Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt

Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton*
Carnegie Mellon University Software Engineering Institute
Pittsburgh, PA, USA
{nernst,sbellomo,ozkaya,rn,igorton}@sei.cmu.edu

ABSTRACT

The technical debt metaphor is widely used to encapsulate numerous software quality problems. The metaphor is attractive to practitioners as it communicates to both technical and nontechnical audiences that if quality problems are not addressed, things may get worse. However, it is unclear whether there are practices that move this metaphor beyond a mere communication mechanism. Existing studies of technical debt have largely focused on code metrics and small surveys of developers. In this paper, we report on our survey of 1,831 participants, primarily software engineers and architects working in long-lived, software-intensive projects from three large organizations, and follow-up interviews of seven software engineers. We analyzed our data using both nonparametric statistics and qualitative text analysis. We found that architectural decisions are the most important source of technical debt. Furthermore, while respondents believe the metaphor is itself important for communication, existing tools are not currently helpful in managing the details. We use our results to motivate a technical debt timeline to focus management and tooling approaches.

Categories and Subject Descriptors

K.6.3 [**Software Management**]: Management–software development, software selection

Keywords

Technical debt, architecture, survey

1. INTRODUCTION

The technical debt metaphor concisely describes a universal problem that software engineers face when developing software: how to balance near-term value with long-term quality. The appeal is that once technical debt is made visible, engineers (and eventually, their managers) can begin to understand in what ways debt can be harmful or beneficial to a project. Debt accumulates and causes ongoing costs ("interest") to system quality in maintenance and evolution. Debt can be taken on deliberately, then monitored and managed ("principal repaid"), to achieve business value.

The usefulness of this concept prompted the software engineering research community, software consultants, and tool vendors alike to pay more attention to understanding what constitutes technical debt and how to measure, manage, and communicate technical debt. Recent systematic literature reviews [19, 33] report that

- using code quality analysis techniques to understand technical debt has been the dominant focus in research and by tool vendors.
- beyond code quality, other work explores the suitability of the metaphor in other phases of the software life cycle: for example, "requirements debt," "testing debt," "code debt," and "design debt."

Practitioners currently use the term *technical debt* to mean, broadly, a "shortcut for expediency" [23] and, more specifically, bad code or inadequate refactoring [15]. Shull et al. [29], in a review paper on research in technical debt, highlight that technical debt is a multi-faceted problem. Addressing it effectively in practice requires research in software evolution, risk management, qualitative assessment of context, software metrics, program analysis, and software quality. Applying technical debt research starts with identifying a project's sources of "pain." In order to give guidance to a specific project, research in this area must be grounded in the project's context.

This combination of diverse definitions of technical debt in research, alongside the need to ground research on technical debt in practice, raised three research questions:

1. To what extent do practitioners have a commonly shared definition of technical debt?
2. How much of technical debt is architectural in nature?
3. What management practices and tools are used in industry to deal with debt?

To investigate these questions, we conducted a two-part study. First, we administered a survey of software professionals in three large organizations, with 1,831 responses; second, we held semistructured, follow-up interviews with seven respondents, all professional software engineers, to further investigate the emerging themes.

*Now at College of Computer and Information Science, Northeastern University, i.gorton@neu.edu

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786848

Our study results show that the concept of technical debt has a broad shared understanding among software practitioners and managers. Our main finding is that architectural choices, often made very early in the software life cycle, are a key source of technical debt. In the perception of developers and architects, management remains unaware of the importance of technical debt. The lack of tool support for accurately managing and tracking architectural sources of debt is a key issue and remains a gap in practice. This stands in contrast to the existing heavy focus on code-based analysis of technical debt. Building on these observations, we introduce a timeline of technical debt to better distinguish between the time when debt is incurred and the ongoing problems it causes to help clarify issues and focus future research.

2. RELATED WORK

Defining, analyzing, visualizing, and managing technical debt have received increasing research attention. Here we present relevant work that maps to our research questions and findings.

Defining technical debt. The original definition of technical debt is from Ward Cunningham [4]. Since then, numerous people have proposed definitions, including McConnell [23], Kruchten [16], and Seaman [29], among others (e.g., Fowler[8] or Kniberg [15]). In order to understand implications of technical debt, there have been attempts to create categories of debt or relate it to different development life cycles [13, 1]. Small-scale interview-based studies on understanding how developers talk about technical debt also exist [21, 31]. Our study is the first study that looks at the perceptions of software professionals with a broad empirical basis.

Correlating code violations with technical debt measures. A number of studies have examined the relationship, if any, between software metrics and technical debt. These studies applied existing code smells, coupling and cohesion, and dependency analysis to identify areas of technical debt [7, 11, 22]. There is also work investigating what adequate tool support means when focusing on code analysis and technical debt [6]. Our study highlights that the use of code violation tools and techniques is perceived to be of minimal value in understanding technical debt.

Architecture issues as technical debt. Understanding how to manage architectural concerns to avoid technical debt accumulation is an important research topic, as reflected by systematic literature reviews. This body of work includes efforts to come up with architectural measures [25], to map architectural dependencies [24] or pattern drift to decay [10], and to provide an uncertainty-based evolution of architectural refactoring priorities [20]. Our study is the first to empirically demonstrate that a focus on architectural issues has merit in practice as well.

Theoretical foundations of technical debt. Schmid explores technical debt as an aspect of software evolution [27], and Shull and others focus on empirical foundations [21]. Understanding the benefit, principle, and interest and mapping properly to software design decisions and evolution have been investigated in industrial case studies without generalizable results to date [3, 12, 30]. A theory of technical debt and its management practices is yet to emerge. Our study is different in that it investigates broad-based professional perceptions and contributes to further understanding of a theory of technical debt, with a timeline for mapping root causes and symptoms.

3. RESEARCH QUESTIONS

Our goal in this study was to understand how software engineers relate to technical debt and whether they use tools and techniques to manage it. We formulated the following research questions.

RQ1. Do professional software engineers have a shared definition of technical debt?

Research Question 1 investigates whether technical debt is a concept with shared meaning and whether there is consensus on what technical debt is at a fine-grained level (e.g., software life-cycle stage). This research topic was motivated by questions raised in the Managing Technical Debt workshop series [17] and a literature review from Li et al. [19].

RQ2. Are issues with architectural elements—such as module dependencies, external dependencies, external team dependencies, and architecture decisions—among the most significant sources of technical debt?

Research Question 2 investigates a mapping to a portion of the definition of technical debt by Steve McConnell: "A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time)" [23]. What do these short-term design approaches consist of?

RQ3. Are there practices and tools for managing technical debt?

Research Question 3 investigates whether software professionals are measuring technical debt, whether they think they should actively manage technical debt, and the role that tools may play.

4. STUDY DESIGN

Since we were interested in broad-based, industry practices, a targeted survey of practicing software professionals was appropriate. We paired that with follow-up interviews to investigate our research questions in more detail. We conducted the survey with software professionals—including developers, testers, architects, and managers—at two large multinational corporations and one government research lab (not ours). Based on the responses to the survey, and motivated by Research Question 2, we identified engineers who were concerned about bad architecture choices as technical debt for follow-up one-hour interviews.

Our study design is inspired by that described in Kim et al. on refactoring [14], Gousios et al. on pull-request integration [9], and Seaman [28] on survey and interview analysis.

4.1 Research Protocol

The research questions formed the basis for developing our survey and interview instruments and guided our analysis of the data.

If respondents indicated they were unfamiliar with the term, our survey instrument first gave McConnell's definition of technical debt ("a design or construction approach that is expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now, including increased cost over time" [23]). It then opened with multiple-choice ques-

Table 1: Collaborator Characteristics

Org.	Type	Description	Num. Surveys Sent
A	Multi-national for-profit	Fortune 500 organization with headquarters in the U.S., hardware and software organization	3,500+
B	Multi-national for-profit	Fortune 500 organization with headquarters in Europe, hardware and software organization	15,000+
C	Govt. R&D	U.S.-based research software development lab	200+

tions on demographics and system context. Next, we asked two open-ended (free-text) questions to hear in the respondents' own words their understanding of technical debt by asking them to provide a definition and an example. This was followed by closed questions based primarily on Likert-style rating of agreement or disagreement, with one ranking question. These closed questions were influenced by the definition and practices emerging from the reported results of the Managing Technical Debt research community (attended by academia and industry) [16]. We asked two additional open-ended questions about challenges managing technical debt and tools used.

We piloted our survey with a focus group consisting of other researchers and practicing architects to refine our questions. We mapped our questions in the survey to our research questions before analyzing the response data. Our survey and interview instruments are available online.[1] We cannot share actual responses due to confidentiality agreements with our collaborators.

To distribute the survey, we leveraged connections to industrial partners, and our partners were enthusiastic and helpful in obtaining broad corporate acceptance of the survey. We distributed the survey to the three different companies using their internal software developer channels, including special interest groups and custom mailing aliases. Table 1 lists their characteristics.

The survey was open at each organization for three weeks, with one initial announcement and one reminder midway. In two organizations, a drawing for an iPad was offered for those who participated in the survey.

After the survey, we conducted seven follow-up interviews with people who indicated that they were willing to help: four from Organization A, two from Organization B, and one from Organization C. All were senior developers or architects (a distinction without a difference for our subjects). Their systems ranged from 100,000 source lines of code (SLOC) to 1 million SLOC. We followed a predefined interview script (see the online appendix in Note 1) while allowing variation and path exploration as necessary. The interviews focused on the role of architectural decisions in technical debt.

4.2 Respondents

We sent the surveys to more than 18,000 potential targets, although it is hard to quantify how many were duplicates or inactive accounts. Across all three collaborators, 1,831 respondents began the survey, and 536 respondents answered all questions, an overall response rate of 29%, although response rate varied per question (which we report as we discuss each question). None of the questions were mandatory. Respondents had on average over 6 years experience, with 32% over 15 years; roles selected included developers (42%), system engineers (7%), QA/testers (7%), project leads/managers (32%), architects (7%), and other (6%).

There were 39 separate business units represented among the three companies, covering a broad set of domains, from scientific computing to command and control, business information systems, and embedded software. Most projects were web systems (24%) or embedded systems (31%). Projects generally consisted of 10 to 20 people, although 32% had fewer than 9 staff (including contractors and business staff). The systems were on average 3 to 5 years old, but a significant number (29%) were more than 10 years old. The systems were typically between 100,000 SLOC and 1 million SLOC in size. Most respondents used Scrum (33%) or incremental development methods (20%), but some used self-admitted waterfall methods (15%) and some had no methodology (17%!).

4.3 Analysis

For the closed-ended questions, we prefaced each question with a statement to "think of your current or most recent project" in order to ground the responses. We treat answers to Likert-style questions as ordinal (and not continuously distributed); therefore, comparing answer means is not appropriate.

We combined codes from three open-ended questions into a single set, since we found that our codes applied equally to all. These questions asked participants to (1) define technical debt ($n = 454$), (2) provide an example with cause and result ($n = 393$), and (3) list challenges to managing technical debt ($n = 304$). We applied manual coding on the three open-ended questions as follows: Initially, two authors individually coded a set of all answers for one question. This involved open coding as described in Strauss and Corbin [32], then axial coding to derive higher level categories. We chose a research focus—technical debt and architecture—but otherwise allowed the data to suggest relevant codes. For example, the definition "The added cost to properly implement system requirements after short term fixes are removed and proper design is installed" led to, among others, the *in vivo* code "added cost."

As we did this coding, the first two authors wrote theoretical memos about possible relationships among some of the emerging codes. We used those memos to sort our codes into higher level categories. We then presented a list of the 40 most commonly occurring codes to the remaining authors, along with a sample of 50 survey responses. All of the authors then coded the sample, after which we conducted card sorting to identify common themes and duplicates. Definitions of our codes can be found with our supplementary material (Note 1). Following the synchronization exercise, two of us coded each survey response with at least one and up to three of the finalized codes. We report on interview quotes with a letter and number corresponding to column 1 from Table 1. Other quotes are taken from open-ended responses.

[1] http://github.com/neilernst/td-survey

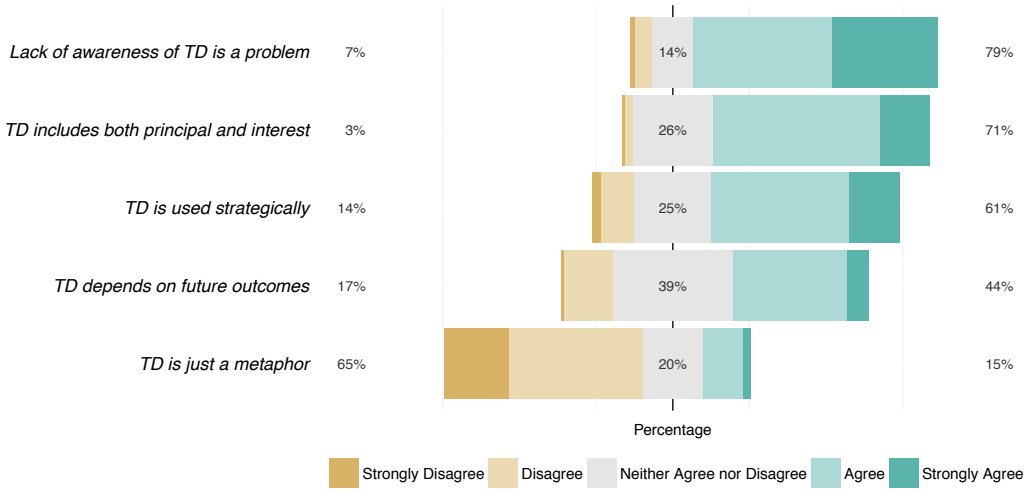

Figure 1: High-level definitions of technical debt. Percentages indicate SD + D, N, and A + SA, respectively.

5. DEFINING TECHNICAL DEBT

RQ1 asked whether software professionals have a shared definition of technical debt.

5.1 Technical Debt: A Useful Abstract Metaphor

Our data indicated that technical debt was a useful metaphor to communicate more abstract concepts, such as the need for investment.

Our respondents know what technical debt is, and they are struggling with it. Of 574 responses to the statement "Dealing with the consequences of technical debt has consumed a painful chunk of project resources," 75% agreed or strongly agreed, and only 10% disagreed or strongly disagreed (15% neutral). We found support on a number of questions for convergence on the concept of technical debt (see Figure 1): 79% agree or strongly agree that "lack of awareness is a problem" and 71% that "technical debt implies dealing with both principal and interest." This suggests that there is widespread agreement on high-level aspects of the technical debt metaphor, including some popular financial extensions of the metaphor.

Some of the most commonly occurring codes (*Awareness, Interest, Time Pressure*) on the open-ended questions (Figure 2) were similarly high-level concepts. For example, we coded as *Interest* the definition "extra effort in projects which is not required for purely technical reasons." We coded as *Time Pressure* the response "Business pressure to meet deadline. Less time allocation for design and development and quality testing." These were fairly common as examples of level of detail for an answer to the question "How would you define technical debt?" Another example was a challenge, coded as *Awareness*: "Getting programs to acknowledge they have it to begin with." These abstract concepts lack the detail for delineating the source of technical debt from the causes and consequences. Less common were answers pointing to the source such as "Code that has been incrementally developed over the years that is now so complicated" or "bugs and crash-downs."

Interview data also supported that technical debt is a useful metaphor for communicating with technical management. For example, "**(A2)** I think the vocabulary of technical debt is useful for getting the interests aligned."

The answers coded as *Awareness* seemed contradictory: how can technical debt be a useful metaphor, yet awareness still be a challenge? We think this is due to technical debt being at a nascent stage as a concept in software engineering. While early adopters (our respondents) have recognized its importance, a major perceived use is to begin conversations with those who are unaware of it, that is, "convincing product managers and stakeholders on the value proposition of managing the debt."

5.2 Moving Beyond the Metaphor

65% of respondents disagreed that technical debt was "only a metaphor," possibly because they recognize that a metaphor is not on its own sufficient to drive measurable outcomes. For that, one needs a reification of the metaphor: what specific elements of this project (architecture, code, defects, for example) need improvement? Is there one particular element that merits more attention?

We found that only architecture was commonly seen as a major source of technical debt, irrespective of context. Project context—such as framework choices, language, and life-cycle phase—clearly changes how technical debt is associated with each project instance. We can point to the broad number of business areas our survey covered (39), as well as the array of project ages and methodologies, as evidence of this. This diversity appears in the data as well. In some cases, respondents were concerned about hardware impacts on technical debt ("Changes in hardware area are made without understanding the impact on other areas (software) of the system."). Others were more concerned with documentation ("Comments in code have not been kept commensurate with the actions of the code whenever functionality has been changed."). In another, the database decisions long ago were paramount sources of technical debt ("Over 10 years ago there was a performance problem caused by the increasing size of the operational database.").

We asked respondents a number of questions to probe

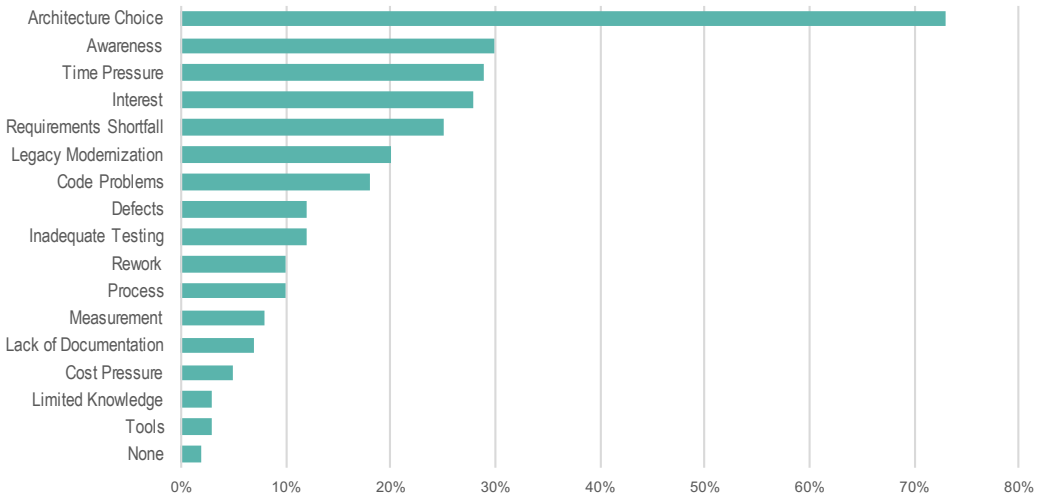

Figure 2: Coding frequency for open-ended questions.

whether there was a shared understanding of what constitutes sources of technical debt ($n = 570$). Figure 3 shows agreement on some items (e.g., architecture, that technical debt should not be treated in isolation from software development) and disagreement on others (e.g., defects and process). We interpret this to mean that for some project elements, there is confusion over what is and is not technical debt. For example, 45% of respondents agree or strongly agree that defects are technical debt, while 32% disagree or strongly disagree, showing little agreement. Figure 2—the open-coding results—showed a similar result: apart from architecture, which we cover in the following section, there is no obvious agreement on the source of technical debt. For instance, code occurred in only 18% of answers, and defects in only 11%. This is in contrast to the focus of current commercial tools and research, which predominantly deal with code quality, or industry thought leaders, such as Cunningham, who referred to "shipping first time code" [4], although ideally, code in an agile perspective should encapsulate architectural choices directly.

RQ1: Do professional software engineers have a shared definition of technical debt?
Finding 1: The technical debt metaphor is useful and commonly understood at an abstract level to convey urgency about accumulating software costs.
Finding 2: Apart from architecture, software professionals do not agree on which other project elements are sources of technical debt.

6. ARCHITECTURAL SOURCES OF DEBT

RQ2 asked whether or not architecture was perceived as a major source of technical debt. Our interest in architecture is motivated by our past research [26, 25], as well as the high level of responses that included examples of technical debt as architecture choices and architecture drift in the data we collected. We examined what problems it might cause and what such issues looked like. We found that architectural choices have the greatest impact based on our analysis of the closed and open-ended questions and interview data.

6.1 Architectural Choices Are Key

We asked participants ($n = 544$) to rank a randomly ordered list of 14 choices (shown in Figure 4) "with respect to the amount of debt (1 = high, 14 = low) they represent on this project." These choices reflect different possible sources, including code, requirements, and architecture, that emerged from several workshops, detailed in [17]. There are a number of ways to measure what the top choice was. Figure 4 shows the percentage of times that respondents ranked a choice in the top three choices. *Bad architecture choices* stood out from others at 296 of the top three responses (54%). In terms of the mean rank, the top three were *Bad architecture choices* (mean rank 4.3), *Overly complex code* (6.0), and *Lack of code documentation* (6.5). For the median, we have the same top three choices, with values 3, 5, 6, respectively. Finally, we conducted a Friedman rank sum test [5], which rejected the null hypothesis that all items are ranked equally ($df = 13, p < 2.2E6$).

Architecture choice was a code in the open-ended questions that we defined as a combination of intentional shortcut and poor decision in hindsight. It describes choices (intentional or not) made about high-level project design, including library choices, framework use, and reference architectures (such as JEE). 73% of the open-coded examples fall into this category. For example, one response gave as an example "...'platform' was not designed with scalability in mind so transferring design from a traditional microprocessor + external memory system to a microcontroller single chip system required about 4 man-years of work."

Definitions of technical debt from McConnell [23] and Cunningham [4] both support the notion of a shortcut for expediency that is interpreted mostly as "bad code." Our examples offer cases different from bad code, since decisions are made earlier and involve more strategic design. For example, "**(B2)** the work that we're doing now to introduce a service layer and also building some clients using other technology is an example of, you know, decisions that could have been done at an earlier stage if we had had more time and had the funding and the resources to do them at the time instead of doing it now."

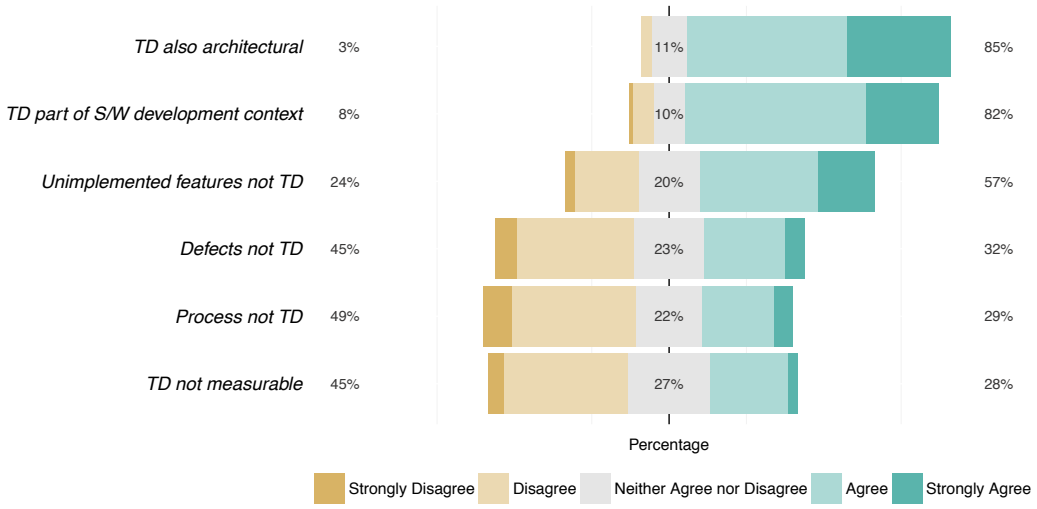

Figure 3: Sources of technical debt. Percentages indicate SD + D, N, and A + SA, respectively.

We heard a similar architecture focus in our interviews. Five of seven participants told stories about architecture choices in the context of a heavy emphasis on fast delivery of features and limited budget. Participants framed these choices in terms of development veering from an important architectural decision (in the form of a pattern or application framework) that was no longer followed.

One participant offered an example of the model-view-controller pattern: "**(A3)** In retrospect we put messaging/ communication . . . in the wrong place in the model view controller architecture which limited flexibility. The correct implementation would put it at the model layer (supporting communication interaction between models) rather than at the presentation layer. As a result modifying or adding new roles requires more work than it should."

6.2 The Need to Manage Drift Between Source and Original Design

The age of most of the systems represented by the respondents was greater than 3 years (74%), and 28% had more than 1 million SLOC. This time frame and size inevitably led to variance over time from the original vision, observed as technical debt today. This drift is consistent with what Lehman refers to as software entropy (systems needing ongoing maintenance) [18]. Some of the examples for this drift include changing system uses ("over the years, other sites would begin using the system and would require changes to how the workflow operated"); poor initial understanding of requirements, off-the-shelf technology, and quality attribute interaction ("poor business knowledge led to poor system design lead to poor user experience lead to rework"); and prototype-becomes-product antipatterns ("The original objective was simply a proof-of-concept and using good software design practices was not a concern.").

While architecture choices were the greatest source of technical debt, dealing with that debt was more problematic. Both the long life spans of many of these projects and the drift from the original decisions, designs, and documentation make paying down technical debt a challenge. For example, "**(A2)** There were some problems in the infrastructure code where there was originally an architecture in place, but it wasn't necessarily followed consistently. . . . So thought had been given to that, but in the implementation . . . shortcuts were taken and dependencies were not clean."

The challenge of paying down technical debt and its interest charges is also highlighted by codes for requirements and legacy migration, with 15% and 10% frequency, respectively (Figure 2). These codes applied often to definitions of technical debt or examples of systems that were facing high development costs to re-architect to fix previous decisions. These challenges could be due to poor requirements gathering (e.g., "[we had] an unrefreshed approach for capture of infrastructure software requirements") or legacy decisions made without foresight ("our development follows the lifecycle of MS Technologies like .NET, Windows OS etc. A significant effort is wasted in the technology migration without adding 'real' value to the software").

The degree of drift is related to system age. We found a weak association between system age and the perceived importance of architectural issues, using Yule's Q [34] as a measure ($n = 483$, $q = -0.42$). 89% of those with longer lived systems ($\geq$6 years old) agreed or strongly agreed with the notion that architectural issues were a significant source of debt, compared to 80% of those with newer systems ($<$3 years old). A chi-square test of independence ($\chi^2 = 30.512$, $df = 12$, $p = 0.0023$) showed that the two factors (system age and importance of architectural debt) are not independent.

Incurring technical debt is often positioned as either deliberate and strategic or inadvertent and accidental (e.g., Martin Fowler's quadrant[2]). Our results show most debt occurs in the "inadvertent/prudent" quadrant. That is, most of the decisions incurring debt were made long ago by other people. The decision was likely deliberate, and may have appeared correct at the time, but subsequent events led to problems: "Not foreseeing the need of final users to customize results of our [redacted] tools" or "The initial design was intended to support our nearer term needs with regards to performance." What is not happening, in contravention

[2]http://martinfowler.com/bliki/TechnicalDebtQuadrant.html

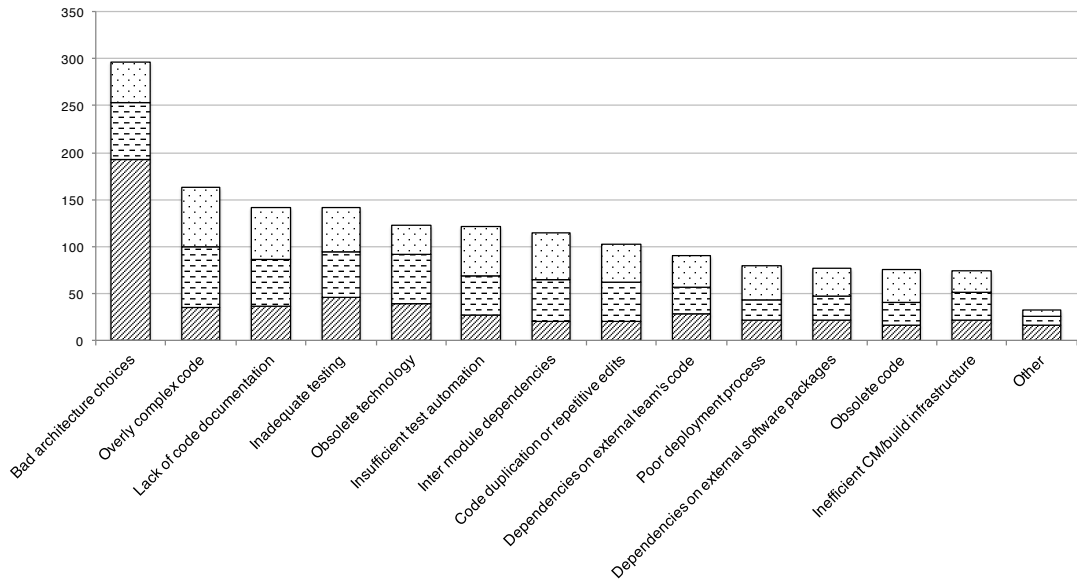

Figure 4: Ranking sources of technical debt. Choice 1 is represented by hatches; Choice 2, dashes; and Choice 3, dots.

to commonly accepted practice in the agile community, is refactoring/re-architecting. As Cunningham says, refactoring or "modifying the program to look as if we had known what we were doing all along."[3] is an important aspect of building and maintaining software, yet that activity rarely appeared in our results. This may be because the teams which are refactoring do not deal much with technical debt, and so our investigation did not reach those teams.

One implication of this drift from original designs is a need for better monitoring of decisions and approaches. Respondents gave many examples of engineers unaware of critical decisions. For example, "**(A1)** We have ... an architectural concept [for managing databases]. And so originally it was supposed to be a series of classes that you would write for a particular feature that would check your database data after you're done. ... And over the years what has happened is because people didn't realize that's what it was supposed to be, we now get basically all kinds of database stuff in there."

An example of what type of monitoring might help is typified in the following anecdote from our interviews: "**(A2)** I believe we started tracking technical debt kinds of issues when we saw them. So that we could get a handle on what's there and we could point to that 'This particular test took longer than expected because we ran into certain kinds of problems.'"

RQ2. Are issues with architectural elements among the most significant sources of technical debt?
Finding 3: Architectural issues are the greatest source of technical debt.
Finding 4: Architectural issues are difficult to deal with, since they were often caused many years previously.
Finding 5: Monitoring and tracking drift from original design and rationale are vital.

[3]http://www.c2.com/cgi/wiki?WardExplainsDebtMetaphor

7. MANAGING TECHNICAL DEBT

RQ3 asked, "Are there practices for managing technical debt?" In particular, we wanted to understand what tools and practices might be used and whether those tools were adequate. Contrary to the increased emphasis of research, consulting, and tool vendors on measuring technical debt, tooling and measurement did not appear much in our data.

7.1 Few Systematic Management Practices

65% of respondents had no defined technical debt management practice, and of the remaining respondents, 25% managed it at the team level ($n = 490$, Figure 5). While there is no explicit standard approach for managing technical debt, there is some management of technical debt within existing processes in many cases (e.g., 60% track technical debt as part of risk processes or backlog grooming ($n = 480$, Figure 6). We asked about tool use ($n = 482$), and 41% do not use tools for managing technical debt (26% have no opinion; only 16% thought tools were giving appropriate detail). For our question concerning who is aware of technical debt, our respondents—most of whom are developers, architects, or program managers—said that executives and business managers were largely unaware (42%). Only 10% said their business managers were actively managing technical debt.

7.2 Tools Are Inadequate

Figure 6 suggests that tool use in identifying technical debt is low (16%, $n = 480$). A substantial number of respondents (27%) do not identify technical debt: "fail first, identify technical debt as cause," or "when the schedule exploded." Where tools are used, issue tracking predominates (25% explicitly track it). Some respondents reported using social processes (using architecture evaluations, risk management, or retrospectives). Furthermore, the open-coded data (Figure 2) indicated that awareness is a big obstacle to managing technical debt. This suggests that making techni-

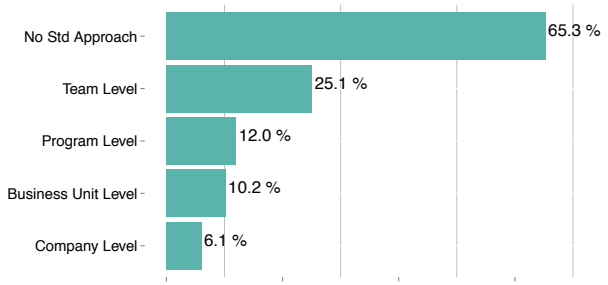

Figure 5: At what level is technical debt management standardized?

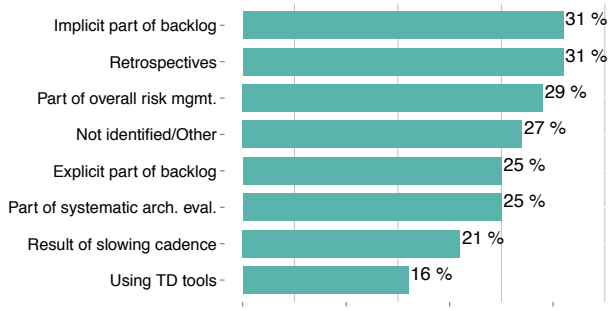

Figure 6: At what point do you identify technical debt?

cal debt visible and measurable may be a key gap in practice.

Tools specific to managing technical debt were rarely used to manage architectural issues. Some were installed, but the complexity of configuring them or interpreting results meant that they sat unused. We collated responses to an open-ended question on tools ("What tools, if any, are used to analyze technical debt on this project?"; $n = 242$) into the most-frequently cited tool categories, seen in Figure 7. Issue trackers, which include tools such as Redmine, Jira, and Team Foundation Server, were the most prevalent (28%). After that, no tool category exceeded 11%, including dependency analysis (e.g., SonarQube, Understand), code rule checking (e.g., CPPCheck, Findbugs, SonarQube), and code metrics (e.g., Sloccount). 50% of respondents said that no tools were used. A related question on tool use ($n = 482$) found that 41% of respondents did not use tools, and only 16% said tools gave enough detail.

The interviews revealed that tools produce too many false positives, which is a well-known challenge [2], and the tools often require context-specific tailoring. For example, tools do not know what areas of code are poor quality, but will never be fixed for other reasons, and thus should be ignored. They need a way to better narrow and focus static analysis results when modifiability is a key concern.

Outputs from tools do not deal well with the challenge of making management aware of the problem. For example, "**(B1)** regarding static analysis we have the source code static analysis tools, but this is to assure proper quality of source code. But how architectural changes are impacting I don't know. And, in fact, this is something we don't do." Furthermore, those tools are difficult to set up: "**(C1)** it showed up on Jenkins - the CI server - there's a billion little warnings. And so it seems a little bit overwhelming." But it was only after explicitly identifying the issue that resources were available: "**(A2)** Yeah, so once a structural problem was identified, we were able to take that to the program management and get authorization and kind of buy-in to tackle that problem directly. And we were able to say, 'Because of the way this is currently structured, we keep encountering these problems and it's making it more expensive to maintain' so we like focused efforts going."

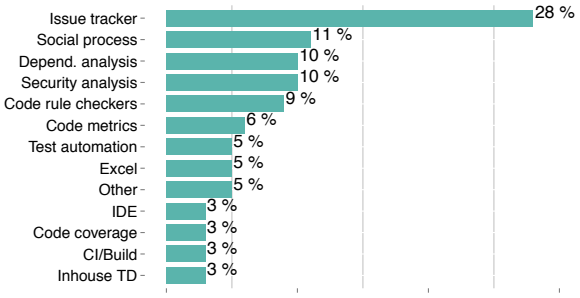

Figure 7: Tool use as percentage of answers, with "Unknown" and "None" excluded.

So why do static analysis tools not help with management of technical debt, and what is taking their place? Our interviews suggest a wide range of potential architectural issues. Due to the nature of the problem, some of these tools may be a good fit for module static analysis and some may not. In many cases, technical debt remains something that at best is managed with identifying tags on the issue tracker. For example, "[we track] occasionally by explicit tech debt items, usually by pain, or not at all."

RQ3. Are there practices and tools for managing technical debt?
Finding 6: Tools do not capture the key areas of accumulating problems in technical debt.
Finding 7: From a developer's perspective, management remains largely unaware of technical debt and the value of managing it.

8. DISCUSSION

As a result of our empirical investigation, we found the following:

1. The technical debt metaphor is useful and commonly understood at an abstract level to convey urgency about accumulating software costs.
2. Apart from architecture, software professionals do not agree on which other project elements are sources of technical debt.
3. Architectural issues are the greatest source of technical debt.
4. Architectural issues are difficult to deal with, since they were often caused many years previously.
5. Monitoring and tracking drift from original design and rationale are vital.
6. Tools do not capture the key areas of accumulating problems in technical debt.

7. From a developer's perspective, management remains largely unaware of technical debt and the value of managing it.

Interviewees and survey respondents placed significant emphasis on describing perceived root causes of technical debt (e.g., uneducated developers, time pressure) that are outside of their control. Consequently, technical debt management is largely reactive since it often must cause significant pain on multiple fronts before it is addressed. Fixing root causes requires different strategies than fixing the current state of the system. This is an area where ongoing and future research on technical debt should contribute, providing clarifications on separating analysis techniques that focus on the artifacts of the system to fix the problem today, from process and awareness approaches to to avoid making early decisions that will incur technical debt later in the life cycle.

8.1 Technical Debt Timeline

We propose a simple timeline approach as an aid to identifying and understanding technical debt, particularly debt caused by variation from an original design. The timeline captures the states that a particular technical debt issue goes through. We identify some key points, shown by numbers in Figure 8.

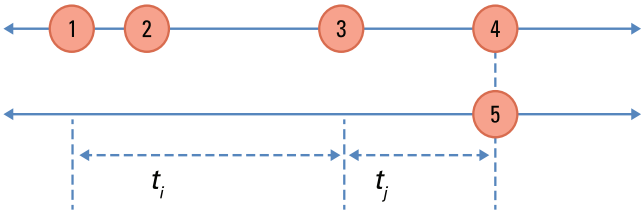

Figure 8: Technical debt timeline, with time increasing to the right.

1. The time when technical debt is taken on (incurred). For example, rather than investing in identifying common services, developers copy and modify code. The respondents could mostly identify architectural issues that map to the point where the debt starts. Beyond these, most examples stayed at the abstract level, such as inexperienced developers writing inadequate code that confuses the developers and prohibits taking concrete action against the debt.

2. The time when technical debt is recognized and traced back to the source. Ideally, this should be visible to both engineers and management and overlap with the time that technical debt is taken on (Point 1). The respondents reported identifying technical debt at different points along the spectrum, as part of the development backlog (56%), in retrospectives (31%), and as a result of slowing project cadence (21%). We found tools are not heavily used to analyze technical debt; those who used tools most frequently used issue trackers and then dependency-focused metrics tools. However, in many cases the metrics tools were run to check a box for management, and teams took a one-size-fits-all approach to running generic reports. Teams often became overwhelmed with the results and threw out the data.

3. The ideal time to plan and re-architect to pay back the debt. This is a function of the ongoing cost of the debt accumulating during the interval t_i. 61% of the respondents agreed or strongly agreed that technical debt is strategically used to support business objectives. Yet most of their examples showed that technical debt is handled after reaching this tipping point where paying back the debt can exceed the benefit of the features developed at the expense of the debt.

4. Organizations decide whether to pay back some or all of the debt. The interval, t_j, between Point 3 and Point 4 is often the time when organizations recognize the pain, which is what we saw in our survey and interview responses. Respondents know that the debt exists, but they have no management strategies for dealing with the debt. As noted earlier, a plurality of respondents (66%) do not pay down debt or pay it down only when it becomes a roadblock. The accumulated issues of the technical debt now exceed the initial perceived short-term benefits.

5. Organizations decide to deliberately manage the rest of the existing and future debt. They ideally begin monitoring the accumulating technical debt and accounting for it during planning cycles.

A timeline perspective helps bring visibility and understanding to the causes, sources, symptoms, and consequences so they can be managed. One of our respondents recognized this challenge: "I can see that a large challenge is corralling what different people mean by technical debt. Based on the questions, I wonder if my definition and thoughts are broader than most; maybe that is limiting what I believe I can affect." Mapping techniques for identifying and tracking technical debt can assist with determining how tools can benefit development teams. Our future work includes validating this time line and mapping analysis approaches onto it.

8.2 Study Limitations

External validity: Our corporate partners are large organizations dealing, in most cases, with long-lived, complex software and hardware systems built for external customers. Smaller, self-contained teams developing product software or doing greenfield development would likely be less concerned with legacy decisions. We would not expect to hear much about architectural technical debt from teams that have the time and management support to follow established design best practices, or that have no legacy code with which to contend. There is a chance that the finding of architecture as problem may be a Hawthorne effect of our institution being widely known as an architecture research institute. However, none of the open-ended questions, nor the preliminary text, mention architecture as an important issue. It was only in the interviews that we specifically asked about architecture. Our results reflect the views of people who presumably had an interest in the issue of technical debt. However, we attracted participation from a wide range of experience, system size, and domains.

Internal validity: We checked our inter-rater reliability by having two of us code each response to a survey question or interview transcript, then comparing the codes using Cohen's kappa statistic. This was 0.45, which is low to moderate agreement. We resolved disagreements between raters

by always choosing from the first rater. We confirmed the sensitivity to architecture results by noting that two survey questions independently query for architecture as a choice. There is a possibility of acquiescence bias to truisms like "architecture is important," but we asked for specific examples in order to avoid this. Interview respondents were selected opportunistically and may not be typical.

Construct validity: Likert scales are one-dimensional and assume that respondents can accurately map their responses to a question into that dimension (e.g., strongly agree or disagree). In some cases, since technical debt is a complex concept, this may not be realistic. Survey rank questions may not be 100% mutually exclusive and exhaustive, although we tried to ameliorate this with our pilot survey.

9. CONCLUSION

Our findings tell us the following:

- Software practitioners agree on the usefulness of the metaphor (Finding 1), notwithstanding different interpretations of what makes up technical debt in particular contexts (Finding 2). There is consensus on McConnell's definition of "a design and construction approach that is expedient in the short term" [23].
- Our data and analysis strongly support that the leading sources of technical debt are architectural choices (Finding 3), answering our second research question. Architectural design decisions take many years to evolve; hence they are difficult to deal with (Finding 4), and managing this drift is vital in managing technical debt (Finding 5).
- Developers perceive management as unaware of technical debt issues (Finding 7), and they desire standard practices and tools to manage technical debt that do not currently exist (Finding 6).

We suggest that research in technical debt tooling focus on monitoring the gap between development and architecture, improving ongoing architecture analysis and conformance. Tooling is a necessary component of any technical debt management strategy. We offered the technical debt timeline as a way to map discovered issues in order to guide a management strategy.

Continued work moving technical debt from metaphor to practice is an important challenge. As one of our respondents said, "I am pleased to see that [our organization] (and hopefully the industry as a whole) is taking an interest in the issue of technical debt. Traditionally it has been very difficult for developers to relay this concern to leadership and customers."

10. ACKNOWLEDGMENTS

Many thanks to the many participants in our surveys and interviews, as well as our industrial partners.

 This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002252

11. REFERENCES

[1] N. Alves, L. Ribeiro, V. Caires, T. Mendes, and R. Spínola. Towards an ontology of terms on technical debt. In *International Workshop on Managing Technical Debt*, 2014.

[2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, Feb. 2010.

[3] Z. Codabux and B. Williams. Managing technical debt: an industrial case study. In *International Workshop on Managing Technical Debt*, pages 8–15, San Francisco, 2013.

[4] W. Cunningham. The WyCash portfolio management system. In *Object-oriented Programming Systems, Languages, and Applications*, pages 29–30. Vancouver, 1992.

[5] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

[6] D. Falessi, M. Shaw, K. Mullen, and M. Stein. Practical considerations, challenges, and requirements of tool-support for managing technical debt. In *International Workshop on Managing Technical Debt*, pages 16–19, San Francisco, 2013.

[7] F. Fontana, V. Ferme, and S. Spinelli. Investigating the impact of code smells debt on quality code evaluation. In *International Workshop on Managing Technical Debt*, pages 15–22, Zurich, 2012.

[8] M. Fowler. Technical debt quadrant. `http://martinfowler.com/bliki/TechnicalDebtQuadrant.html`, 2009.

[9] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *International Conference on Software Engineering*, 2015.

[10] I. Griffith and C. Izurieta. Design pattern decay: an extended taxonomy and empirical study of grime and its impact on design pattern evolution. In *Conference on Empirical Software Engineering and Measurement*, Baltimore, MD, 2013.

[11] I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, and B. Williams. The correspondence between software quality models and technical debt estimation approaches. In *International Workshop on Managing Technical Debt*, pages 19–26, Victoria, 2014.

[12] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. DaSilva, A. Santos, and C. Siebra. Tracking technical debt–an exploratory case study. In *International Conference on Software Maintenance*, pages 528–531, Williamsburg, 2011.

[13] C. Izurieta, A. Vetro, N. Zazworka, Y. Cai, C. Seaman, and F. Shull. Organizing the technical debt landscape. In *International Workshop on Managing Technical Debt*, pages 23–26, 2012.

[14] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *SIGSOFT Symposium on Foundations of Software Engineering*, Raleigh, NC, November 2012.

[15] H. Kniberg. Good and bad technical debt (and how

TDD helps), http://blog.crisp.se/2013/10/11/henrikkniberg/good-and-bad-technical-debt, October 2013.

[16] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software Special Issue on Technical Debt*, 29(6):18–21, 2012.

[17] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi. Technical debt: towards a crisper definition; report on the 4th International Workshop on Managing Technical Debt. *SIGSOFT Softw. Eng. Notes*, 38(5):51–54, 2013.

[18] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[19] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2014.

[20] Z. Li, P. Liang, and P. Avgeriou. Architectural debt management in value-oriented architecting. *Economics-Driven Software Architecture*, pages 65–86, 2014.

[21] E. Lim, N. Taksande, and C. Seaman. A balancing act: what software practitioners have to say about technical debt. *IEEE Software*, 29(6):22–27, 2012.

[22] R. Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5):9:1–9:13, 2012.

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

[24] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic. Mapping architectural decay instances to dependency models. In *International Workshop on Managing Technical Debt*, pages 39–46, San Francisco, 2013.

[25] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. In search of a metric for managing architectural technical debt. In *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (ECSA)*, pages 91–100, Helsinki, Aug. 2012. IEEE.

[26] R. L. Nord, I. Ozkaya, R. Sangwan, J. Delange, M. Gonzalez, and P. Kruchten. Variations on using propagation cost to measure architecture modifiability properties. In *International Conference on Software Maintenance*, pages 400–403, Eindhoven, September 2013.

[27] K. Schmid. A formal approach to technical debt decision making. In *International Conference on Quality of Software Architectures*, pages 153–162, Vancouver, British Columbia, Canada, 2013.

[28] C. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[29] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman. Technical debt: Showing the way for better transfer of empirical results. In *Perspectives on the Future of Software Engineering*, pages 179–190, 2013.

[30] C. Siebra, G. Tonin, F. DaSilva, R. Oliveira, L. Antonio, R. Miranda, and A. Santos. Managing technical debt in practice: An industrial report. In *International Symposium on Empirical Software Engineering and Measurement*, pages 247–250, 2012.

[31] R. Spinola, N. Zazworka, A. Vetrò, C. Seaman, and F. Shull. Investigating technical debt folklore: Shedding some light on technical debt opinion. In *International Workshop on Managing Technical Debt*, 2013.

[32] A. Strauss and J. M. Corbin. *Basics of Qualitative Research : Techniques and Procedures for Developing Grounded Theory*. Sage Publications, Thousand Oaks, 1998.

[33] E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[34] G. Yule. On the methods of measuring association between two attributes. *Journal of the Royal Statistical Society*, LXXV:579–652, 1912.