the major difference between these studies and our study is that our study focused on industry practitioners (software architects). Besides, this work includes results from four replications of InsighTD (72 responses), which give us a broader spectrum of analysis. Our study focuses on TD payment practices, their similarities according to their corresponding TD type (RQ1.1) and their possible relationship among the causes leading to TD occurrence (RQ1.2). Lastly, our study includes an analysis of the preventive actions according to the experience of software architects (RQ2.1), and its comparison with engineer roles and management roles (RQ2.2).

7. Threats to validity

There are threats to validity in this study that we attempt to mitigate and remove when possible. The main threats regarding this study are [43]: external validity, internal validity, construct validity, and reliability.

External validity. Respondents are well distributed based on their experience, software project participation, company size, and country. Thus, by achieving a diversity of participants who answered the survey we look to reduced this threat. These results cannot be generalized, considering the small number of software architects. Also, the anonymous nature of the survey could reduce the possibility of characterizing 72 different TD cases because one or more respondent could describe the same TD case example (Q13). However, it can be argued that a more narrow definition of external validity (ecological validity), the extent to which these findings approximate similar situations in other real world environments, is likely to hold. We expect that software architects outside our study are very likely to encounter similar situations. Hawthorne effects, which implies behavior changes in subjects of an experimental study because they are being studied, were mitigated by both the online and anonymous nature of the survey. Finally, another threat to this study is related to the level of architect's experience. There were not a way to get to know if the level of experience selected by the architects are actually the real one. However, considering that no pressure was done by the survey nor prizes given for their answers, we could expect truly honest answers.

Internal validity. One limitation of this study was *maturation*. It implies that the participants can react differently if the survey is too long, for example, if respondents were tired or bored during the experiment [43]. This threat is also related to *history*, where results could be affected by answer the survey on the first day after a holiday or on a normal day. These two threats were not possible to handle due to the online nature of the survey. Respondents could answer the survey having a good mood or maybe after some boring meeting. Another limitation was *mortality*, and it implies to get to know if the number of persons who drop out from the experiment are representative of the total sample. This threat was not raised considering that all participants answered the whole questionnaire. Related to *selection* bias, the LinkedIn platform was used to contact to participants from a variety of roles and cities within a country. First, an invitation to connect was sent, and after their acceptance, a second message with the invitation to participate in the survey was sent. Also, related to the software architect role used for this study, respondents were asked to select their project role or to enter a not listed project role through question Q6. This way we were able to select only those who identified themselves as software architects. Finally, 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.

Construct validity. To prevent *hypothesis guessing* and *evaluation apprehension* [43], the goal of the study was explained in the invitation to the survey. Also, we requested respondents to answer the survey by relying on their background. We provided the definition of TD according to McConnell [29] to make sure the respondents were considering the same interpretation of TD. However, a definition of the main practices of TD payment was missing as a part of the survey. Respondents may have mistaken the differences between refactoring and rewriting, and differences among types of refactoring. Not all refactoring is at the code level, as stated in the literature [10,33].

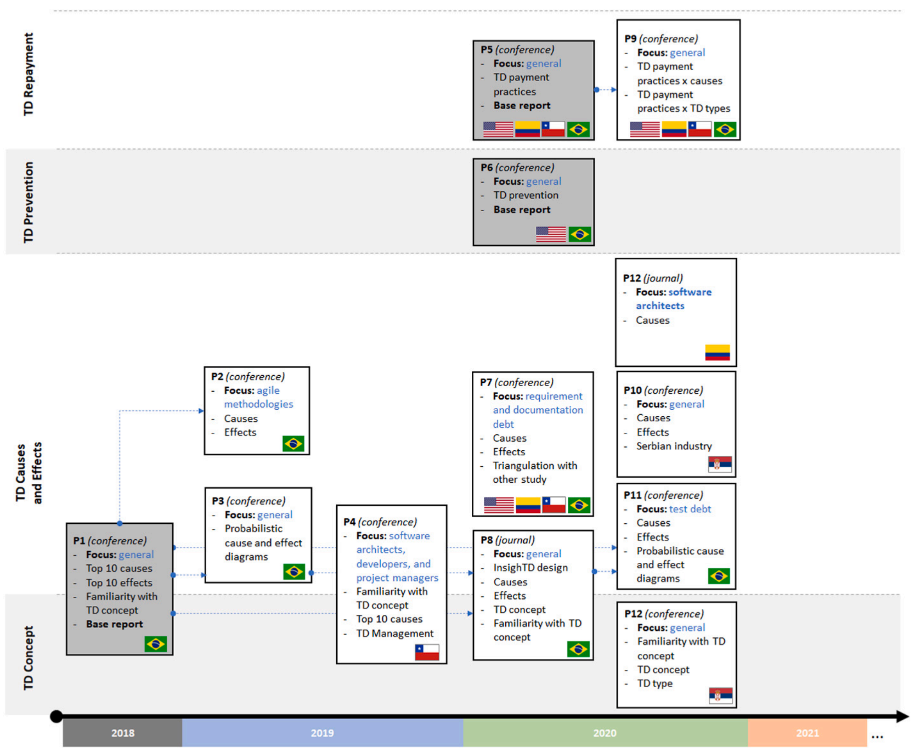

TD Repayment
TD Prevention
TD Causes and Effects
TD Concept
P1 (conference) - Focus: general - Top 10 causes - Top 10 effects - Familiarity with TD concept - Base report
P2 (conference) - Focus: agile methodologies - Causes - Effects
P3 (conference) - Focus: general - Probabilistic cause and effect diagrams
P4 (conference) - Focus: software architects, developers, and project managers - Familiarity with TD concept - Top 10 causes - TD Management
P5 (conference) - Focus: general - TD payment practices - Base report
P6 (conference) - Focus: general - TD prevention - Base report
P7 (conference) - Focus: requirement and documentation debt - Causes - Effects - Triangulation with other study
P8 (journal) - Focus: general - InsighTD design - Causes - Effects - TD concept - Familiarity with TD concept
P9 (conference) - Focus: general - TD payment practices x causes - TD payment practices x TD types
P10 (conference) - Focus: general - Causes - Effects - Serbian industry
P11 (conference) - Focus: test debt - Causes - Effects - Probabilistic cause and effect diagrams
P12 (journal) - Focus: software architects - Causes
P12 (conference) - Focus: general - Familiarity with TD concept - TD concept - TD type
2018
2019
2020
2021
...

Fig. 6. The InsighTD publication map [41].

Reliability. To mitigate this threat and to avoid potential coding process dependencies on the researcher's subjective criteria, the coding activity was performed separately and independently by two researchers, and then, discussed until an agreement was reached. A referee was required in some cases to help resolve disagreements in codes identified. Another threat related to reliability is *random irrelevancies in experimental setting*, where elements outside the experimental setting may disturb the results, such as noise outside the room or a sudden interrupt in the experiment. This threat was not possible to handle due the online and self-paced nature of the survey. Respondents could answer the survey in their homes or in their offices, having all kinds of working or personal distractions.

8. Conclusions

This study focused on understanding how TD is perceive form the point of view of a software architect. This perception was studied by analyzing what TD payment practices were used by software teams as reported by the architects, and the TD preventive practices performed by them to avoid or reduce debt presence in software projects.

The contributions of this work are two-fold. First, an analysis of the most used TD payment-related practices is presented. Based on the 16 TD payment-related practices, we found that *refactoring* and *improve design* were the most cited TD payment practices used by software teams, as reported by software architects. This was relevant considering that both practices are intertwined.

As part of this first contribution, a comparison of the similarity of the payment practices according to the three major TD types cases (code debt, test debt, design debt) was presented. We found that software architects, in the context of our study, have in mind more cases of code debt than cases of design debt or test debt. This comparison showed that the practices used in the cases of code debt and design debt were the most similar. A last part of this first contribution was the building of a heatmap of the most cited TD payment-related practices against the most cited TD causes reported by software architects. From this, we identified that more causes cited by architects have technical inkling, and that *refactoring* was used in 5 out of 9 causes leading to TD occurrence: *non-adoption of good practices*, *inappropriate planning*, *lack of qualified professionals*, *change of requirements*, and *team overload*. We also found that causes related to TD injection were mostly related to the lack of good practices and unskilled developers.

In the second contribution of this study, an analysis of the most frequently performed TD preventive practices was presented. We found that *well-defined architecture/design* and *well defined scope/requirements* were the most cited preventive practices. Software architects were aware of the relevance of their practice within the software development process. We reviewed these practices according to the level of experience of the architects. We found that expert and proficient groups have their own most cited practice: *well defined scope/requirements* and *well-defined architecture/design*. Expert software architects understand that a well-defined architecture as a goal will require a well-defined scope and requirements. Finally, when comparing preventive practices among the three major roles derived from the survey (software architects, engineer roles, and management roles), we found that none of the roles shared the most cited practice, meaning that each role had its worries and focus on different strategies to reduce TD's presence in the software.

TD payment and TD prevention are important because they are related to each other. Debt prevention can be better and cheaper for the development team than incurring debt and paying it off later. On the other side, incurring TD items is quite expected and even necessary during software development projects, leading to the necessity of seeking the balance between preventing their occurrence and paying them off. Further research should be undertaken to go deeper in understanding this relationship. In the context of the InsighTD project, we intend to go further into this topic by the means of interviews and focus groups, and also, by including more data from other replications of InsighTD. Finally, additional work is necessary to understand how the findings of this work correlate with success factors.

CRediT authorship contribution statement

Boris Pérez: Conceptualization, Formal analysis, Investigation, Writing – original draft. **Camilo Castellanos:** Formal analysis. **Darío Correal:** Writing – review & editing. **Nicolli Rios:** Methodology, Resources. **Sávio Freire:** Methodology, Resources. **Rodrigo Spínola:** Methodology, Resources, Writing – review & editing. **Carolyn Seaman:** Methodology, Resources. **Clemente Izurieta:** Conceptualization, Resources, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] J. Yli-Huumo, A. Maglyas, K. Smolander, How do software development teams manage technical debt? – an empirical study, J. Syst. Softw. 120 (2016) 195–218, http://dx.doi.org/10.1016/j.jss.2016.05.018, URL http://www.sciencedirect.com/science/article/pii/S016412121630053X.

[2] L. Leite, C. Rocha, F. Kon, D. Milojicic, P. Meirelles, A survey of devops concepts and challenges, ACM Comput. Surv. 52 (6) (2019) http://dx.doi.org/10.1145/3359981.

[3] P. Kruchten, R.L. Nord, I. Ozkaya, Technical debt: From metaphor to theory and practice, Ieee Softw. 29 (6) (2012) 18–21.

[4] C. Seaman, Y. Guo, Measuring and monitoring technical debt, in: Advances in Computers, Vol. 82, Elsevier, 2011, pp. 25–46.

[5] N. Rios, R.O. Spínola, M. Mendonça, C. Seaman, The practitioners' point of view on the concept of technical debt and its causes and consequences: a design for a global family of industrial surveys and its first results from brazil, Empir. Softw. Eng. (2020) 1–72.

[6] A. Martini, J. Bosch, M. Chaudron, Architecture technical debt: Understanding causes and a qualitative model, in: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, 2014, pp. 85–92, http://dx.doi.org/10.1109/SEAA.2014.65.

[7] N.A. Ernst, S. Bellomo, I. Ozkaya, R.L. Nord, I. Gorton, Measure it? manage it? ignore it? software practitioners and technical debt, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 50–60.

[8] N. Rios, R. Oliveira Spinola, M.G. de Mendonça Neto, C. Seaman, A study of factors that lead development teams to incur technical debt in software projects, in: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2018, pp. 429–436.

[9] S. Freire, M. Mendonça, D. Falessi, C. Seaman, C. Izurieta, R.O. Spínola, Actions and impediments for technical debt prevention: Results from a global family of industrial surveys, in: The Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing, ACM, 2020, http://dx.doi.org/10.1145/3341105.3373912.

[10] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, J. Syst. Softw. 101 (2015) 193–220.

[11] N. Rios, M.G. de Mendonça Neto, R.O. Spínola, A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners, Inf. Softw. Technol. 102 (2018) 117–145, http://dx.doi.org/10.1016/j.infsof.2018.05.010.

[12] B. Pérez, C. Castellanos, D. Correal, N. Rios, S. Freire, R. Spínola, C. Seaman, What are the practices used by software practitioners on technical debt payment? Results from an international family of surveys, in: Proceedings of the IEEE/ACM International Conference on Technical Debt (TechDebt), 2020.

[13] N. Rios, R.O. Spínola, M. Mendonça, C. Seaman, 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, 2018, p. 39.

[14] J. Yli-Huumo, A. Maglyas, K. Smolander, 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, 2014, pp. 93–107.

[15] N.S. Alves, T.S. Mendes, M.G. de Mendonça, R.O. Spínola, F. Shull, C. Seaman, Identification and management of technical debt: A systematic mapping study, Inf. Softw. Technol. 70 (2016) 100–121, http://dx.doi.org/10.1016/j.infsof.2015.10.008, URL http://www.sciencedirect.com/science/article/pii/S0950584915001743.

[16] N. Rios, M.G. Mendonça, C. Seaman, R.O. Spinola, Causes and effects of the presence of technical debt in agile software projects, 2019.

[17] B. Pérez, J.P. Brito, H. Astudillo, D. Correal, N. Rios, R.O. Spínola, M. Mendonça, C. Seaman, 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, 2019.

[18] N. Rios, L. Mendes, C. Cerdeiral, A.P.F. Magalhães, B. Perez, D. Correal, H. Astudillo, C. Seaman, C. Izurieta, G. Santos, R. Oliveira Spínola, Hearing the voice of software practitioners on causes, effects, and practices to deal with documentation debt, in: N. Madhavji, L. Pasquale, A. Ferrari, S. Gnesi (Eds.), Requirements Engineering: Foundation for Software Quality, Springer International Publishing, Cham, 2020, pp. 55–70.

[19] A. Pacheco, G. Marín-Raventós, G. López, Technical debt in costa rica: An insightd survey replication, in: International Conference on Product-Focused Software Process Improvement, Springer, 2019, pp. 236–243.

[20] N. Rios, R.O. Spínola, M.G. de Mendonça Neto, C. Seaman, 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, 2019, pp. 3–12.

[21] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing technical debt in software engineering (dagstuhl seminar 16162), in: Dagstuhl Reports, Vol. 4, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[22] S. McConnell, Code Complete, Pearson Education, 2004.

[23] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, N. Zazworka, Managing technical debt in software-reliant systems, in: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, in: FoSER '10, ACM, New York, NY, USA, 2010, pp. 47–52, http://dx.doi.org/10.1145/1882362.1882373, URL http://doi.acm.org/10.1145/1882362.1882373.

[24] A. Martini, T. Besker, J. Bosch, The introduction of technical debt tracking in large companies, in: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), 2016, pp. 161–168, http://dx.doi.org/10.1109/APSEC.2016.032.

[25] T. Besker, A. Martini, J. Bosch, Managing architectural technical debt: A unified model and systematic literature review, J. Syst. Softw. 135 (2018) 1–16.

[26] V.L. Johannes Holvitie, S. Hyrynsalmi, Technical debt and the effect of agile software development practices on it - an industry practitioner survey, in: 2014 Sixth International Workshop on Managing Technical Debt, 0000.

[27] F. Team, 3 types of survey research, when to use them, and how they can benefit your organization!, 2014, URL http://fluidsurveys.com/university/3-types-survey-research-use-can-benefit-organization/.

[28] B.A. Kitchenham, S.L. Pfleeger, Principles of survey research part 2: Designing a survey, SIGSOFT Softw. Eng. Notes 27 (1) (2002) 18–20, http://dx.doi.org/10.1145/566493.566495, URL http://doi.acm.org/10.1145/566493.566495.

[29] S. McConnell, Technical debt, 2007, URL http://www.construx.com/10x_Software_Development/Technical_Debt/.

[30] S. Freire, N. Rios, B. Gutierrez, D. Torres, M. Mendonça, C. Izurieta, C. Seaman, R.O. Spínola, Surveying software practitioners on technical debt payment practices and reasons for not paying off debt items, in: Proceedings of the Evaluation and Assessment in Software Engineering, in: EASE '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 210–219, http://dx.doi.org/10.1145/3383219.3383241.

[31] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, F. Shull, Organizing the technical debt landscape, in: 2012 Third International Workshop on Managing Technical Debt (MTD), 2012, pp. 23–26, http://dx.doi.org/10.1109/MTD.2012.6225995.

[32] W. Webber, A. Moffat, J. Zobel, A similarity measure for indefinite rankings, ACM Trans. Inf. Syst. 28 (4) (2010) 20:1–20:38.

[33] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 2018.

[34] A. Martini, J. Bosch, M. Chaudron, Investigating architectural technical debt accumulation and refactoring over time, Inf. Softw. Technol. 67 (C) (2015) 237–253, http://dx.doi.org/10.1016/j.infsof.2015.07.005.

[35] K. Power, R. Wirfs-Brock, An exploratory study of naturalistic decision making in complex software architecture environments, in: T. Bures, L. Duchien, P. Inverardi (Eds.), Software Architecture, Springer International Publishing, Cham, 2019, pp. 55–70.

[36] J. Yli-Huumo, A. Maglyas, K. Smolander, The sources and approaches to management of technical debt: A case study of two product lines in a middle-size finnish software company, in: A. Jedlitschka, P. Kuvaja, M. Kuhrmann, T. Männistö, J. Münch, M. Raatikainen (Eds.), Product-Focused Software Process Improvement, Springer International Publishing, Cham, 2014, pp. 93–107.

[37] Z. Codabux, B.J. Williams, G.L. Bradshaw, M. Cantor, An empirical assessment of technical debt practices in industry, J. Softw.: Evol. Process. 29 (10) (2017) e1894, http://dx.doi.org/10.1002/smr.1894, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1894 URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1894, e1894 JSME-16-0113.R2.

[38] Z. Codabux, B. Williams, Managing technical debt: An industrial case study, in: 2013 4th International Workshop on Managing Technical Debt (MTD), 2013, pp. 8–15.

[39] L. Souza, S. Freire, V. Rocha, N. Rios, R.O. Spínola, M. Mendonça, Using surveys to build-up empirical evidence on test-related technical debt, in: Proceedings of the 34th Brazilian Symposium on Software Engineering, in: SBES '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 750–759, http://dx.doi.org/10.1145/3422392.3422430.

[40] R. Ramac, V. Mandic, N. Taušan, N. Rios, M. Mendonça, C. Seaman, R.O. Spínola, Common causes and effects of technical debt in serbian it: Insightd survey replication, in: The Proceedings of the Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA), ACM, 2020, http://dx.doi.org/10.1109/SEAA51224.2020.00065.

[41] InsighTD, Publications, 2020, URL http://www.td-survey.com/publication-map/.

[42] B. Pérez, D. Correal, F.H. Vera-Rivera, How do software architects perceive technical debt in colombian industry? an analysis of technical debt causes, J. Phys. Conf. Ser. 1513 (2020) 012003, http://dx.doi.org/10.1088/1742-6596/1513/1/012003.

[43] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, 2012.