Got Technical Debt?

Surfacing Elusive Technical Debt in Issue Trackers

Stephany Bellomo, Robert L. Nord, Ipek Ozkaya, and Mary Popeck

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
1-412-268-5800
sbellomo, rn, ozkaya, mpopeck@sei.cmu.edu

ABSTRACT

Concretely communicating technical debt and its consequences is of common interest to both researchers and software engineers. In the absence of validated tools and techniques to achieve this goal with repeatable results, developers resort to ad hoc practices. Most commonly they report using issue trackers or their existing backlog management practices to capture and track technical debt. In a manual examination of 1,264 issues from four issue trackers from open source industry and government projects, we identified 109 examples of technical debt. Our study reveals that technical debt and its related concepts have entered the vernacular of developers as they discuss development tasks through issue trackers. Even when issues are not explicitly tagged as technical debt, it is possible to identify technical debt items in these issue trackers using a categorization method we developed. We use our results and data to motivate an improved definition and an approach to explicitly report technical debt in issue trackers.

CCS Concepts

Software and its engineering → Software creation and management → Software post-development issues → **Maintaining software**

Keywords

Technical debt; software anomalies; issue tracking; text categorization; software design.

1. INTRODUCTION

What is technical debt? Why identify technical debt? Shouldn't these issues be captured as defects and bugs? The inability to answer these questions empirically, supported by a software economics theory, can result in technical debt attaining a legendary status [31]. We know its value as a metaphor, and we hear stories from developers and project folklore about its symptoms and their consequences, but can we see, describe, and hold the thing itself as a concrete software development artifact? While progress is being made toward refining our understanding of technical debt theoretically, data-driven studies to contribute to theoretical research endeavors lag behind.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.
Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MSR'16, May 14-15, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4186-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901739.2901754

Results of our recent, broad practitioner survey of 1,831 software engineers and managers demonstrate that they share a common understanding of the concept of technical debt [12]. According to participants, lack of proven tool support to accurately identify, communicate, and track technical debt is a key issue and remains a gap in practice. In the absence of validated tools to concretely communicate technical debt and its consequences, developers resort to practices that they are familiar with.

More than half of the participants in our survey reported using issue trackers to communicate technical debt either explicitly ("technical debt" is mentioned) or implicitly (the concept of "technical debt" is discussed but not explicitly mentioned). This is consistent with anecdotal feedback from our own experiences of working with organizations as well as case studies represented in literature [34].

Intuitively it makes sense for issue trackers to serve as an entry point for communicating technical debt since developers use issue trackers as one tool to manage task priorities. To understand how issue trackers are used to communicate technical debt by software developers, we conducted an exploratory study of four issue trackers, including the Chromium [8] and Connect Health IT Exchange [10] open source projects and two government IT projects for which we are aware of technical debt issues.

We address the following questions:

- RQ1: Do developers use the term *technical debt* explicitly when discussing issues and tasks in their issue trackers?
- RQ2: Can technical debt items be discovered systematically within issue trackers?
- RQ3: What are the distinguishing characteristics of technical debt items discovered in issue trackers?

We identified 109 examples of technical debt from a sample of the 1,264 issues in the issue trackers we studied and evaluated them with experts and the developers of the systems when applicable. A summary of our findings include the following:

Finding 1: While technical debt items were not labeled explicitly in the issue trackers we studied, we identified 58 examples where developers explicitly use the term *technical debt* and related concepts to understand an issue. **Concepts related to technical debt, such as take-on debt, accumulate debt, and pay-back debt, have entered the developers' vocabulary, and they are using issue trackers to communicate technical debt in an ad hoc manner.**

Finding 2: We developed and used a classification approach to find additional examples where developers articulated concerns related to technical debt, but did not use the term. Using this approach, we identified 51 more examples of technical debt. Many issues could not be classified because **developers do not always clearly identify the consequences of not paying down the debt.**

Finding 3: We analyzed the technical debt issues that we found for generalizable characteristics:

- Studying the characteristics of project indicators recorded in the issue trackers (time open, number of watchers, priority) demonstrate that more **data analysis is needed to find consistent values that may show a relationship between these characteristics and technical debt**.
- The examples provide **early results for design areas where technical debt mostly occurs**, including consequences of dead code, duplicate code, and API mismatches.
- The promise of understanding technical debt is to use design choices strategically to intentionally trade off speed and development effort with minimal compromise of quality. The reality is that the issues we found are **mostly the result of unintentional design choices** and surface in issue trackers when their unintended consequences become visible.

Our findings demonstrate that when developers refer to technical debt and related concepts in the issue trackers they also point their finger at where the problem is in the code. They are not just talking about a high-level conversation piece of system quality. Consequently, technical debt is an artifact of software development, similar to design, code, and defects. Based on these findings, we recommend that to take best advantage of technical debt and pay it back before the debt grows, its definition should concretely map to development artifacts. We propose the following definition:

> Technical debt is design work relating to software units that have evidence of present or anticipated accumulation of extra work.

Design work is manifested through implementation and changes to supporting work products such as code, data, build scripts, and test suites. Conversely, technical debt is not work related to a non-software unit (e.g., requirement, documentation, process concern), nor is it a low-impact, executable artifact change with minimal consequence if not fixed (e.g., uncommented code).

In this paper, we present our analysis results and recommendations for how to report a technical debt item.

2. APPROACH

We conducted our study on four projects. The data sets included subsets of items from the Connect Health IT Exchange (Connect) [10] and Chromium [8] issue trackers and two government IT projects, Project A and Project B, as described in Table 1.

During data setup we first looked through the data sets for a technical debt label. We also searched for the term "technical debt" and expanded our search for technical debt concepts using terms we extracted from the vocabulary of a cross-section of developers who provided 26 exemplar descriptions of technical debt items.

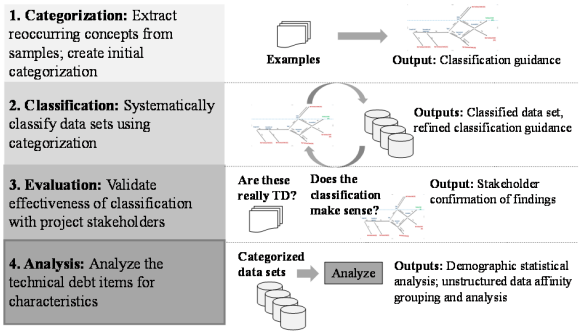

Figure 1. The four research phases followed.

We performed our study in four phases as summarized in Figure 1.

In Phase 1, we extracted recurring technical debt concepts and created categories to classify issues as technical debt, even if not explicitly tagged as such by developers. In Phase 2, we used the categories to manually classify issues as technical debt or not.

In Phase 3, we evaluated whether we were able to systematically discover technical debt within issues trackers correctly by talking to the developers of the systems under study. In Phase 4, we looked across all the identified technical debt examples for distinguishing characteristics that might serve as consistent indicators of technical debt.

We selected Connect for the study because it is an active open-source, open-contribution project with public access to its Jira repository. Connect aims to enable secure, electronic health data exchange among health-care providers, insurers, government agencies, and consumer services in the United States by establishing a gateway between health information systems and organizations. It is based on service-oriented architecture design principles and web service interfaces [9]. It has been in development and use since 2008.

Projects A and B are government-related data sets from ongoing clients, selected because they have a history of releases focused on rework to address technical debt. Project A is a mission-critical compliance tracking system for a large government organization that centralizes the data gathered from several sources into one nation-wide system. This Oracle client-server system has been in

Table 1. Projects studied by phase.

	Data set	Source	Filter criteria	# Records analyzed
Setup	Chromium	Google issue tracker	Text search "technical debt"	56
	Connect	Jira	Text search "technical debt"	15
	Technical debt survey	Examples (as text)	N/A	265
Phase 1 Categorization	Connect	Jira	2012, first 200 records	200
Phases 2–4 Classification, evaluation, and analysis **Total:** 727 issues	Connect	Jira	March 2012	286
	Project A	Jira	Defects/CRs Sep. 2010 to Dec. 2014	86
	Project B	FogBugz	All year 2013	193
	Chromium	Google issue tracker	M(ilestone): 48 OS: All Stars (watchers) > 3	163
			Total	1,264

use for over 15 years. Since the initial release, over 1.2 million work assignments have been created in the system. Most issues are entered into the issue tracker by the product manager.

Project B is an IT system for federal government employees to protect the system environment by monitoring physical and environmental conditions against cyberattack. The software has been in use for 4 years and serves 125 users. Key technologies used include web services, embedded OS on BeagleBone Black, Socket IO services, and a customized sensor data communication protocol and collection mechanism. Programming languages used include Python, JS, HTML, CSS, C, C++, and DB: PostgreSQL. Several stakeholders create issues in the issue trackers, but the main person entering them is the product owner/manager.

In all three data sets, we have additional technical information about known refactoring activities related to technical debt and access to developers who can serve as experts to validate our findings. We included Chromium as our fourth and control data set due to the relatively robust issue-tracker practices followed.

The subset of the issues studied from these projects were selected randomly to minimize bias. Table 1 summarizes the subsets of the issues selected from these projects. The issues were not triaged based on any particular classification that the specific project may impose, such as bug, defect, and new feature. Some data sets had more disciplined issue-tracking practices than others, such as tracking priority, release assigned, number of watchers, code commits, and the like. All data sets had sufficient description of the issues to allow researchers to make classifications and judgments.

2.1 Phase 1: Creating Categories

In Phase 1, we extracted concepts related to technical debt following the concept extraction approach described in [11]. The sample data set was prepared by copying a subset of Connect issues into a spreadsheet where each line represented an issue tracker record. The record contained both predefined field data (e.g., type, priority, duration) and issue descriptions.

Two researchers acting as technical debt subject-matter experts independently tagged each issue. The researchers did not have domain knowledge about the project. Three decision outcomes were possible: yes, no, or cannot determine. Researchers were asked to highlight portions of the issue relevant to their decision, capture recurring concepts (e.g., abstract concepts [11] such as executable artifact or design concern and specific concepts such as duplicate code or incorrect functionality), and provide rationale for categorization. After each categorization round, we met to resolve discrepancies and improve the categorization.

We repeated this categorization process three times, each time elaborating and refining the categorization method. We conducted two rounds of categorization using the first 100 records. Then we did a third round of categorization using the second 100 records of the Phase 1 data set. We set a target threshold of achieving 80% rater consistency before exiting Phase 1 to allow some natural rater inconsistencies, mostly arising from cases where one researcher thought there was not enough information while another used expert judgement.

A known expert in the field of software engineering and technical debt, external to our research team, assessed the results of our Phase 1 classification. The expert categorized a random sample of our issues without knowledge of the software system under study or the approach we used to categorize the sample. As he categorized each entry, we asked him to discuss his rationale aloud and extracted concepts as feedback to inform our categories and guidance.

The output of Phase 1 is an initial categorization summarized in Figure 2.

2.2 Phase 2: Classifying Issues

In Phase 2, pairs of researchers manually classified the four selected data sets using the categorization developed in Phase 1 (Figure 2) and the following steps:

Step 1. One researcher prepared the data sets by selecting a random subset of issues.
Step 2. Two reviewers independently classified the issues.
Step 3. One researcher consolidated results into a single spreadsheet that highlighted agreements and discrepancies.
Step 4. Researchers together discussed classification discrepancies and extracted recurring concepts.

The two major outputs of Phase 2 include (1) a data set of issues classified as technical debt or not and (2) refined classification guidance.

2.3 Phase 3: Evaluating Results

In Phase 3, we walked through the identified technical debt records with project representatives from Connect, Project A, and Project B. We started by asking them whether they were familiar with the concept of technical debt and if their project had technical debt. We did not offer our definition of technical debt to avoid biasing them, but allowed them to offer us theirs to ensure that there was sufficient understanding for them to proceed with the task.

We asked the project representatives to indicate whether they agreed with our assessment of identified technical debt issues. We also asked if we had missed any examples of technical debt. We did not reach out to Chromium developers. The issues we looked at from Chromium are a representative but small subset of all the Chromium issues.

We asked a second expert in the field of software engineering and external to our research team to use our categorization under the same conditions to see if he would generate the same results. We did this to ensure that unintentional bias of the researchers did not influence the integrity of the results and that the classification and its guidance are understandable, logical, and easy to follow. The expert received instructions for conducting the study, read through the guidance, and had an opportunity to ask questions. The expert tagged a sample data set from Project A. The expert was then given a post-experiment questionnaire that included questions gauging quality of the data, ease of use, and quality of guidance for classifying the issues. We then incorporated several minor improvements into the guidance document.

Outputs of Phase 3 resulted in our final data set of items classified as technical debt, 51 from the four project data sets.

2.4 Phase 4: Analyzing Tagged Issues

In Phase 4, we analyzed the selected issues for distinguishing characteristics that potentially identify technical debt. We examined structured data in the issue tracker's predefined fields such as priority, duration open, number of watchers, and number of linked issues. Our motivation was to see if we could observe trends. For example, did technical debt issues take longer to resolve, have higher priority, or cause changes that rippled through a number of issues, hence suggesting additional time spent dealing with consequences?

We also examined the unstructured text in fields such as summary and description. A pair of researchers followed an open-coding approach [11], identified reoccurring design concepts from the records in which we identified technical debt, and affinity grouped this data. We discuss these results in Section 5.We also extracted concepts that signal accumulation of consequences of debt. There was variation among these concepts such that we could not create meaningful affinity groups. The implications of this result for future research are discussed in Section 6.

Outputs of Phase 4 include analysis results of predefined fields (e.g., priority, opened date, closed date) and text for design concerns (organized by affinity grouping) and intentionality.

3. TECHNICAL DEBT IN ISSUE TRACKERS

None of the four issue trackers used technical debt as a predefined label or as an issue type. Consequently, we searched for the key word "technical debt" in each data set. Only Chromium and Connect returned positive results, a total of 71 (Table 2). Both Connect and Chromium include data that dates back several years. The use of the term first appears in 2010 in Connect and 2012 in Chromium.

To eliminate false positives, one researcher read all of the issues and discarded 7 of the 15 Connect issues and 6 of the 56 Chromium issues, resulting in 58 examples where technical debt was explicitly used to refer to a concrete system problem that caused rework. Reasons for discarding issues included issues that were duplicate entries and documentation tasks (e.g., a blog post, a software architecture design document). In addition, two issues were discarded because the developer discussion indicated that the issue did not describe technical debt.

For example, we discarded the following because it was a documentation task, despite explicit reference to technical debt. Hereafter, we refer to the issues by the project name followed by the issue number:

> [Connect #1650] The brief *blog post* should describe the *detriments of technical debt*, the balances of keeping vs. jettisoning, and how the CONNECT team approached the decision and what we did about it - compromise and balance.

While this issue suggests that the developers are aware of technical debt, the issue is a blog post task. It does not reveal where the debt was, how it accumulated, and how it was paid; hence it does not represent a technical debt item.

Another example of a discarded issue is one in which the developers conclude that the issue is a "legitimate bug":

> [Chromium #496267] The NCN registers for connectivity messages iNetworkChangeNotifierAutoDetect::onApplication, but fails to register when the device starts. …*This is a legit bug, not cleanup/refactoring/technical-debt-reduction*.

Table 2. TD discussion occurrence.

Project	# Issues	# Times TD key word found	Date first occurred
Connect	5,186 since July 2009	15	Jan. 2012
Project A	86	0	NA
Project B	193	0	NA
Chromium	>390,000 since Sep. 2008	56	Oct. 2010

Errors that are visible to users and result from coding mistakes are bugs or defects and should not be confused with technical debt. In this example, Chromium developers demonstrate that they are well aware of these concepts and declare that this issue should be handled as a defect, not as cleanup or refactoring.

In issues where technical debt discussions were explicit, we observed developers using concepts related to technical debt. For example, they referred to "taking on debt" when there was a clear design trade-off, "accumulation of debt" when issues were not fixed on time, or "paying off technical debt" when the developers wanted to act or had acted on issues in the system. The following examples demonstrate how developers referred to these related concerns:

> [Chromium #402086] The change looks larger and more complex than it really is: it's mostly plumbing and changes to method signatures adding to the line count. It's been in canary for a week without issue. Landing this will enable WebView to *shed some technical debt*, which is quite a big benefit for us.

> [Connect #Gateway-1942] To address code added into CONNECT to support Deferred Patient Discovery as a Reference Implementation in 3.0 and enhanced as part of 3.2. *This code is now considered technical debt* since the only approved version … supported in production prior to the approved Summer 2011 is version 1.0 which doesn't include functionality for Deferred Patient Discovery.

> [Chromium #500991] However this change is somewhat *dwarfed by the technical debt* that needed to be paid off in order to allow this new change to be tested…

In these examples, developers consciously and correctly refer to development and design tasks required to deal with technical debt and its consequences in their discussions. They talk about specific code snippets, design trade-offs, mapping testing scripts, and their alignment and the consequences. In particular, among Chromium developers an unspoken process seems to have emerged in dealing with technical debt that we could infer from these discussions:

> [Chromium #243948] *Paying off technical debt* becomes a *higher priority, not lower*, when in those rare cases it must be deferred. Tests are not a 'nice to fix' feature. Raising to Pri-1.

These examples show results from a keyword search on "technical debt." We also explored whether other terms that developers used might be useful in extracting examples of technical debt from issue trackers. To broaden our search terms, we analyzed 265 examples from members of a large multinational organization who responded to a survey about technical debt [12]. From these examples, we created a list of search terms that includes the following: duplicate, custom, workaround, inconsistent, hack, legacy, refresh, rewrite, cleanup, refactor, and refresh. Section 6.2 summarizes the results of this analysis.

Based on these examples found in Connect and Chromium issue trackers, we conclude that

- while ad hoc, developers use issue trackers to communicate technical debt
- technical debt concepts have entered developers' vocabulary
- once developers are aware of the symptoms of technical debt, they respond by examining concrete changes that caused the debt to accumulate, such as code snippets, design decisions about implementation, build and testing scripts, and data models. The linkage of technical debt to a concrete artifact leaves less room for confusion in high-level technical debt discussions.

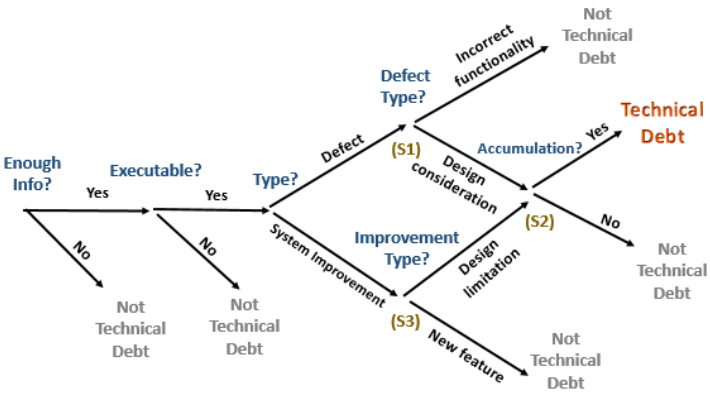

Figure 2. Concepts for classifying technical debt.

4. TECHNICAL DEBT CLASSIFICATION

Experts apply unspoken rules and heuristics when determining whether an issue represents technical debt. We observe this in the results of the examples that we discussed in Section 3 as well as in literature [12][14][24][29]. Our goal in developing the technical debt classification is to capture the expertise and allow repeatable classification of issues. Figure 2 shows the categories that resulted from our classification at the end of Phase 1. Complete guidance for our classification and selected results can be found in [32]. Here we summarize the key decision points with examples.

Enough information: When an issue did not contain enough information, we tagged it as *not technical debt* to minimize biased decision making. These were often one-line issue descriptions that required further context, such as the following example:

> [Connect #Gateway-1616] Update AdapterComponentMpiSecured Service to use PatientDiscoveryFault

Executable or data related: A major source of confusion in dealing with technical debt is overgeneralizing the concept by including related project management activities, such as documentation, requirements analysis, quality assessment, and investigation. For technical debt to be actionable for development teams, it must be related to a concrete artifact of the development, such as code, implementation units, processing units of the executing system, data models, build scripts, and unit tests. We tagged any issue that did not mention a concrete development artifact as *not technical debt*. A good example is the following:

> [Project B #2645] Perform web application security assessment. Ran Netsparker and found 4 issues, 1 major and 3 minor.

Running an assessment tool and examining the issues it reveals do not represent technical debt.

Classification from this point on requires articulation of often fuzzy concepts such as defect, bug, and design concerns. Defects are identified as concerns visible to end users; technical debt tends to be invisible system issues. We separated defects from system improvement issues [17]. In addition, we separated defects as visible incorrect functionality from cases where they were symptoms of an underlying design consideration that may be related to technical debt. Similarly, we separated system improvements as new features from cases where an underlying design limitation impacted the feature request.

Type ➔ Defect type ➔ Incorrect functionality: We found many examples of defects in which the system did not work as expected, such as a tester discovers that a button doesn't work in the UI, the system crashes, or a wrong classification is added. We classified these issues as *not technical debt*. They are visible to the users and represent system errors. Examples include

> [Project A #25] Correct the values for subsystem A to reflect the subsystem b values

> [Project B #265] Update alert authoring UI – 'event window' should be close to 'any rule' checkbox

Type ➔ Defect type ➔ Design consideration: Several defects impacted a quality attribute such as availability, security, or performance. We classified these as design considerations. We also classified as design issues several examples of cleanup activities impacting maintainability. If we also found evidence of accumulation of unintended side effects, or projection that they would accumulate, we then classified these issues as *technical debt* (e.g., duplicate code, nonstandard binding, type mismatch, inconsistent implementation, or unused classes).

An example of a defect that represents a design consideration, but not technical debt, is the following:

> [Project B #2722] …rule engine repeats alerts because of event query, causes the rule engine to keep dragging over the last query…

The researchers tagged this as a defect representing a design consideration because of the implications for the data model, performance implications of the query, and the rule engines. But we did not classify this issue as *not technical debt* because the side effects of accumulating rework and refactoring were not clear.

Type ➔ Improvement type ➔ Feature: New features as system improvements, such as adding a new node to a sensor component or removing a drop-down box, were classified as *not technical debt*. An example is the following:

> [Project B #1485] Filter alert trigger list by date

Type ➔ Improvement type ➔ Design limitation: In some cases, an issue was not a defect or mistake but a system improvement to remedy a design limitation, such as the inability to add a new feature quickly, the current technology not supporting the improvement needed, maintainability issues, or consequences of refactoring. To handle such cases, we introduced the design limitation branch. When evidence of side effects were not clear, even those that clearly mentioned refactoring to remedy a design limitation, we classified the issue as *not technical debt.*

> [Project B #1513] Refactor onclicks in nodes.html into query events

Accumulation: 51 issues were design related and showed some evidence of accumulation such as increased time to make

implementation changes, automated tests not supporting the refactored classes, or security vulnerability. We tagged only those issues for which we could identify an explicit impact of side effects—in other words, accumulating consequences—as *technical debt*. Here is an example from Connect:

> [Connect #Gateway-1631] The re-architecture of the source code to support multiple NwHIN specifications has introduced a new Java packaging scheme. New and existing classes have been moved into these new package folders; however, the previous package folders have been left in place with no class files. No impact to functionality; however, may lead to confusion for users implementing enhancements / modifications to the source code.

Further details from project stakeholders on the issues we classified as *not technical debt* may reveal that they represent technical debt. However, our goal in this study was to uncover those issues that could be classified with available information, then use this output to make progress on a concrete definition of technical debt and an improved reporting mechanism. Data quality of the issue reports is a known concern in such studies; therefore, we erred on the side of false negatives rather than false positives. An issue we discarded may have been technical debt if a subject-matter expert provided further detail, but we aimed to rely on the information available to us through the issue tracker. Our samples represent a starting set to analyze concrete examples of technical debt and its characteristics to help developers communicate and act on such issues.

Table 3 is the summary of our classification of the four data sets. Out of 727 records, we identified 51 as technical debt issues.

We allowed research team members to identify points where they got stuck, represented as S1, S2, and S3 in Figure 2. This surfaced 12 issues that we discussed for future improvements to the classification guidance.

One example where the researchers got stuck is from Project A. There is clearly a design concern about decommissioning a database. However, while the proposed remediation suggests web service implementation to avoid rework later (future accumulation), it is unclear if the current design solution is causing accumulation.

> [Project A #21] Request (made by xx) for read only access to the xx tables in xx database. Requirements are: 1. Web Service implementation a. Since xx is planned for decommission, a database view is not a viable solution. We would like to go with implement it in Web Services to *avoid rework in the future*

We resolved this discrepancy by limiting the scope to evidence on *current* accumulation to avoid biasing the results with researchers' knowledge or interpretation of projects' technical context.

As a result of the several iterations of tagging, discussions, and analysis of the examples, we conclude that

- technical debt exists when design decisions cause unintended work that potentially increases the time to delivery, which we refer to as accumulation. Making accumulation clear is critical in communicating technical debt concretely. In its absence, confusion about whether or not an issue represents technical debt is inevitable.
- technical debt is a design-related concept, as confirmed by the examples we identified.

Table 3. Summary of technical debt classification.

Project	TD	Not TD	Stuck	No agreement	Total
Connect	12	265	1	7	285
Project A	10	74	1	1	86
Project B	13	171	9	0	193
Chromium	16	146	1	0	163
Total	51	656	12	8	**727**

5. CHARACTERISTICS OF TECHNICAL DEBT

We analyzed the 51 examples of technical debt identified in Phase 2 for generalizable characteristics. We looked at both predefined issue fields—including open days, watchers, and priority—and analyzed description text for design concerns and intentionality. We report our analysis results with the questions that we addressed.

Do technical debt issues take longer to close?

We hypothesized that the 51 technical debt issues may take longer to resolve than the 656 non-technical debt issues. We analyzed average days open using mean plots. We calculated days open for each issue in the data set by subtracting closed date from open date. For each project, we divided the data sets into technical debt and non-technical debt sets of issues. We then created subgroupings of 100 days (1 to 100, 101 to 200, and so on), and took the average of these subgroups, and plotted the results on mean plot line charts. These charts allowed us to drill a little deeper in to the data and visually compare the patterns in the technical and non-technical data sets for each project. When we examined the charts, we found that average days open vary widely and do not reveal meaningful patterns from the data sets that we analyzed.

While all projects had large Days Open standard deviations, Chromium and Connect were a little tighter (Chromium, $\sigma = 319$ days; Connect, $\sigma = 251$; Project A, $\sigma = 456$; and Project B, $\sigma = 557$).

Figure 3 shows the cumulative percentage of issues closed for each project, revealing subtle differences in pace of issue closure. Both Chromium and Connect closed 95% or more issues within 2 years compared to Projects A and B, which closed less than 70%. This suggests that issue management practices may be slightly stronger in Chromium and Connect. In addition, for these two projects we found examples in the issue records of language like "technical debt" and "accumulation" in the developer vocabulary.

We conclude that results are not significant to declare days open a distinguishing characteristic of technical debt; however, future analysis in larger data sets with mature issue management practices could yield different results.

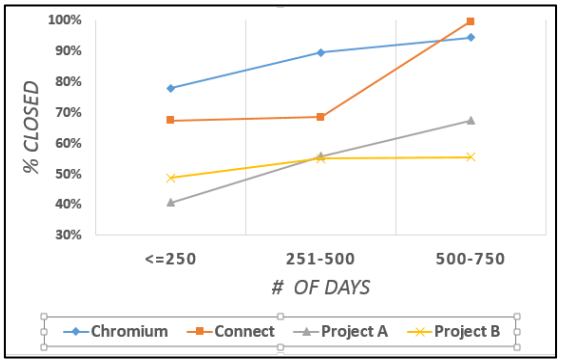

Figure 3. Time issues remain open.

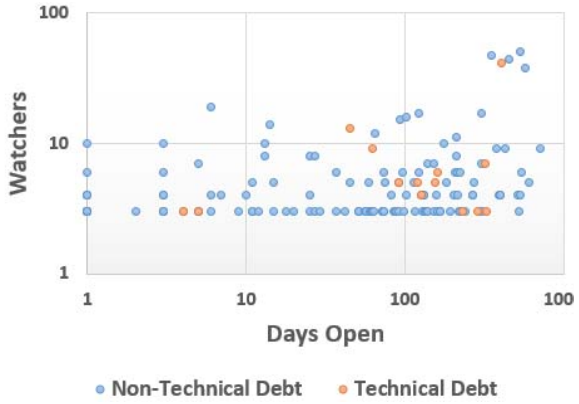

Figure 4. Chromium by number of watchers and days open.

Do technical debt issues have higher numbers of watchers?

Watcher is a measure of the number of people interested in an issue record in the issue tracker. Only Chromium has a fully populated data set for "watcher," so we took a deeper dive into Chromium Watchers. Figure 4 shows technical debt issues in orange and non-technical debt issues in blue.

The patterns of the number of watchers between the two classes of issues are not significantly different. The gap in orange technical debt dots between 8 and 60 days open is likely a random occurrence due to the size of the data set. Therefore, we conclude that we cannot declare a relationship between number of watchers and technical debt from this data set.

Are technical debt issues high priority?

Table 4 compares the issues by priority (1 = highest priority and 3 = lowest). The percentages represent counts of issues with that priority divided by the total count for that row (e.g., 22% of the technical debt issues have a Priority = 1). Both categories have 50–60% of the issues (the majority) assigned to Priority 2. Given this, we do not have evidence to conclude that technical debt issues have higher priority than other issues.

Do the technical debt issues show recurring design concepts?

We analyzed the textual data from the 51 examples of technical debt for recurring design concepts. We created affinity groups derived bottom-up from the issue descriptions (contrary to a top-down approach of creating the concepts first and then classifying them). The resulting affinity groups are shown in Figure 5 with the number of issues that contained the concept as well as the project(s) where we found the concept. If we found the concept in multiple projects, the number of times per project is shown. For example, for the five instances of *event handling*, two were found on the Chromium project and three were found on Connect.

Our resulting data set of identified technical debt items is small; however, it serves as a starting point to do more in-depth analysis of potential issues that may commonly cause unintentional consequences. In particular, refactoring-related consequences—such as dead code, misaligned test and build scripts, and version conflicts—are places to start improving unintentional technical debt accumulation.

Table 4: Analysis of priority.

Issue Type	Priority 1	Priority 2	Priority 3
Technical debt	22%	56%	22%
Not technical debt	24%	50%	26%

Deployment & Build	Out-of-sync build dependencies	3	CN
	Version conflict	1	CN
	Dead code in build scripts	1	CN
Code Structure	Event handling	5	2CH, 3PB
	API/interfaces	5	2CH, 1CN, 2PB
	Unreliable output or behavior	5	4CH, 1PA
	Type conformance issue	3	CN
	UI design	3	PB
	Throttling	2	1CH, 1PB
	Dead code	2	CN
	Large file processing or rendering	2	CH
	Memory limitation	2	CH
	Poor error handling	1	PA
	Performance appending nodes	1	CH
	Encapsulation	1	PB
	Caching issues	1	CN
Data Model	Data integrity	6	PA
	Data persistence	3	PB
	Duplicate data	2	PA
Regression Tests	Test execution	1	CH
	Overly complex tests	1	CH

[a] CH = Chromium, PA = Project A, PB = Project B, CN = CONNECT

Figure 5. Affinity groups of design concerns.

Is technical debt used strategically?

The appeal of technical debt is that it allows development teams to make intentional design trade-offs to accelerate development and revisit them as needed. Yet, 49 of the 51 issues were unintentional design decisions. We provide an example from each of the four affinity groupings from Figure 5.

Deployment & Build: Out-of-sync build dependencies

> [Connect #Gateway-1623] The CONNECT 3.3 release is to be deployed against the 2.1.1 version of the Metro Web Stack. Therefore, the compilation and build dependencies *should* reference the 2.1.1 version of the Metro libraries… Impact to the users enhancing / modifying CONNECT is that they *will not have the correct version* of the Metro Web Stack library for development.

The reference to "will not have correct version" describes the impact of not maintaining accurate build dependencies in the build scripts. The word "should" suggests unintentionality.

Code Structure: Event handling

> [Chromium #294388] The |code| attribute specified in UI Events is intended to accurately identify the physical key associated with a key event. The legacy attribute |keyCode| was previously used by developers for this purpose, but it has problems in that it was *never completely specified* and thus it is *not consistently implemented* across browsers … add a new |code| attribute to WebKeyboardEvent.

The words "not consistently implemented" imply design complexity, and "never completely specified" suggests unintentionality.

Data Model: Data integrity

> [Project A #18] approximately *340 records exist in the database twice … so much time had elapsed* in some cases the duplicate was endorsed.

In this example, "340 records exist in the database twice" implies maintenance complexity, and "so much time has elapsed" suggests unintentionality.

Regression Tests: Overly complex test

> [Chromium #367158] Currently, we have a lot of duplicate/boilerplate code in this test. *We should try to simplify* this test so that it's *easier to maintain* and read.

Here, "easier to maintain" implies maintenance complexity, and "we should try to simplify" suggests unintentionality.

Only two issues among the 51 hint at intentionality; however, we would not go so far as to call them strategic. The two "intentional" decision examples are shown below:

> [Project B #1393] Add "disabled" class to sensor tabs – it's a little bit hacky – disabled tab is still active. But it'll do *for this version*.

> [Connect #Gateway-1771] *Setting Guidance at the Adapter layer* is an idea that we documented and designed, however … we quickly realized some pitfalls and *decided not to go through with the implementation* such as: 1) There were many error cases which we would have to handle…

In the first example, "for this version" suggests that the developer is making an intentional decision to take on technical debt with hopes of refactoring later. In the second example, "Setting Guidance at the Adapter layer" implies a design limitation in the adapter, and "decided not to go through with the implementation" suggests an intentional decision to defer the rework. The issue description does not contain enough information to determine the impact of not making the change (such as increased accumulation in the form of complexity or maintainability).

Do groups of issues suggest technical debt?

When we asked project stakeholders to evaluate the results of our technical debt classification, we uncovered cases in which an issue by itself did not represent technical debt; however, when two or more issues were analyzed together, they suggested design limitations with accumulating side effects.

The Project A stakeholder confirmed that he would have also classified 9 of the 10 issues that we tagged as technical debt. In addition, he pointed out that several of the issues we found point to neglecting the data architecture, causing reliability, complexity, and data integrity issues. As shown in Figure 5, 72% of the technical debt issues in the Data Model group were found on Project A (8 of 11).

The Project B stakeholder positively confirmed 100% of the technical debt examples that we found. The project stakeholder revealed that lack of a robust and extensible UI framework had caused significant rework on the project. He said he would also include some other issues that we did not tag as technical debt due to their dependence on the UI framework. All three of the UI design issues shown in Figure 5 were from Project B.

The Connect stakeholder (one of the architecture evaluation leads) was able to positively confirm only 42% of the technical debt examples because he said the issue description lacked enough detail to make a determination. However, of the 42% positively confirmed technical debt examples (5 of 12 examples), he said that several issues were consistent with maintainability risks discovered during the architecture evaluation. For example, all four of the issues in the Deployment & Build group shown in Figure 5 were related to design concerns about the Connect build script maintainability.

Analysis of the technical debt issues that we identified allows us to conclude that

- issue data such as priority, duration open, and number of watchers does not imply accumulation, so it does not help identify technical debt historically.
- while our data set is small, we identify a starting set of recurring issues in technical debt. Post-refactoring alignment of unit test, build scripts, and versions and removal of dead code emerge as obvious technical debt-related concerns.
- technical debt is mostly the result of unintentional design choices; we were unable to find evidence of intentional technical debt being explicitly discussed in issue trackers.
- groups of issues that appear not to be technical debt when assessed individually can reveal underlying technical debt issues when assessed together.

6. IMPLICATIONS FOR PRACTICE AND RESEARCH

Issue trackers serve as an entry point for communicating technical debt since developers use them to manage task priorities. Anecdotal feedback from developers tells us that even when technical debt is included in the issue tracker, it may languish as it is not given priority or the symptoms are addressed but not the underlying issue. Our findings offer some practical improvements to bring better visibility to technical debt and ideas for future work.

6.1 Practice Improvements

Technical debt fosters dialogue between business and technical actors. Classifying technical debt issues allows developers to justify budgeting project resources for technical debt in a similar manner to allocate a discretionary budget for defects.

There are standards for providing bug reports with enough information so they may be reproduced and fixed [17][18]. These essential properties are encoded in predefined fields in issue trackers. These fields are necessary but not sufficient for describing technical debt. Recent research on technical debt has offered templates for reporting technical debt [34][24]. These contributions have similar goals to our work; however, templates recommend concepts that are at too high a level to overlap with daily routines and tasks of developers, such as estimated interest probability or principal and interest that are directly driven from the financial analogy.

Our analysis and examples demonstrate that technical debt becomes concrete when it relates to software units, as opposed to software process artifacts such as requirements or documentation. This refined scope leads to an understanding of technical debt as the collection of technical debt items associated with a system.

A technical debt item is a single element of technical debt connecting a set of development artifacts; with consequences for the quality, value, and cost of the system; and triggered by some causes related to process, management, context, and business goals. An item can be described using the properties in Table 5 based on the concepts for classifying technical debt (shown in bold), supplementing a typical issue report.

Introducing these properties can help developers understand trade-offs and the longer term consequences of technical debt when discussing an issue among themselves. It can also help make the

Table 5. Properties of technical debt items.

Name	Shorthand designation
Development artifact	**Executable** element of the system or the supporting work products: design, code, data, build scripts, test suites, etc.
Symptoms	Observable qualitative or measurable consequence (**type** of issue and analysis implications of **design**)
Consequences	Effect on value, quality, or cost of the system in the form of **accumulation**: additional costs due to reduced productivity, induced defects, or loss of quality incurred by software depending on an element of technical debt
Analysis	Degree to which the development approach (**design consideration**/**limitation**) meets stakeholder needs or expectations

case for additional resources when communicating to management. We suggest that developers use the properties shown here to write better descriptions and perhaps to increase the degree of automation possible in classifying them.

Table 6 shows an example of organizing the text according to these properties from a CONNECT issue.

The properties can also help parse the issues and identify what is ambiguous or missing. For example, without explicit information about debt accumulation, the issue cannot be properly classified nor the trade-offs understood. Developers may need this information to justify paying down the debt as an alternative to paying ongoing costs associated with addressing the symptoms.

6.2 Future Research

Our results suggest that by using automated text analysis and machine-learning techniques, technical debt issues can be more systematically discovered. To explore this, we ran a manual search against the 727 issues with the following words: duplicate, custom, workaround, inconsistent, hack, legacy, rewrite, cleanup, refactor, and refresh. We hypothesized that there would be a statistically significant difference between the percentage of issues that contain a key word AND are technical debt and the percentage of issues that contain a key word but are not technical debt. We found that 67% of the issues contained one of the key words and were tagged as carrying debt. Only 8% fall in the latter category. These findings suggest that automated word searches of key concepts related to technical debt may hold promise, but more experimentation is needed with large data sets.

Table 6. Example of a technical debt item.

Name	Connect #Gateway-1631: Empty Java package (dead code)
Development artifact	The re-architecture of the source code to support multiple NwHIN specifications has introduced a new Java packaging scheme.
Symptoms	Numerous empty Java package folders present across multiple projects.
Consequences	No impact to functionality; however, may lead to confusion for users implementing enhancements or modifications to the source code.
Analysis	New and existing classes have been moved into these new package folders; however, the previous package folders have been left in place with no class files.

Assessing accumulation was one of the biggest challenges we faced with systematically classifying technical debt issues in this study. Disagreement stemmed from two major sources. First, the language used by developers to describe accumulation is even less explicit than the design issue description. For example, developers made accumulation statements like "so much time has passed that now we have duplicate data," "this may lead to confusion for users," or "we should try to simplify so it is easier to maintain." The implicit, unstructured nature of accumulation language makes it difficult for reviewers to classify consistently, developers to assess impact, and researchers to study how to automate technical debt classification. Second, issues often included three types of accumulation information: (1) existing accumulation related to the current problem, (2) future recurring accumulation related to the current problem, and (3) accumulation related to the potential solution of the current problem, which we refer to a remediation. As discussed in Section 3.1, our response to confusion about this as we classified was to update the classification guidance to limit the scope of accumulation to type (1) for this study. Future research is needed to better define and model accumulation in terms of the costs associated with not fixing the problem and the added costs of fixing the problem at a later time.

Several examples, particularly in the more mature issue trackers (e.g., Chromium, Connect), included extensive developer discussion accompanied by significant code file check-in/check-out activity. A natural next step for this work is to analyze patterns found in the developer text discussion with references to technical debt and commit and change histories.

Our findings indicate several fruitful future research activities, and our plans include the following:

- Evaluate other techniques for mining unstructured data (e.g., pattern matching, island/lake parsers, information retrieval methods, and word categories) to locate technical debt in software repositories.
- Trace technical debt in the developer text discussion to code through the commit log to evaluate efficacy of self-reported debt in issue trackers.
- Model dimensions of accumulation in terms of cost to fix (paying down the principal), cost to not fix (paying interest), and the influence of time (current and future costs) to improve guidelines for describing technical debt.
- Build on the investment in the Chromium data set to conduct correlation studies with defects and software vulnerabilities to better understand the relationships among these kinds of software anomalies.

6.3 Threats to Validity

We identified the following threats to the validity of our study and took steps to minimize them.

Manual inspection: Manual inspection is crucial, especially in an exploratory study like ours that serves as input for creating key concepts. To counter the threat of making classification and interpretation mistakes, we included steps in our study to cross-check and discuss items. We also set a high inter-rater reliability threshold and had multiple researchers classify and code issues. In order to minimize researcher bias, we also had both developers of the system and experts external to the research team classify random samplings of the issues.

Study subjects: Software development management and issue tracking practices of the organizations whose data we used affect the quality of our results. The systems we selected may not have been representative. We aimed to minimize skewing of our results by selecting a variety of data sets from both open and closed systems, representative types of issue trackers, and established empirical analysis approaches.

Data quality and size: Technical debt represents only a small subset of all issues in a system, although its impact may be significant. Technical debt may not have been significantly represented in the data we selected, especially given the varying quality of the issue tracker data. We aimed to minimize this by randomizing the issues we selected, including both projects where we knew technical debt existed as well as others where we had no prior knowledge.

7. RELATED WORK

In empirical software engineering, it has become commonplace to mine data from change request and bug databases to detect where issues have occurred in the past and use that information for improved definitions, quality analysis, development management, and predictive models. Examples include but are not limited to manual and automated mining of issue trackers for misclassification [15], duplicates [5], and correlations of vulnerabilities and bugs [6]. Issue trackers also serve as historical data to help identify patterns to assist with predicting current or future events, such as risks [7]. To our knowledge, our study is the first one that extensively looks at issue databases through the lens of technical debt.

A key challenge in mining software repositories is data quality and missing data [28]. A number of studies look at the quality of reported data and ways to improve it, such as ensuring that missing links between bugs and bug-fix commits are included [4] and studying bug report quality [16][35]. These studies suggest that reports that contain key information get addressed sooner. Our results are consistent with these studies when it comes to reporting issues related to technical debt as well. To our knowledge, our study is the first one that provides key fields that need to be included in an issue report on technical debt.

The ability to accurately create an issue report communicating technical debt assumes a concrete understanding of technical debt. Numerous researchers have proposed a definition of technical debt, including McConnell [27], Li [23], Shull [30], and Kniberg [21]. To date these definitions stay at a conceptual level. Our study is the first that grounds an improved understanding of technical debt in actual software artifacts supported by extensive empirical data, contributing to the envisioned future for an improved data analysis and practice for managing technical debt [3] .

To understand implications of technical debt, systematic literature reviews have created categories and concept ontologies [19][1] or related debt to different stages in the development life cycle [2][23] [33]. Small-scale interview studies on understanding how developers talk about technical debt have focused on sources of technical debt [14][25][31]. These categories and classifications of technical debt rely on limited literature reviews and single-case studies. Our study is the first that demonstrates empirically that a significant amount of data is needed to talk about technical debt classification.

A number of studies have looked for relationships between software metrics and technical debt [13][26]. This work has applied existing code smells, coupling and cohesion, and dependency analysis to identifying areas of technical debt. Other work has looked at extracted examples of technical debt using keywords from developers' comments in code as self-admitted technical debt [29]. All of these stay at the level of code analysis, associating local code changes with technical debt. The work by Kazman et al. [20] relates architectural modularity violations to number of bugs to detect technical debt. This study is closest in its spirit to our findings that systematic issues hint at underlying technical debt.

8. CONCLUSIONS

Our study contributes to research on mining software repositories by looking at issue trackers from the perspective of early representations of technical debt. Our findings tell us the following:

- Technical debt concepts (e.g., taking on, accumulating, and paying back debt) have entered the vernacular of developers. But now they need a simple and formal approach to communicate the most crucial information. We offer the technical debt item and examples as a step toward that goal.
- Our data and analysis weakly support that issues where developers discuss certain classes of changes such as refactoring and cleanup are more likely to contain references to accumulation of technical debt.
- Technical debt conceptually is about conscious design trade-offs. However, the majority of technical debt that developers deal with is a consequence of unintentional design choices. Issue trackers carry information that can assist in uncovering the hidden technical debt.

We suggest that developers adopt a simple practice of concretely tagging and reporting technical debt and its consequences with accumulating side effects as they discover debt or take it on. This practice will help development teams start communicating about these issues more concretely and create a valuable resource for research. This contribution could help increase the sample size and quality of the data to make future research possible, since ambiguity led us to discard many issues in the existing data sets.

The past decade has seen significant progress in the mining software repositories community with substantial outcomes in robust automated analysis and correction tools as well as sound research approaches. Our exploratory study demonstrates that technical debt has become a ripe area in practice where mining software repositories research can be put to use to further improve our understanding, communication, and analysis of technical debt.

9. ACKNOWLEDGMENTS

Copyright 2016 ACM. 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.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003242

We thank Felix Bachmann, Phil Bianco, Philippe Kruchten, Tamara Marshall-Keim, Timothy Palko, and Hasan Yasar for their valuable feedback and expert input.

10. REFERENCES

[1] Alves, N. S. R., Ribeiro, L. F., Caires, V., Mendes, T. S., and Spínola, R. O. 2014. Towards an ontology of terms on technical debt. *ACM SIGSOFT* 40, 2 (Mar. 2015), 32-34.

[2] Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., and Avgeriou, P. 2015. The financial aspect of managing technical debt: A systematic literature review. *Inform. Software Tech.* 64 (Aug. 2015), 52-73.

[3] Avgeriou, P., Kruchten, P., Nord, R., Ozkaya, I., and Seaman, C. 2016. Reducing friction in software development. *IEEE Software* 33, 1 (Jan./Feb. 2016), 66-73.

[4] Bachmann, A., Bird, C., Rahman, F., Devanbu, P., and Bernstein, A. 2010. The missing links: bugs and bug-fix commits. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, NM, Nov. 7-11, 2010). FSE '18. ACM, New York, NY, 97-106.

[5] Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. 2008. Duplicate bug reports considered harmful … really? *In IEEE International Conference on Software Maintenance* (Beijing, China, Sep. 28-Oct. 4, 2008). ICSM '08. IEEE Press, Piscataway, NJ, 337-345.

[6] Camilo, F., Meneely, A., and Nagappan, M. 2015. Do bugs foreshadow vulnerabilities? A study of the Chromium Project. In *IEEE/ACM 12th Working Conference on Mining Software Repositories* (Florence, Italy, May 16-17, 2015). MSR '15. IEEE Press, Piscataway, NJ, 269-279.

[7] Choetkiertikul, M., Dam, H. K., Tran, T., and Ghose, A. 2015. Characterization and prediction of issue-related risks in software projects. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (Florence, Italy, May 16-17, 2015). MSR '15. IEEE Press, Piscataway, NJ, 280-291.

[8] Chromium Issues. https://code.google.com/p/chromium/issues/list

[9] CONNECT Health IT Exchange, U.S. Health and Human Services. 2009-2015 [open source project]. http://www.connectopensource.org/

[10] Connect Health IT Exchange Issue Tracker. https://connectopensource.atlassian.net/secure/Dashboard.jsp a

[11] Corbin, J. and Strauss, A. 2008. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, 3rd ed. Sage, Thousand Oaks, CA, 2008.

[12] Ernst, N., Bellomo, S., Ozkaya, I., Nord, R. L., and Gorton, I. 2015. Measure it? Manage it? Ignore it? Software practitioners and technical debt. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Bergamo, Italy, Aug. 30-Sep. 4, 2015). ESEC/FSE '15. ACM, New York, NY, 50-60.

[13] Fontana, F., Ferme, V., and Spinelli, S. 2012. Investigating the impact of code smells debt on quality code evaluation. In *Third International Workshop on Managing Technical Debt* (Zurich, Switzerland, June 5, 2012). IEEE Press, Piscataway, NJ, 15-22.

[14] Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., DaSilva, F., Santos, A., and Siebra, C. 2011. Tracking technical debt: An exploratory case study. In *Proceedings of the 27th International Conference on Software Maintenance* (Williamsburg, VA, Sep. 25-30, 2011). ICSM '11. IEEE Press, Piscataway, NJ, 528-531.

[15] Herzig, K., Just, S., and Zeller. A. 2013. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, May 18-26, 2013). ICSE '13. IEEE Press, Piscataway, NJ, 392-401.

[16] Hooimeijer, P. and Weimer, W. 2007. Modeling bug report quality. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, GA, Nov. 5-9, 2007) ASE '07. ACM, New York, NY, 34-43.

[17] IEEE Std 1044-2009: IEEE Standard Categorization for Software Anomalies. 2009. IEEE Computer Society, Washington, DC.

[18] ISO/IEC 14764:2006(E): Software Engineering – Software Life Cycle Processes – Maintenance. 2006. ISO/IEC, Geneva Switzerland.

[19] Izurieta, C., Vetro, A., Zazworka, N., Cai, Y., Seaman, C., and Shull, F. 2012. Organizing the technical debt landscape. In *Third International Workshop on Managing Technical Debt* (Zurich, Switzerland, June 5, 2012). IEEE Press, Piscataway, NJ, 23-26.

[20] Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., Fedak, V., and Shapochka, A. 2015. A case study in locating the architectural roots of technical debt. In *Proceedings of the 37th IEEE International Conference on Software Engineering* (Florence, Italy, May 16-24). ICSE '15. IEEE Press, Piscataway, NJ, 179-188.

[21] Kniberg, H. 2013. Good and bad technical debt (and how TDD helps. Oct. 2013. http://blog.crisp.se/2013/10/11/henrikkniberg/good-and-bad-technical-debt.

[22] Kruchten, P., Nord, R. L., and Ozkaya, I. 2012. Technical debt: From metaphor to theory and practice. *IEEE Softw.* Spec. Issue Tech. Debt 29, 6 (Nov.-Dec. 2012), 18-21.

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

[24] Li, Z., Liang, P., and Avgeriou, P. 2014. Architectural debt management. In *Economics-Driven Software Architecture*, I. Mistrik, R. Bahsoon, Y. Zhang, K. Sullivan, and R. Kazman, Eds. Elsevier, San Diego, CA, 2014, 183-204.

[25] Lim, E., Taksande, N., and Seaman, C. 2012. A balancing act: What software practitioners have to say about technical debt. *IEEE Software* 29, 6 (Nov./Dec. 2012), 22-27.

[26] Marinescu, R. 2012. Assessing technical debt by identifying design AWS in software systems. *IBM Journal of Research and Development* 56, 5 (Sep./Oct. 2012), 9:1-9:13.

[27] McConnell, S. 2007. Technical debt. Construx, Nov. 1, 2007. http://www.construx. http://www.construx.com/10x_ Software_Development/Technical_Debt/

[28] Mockus, A. 2008. Missing data in software engineering. In *Guide to Advanced Empirical Software Engineering*, F.

Shull, J. Singer, and D. Sjoberg, Eds. Springer Verlag, New York, NY, 185-200.

[29] Potdar, A. and Shihab, E. 2014. An exploratory study on self-admitted technical debt. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution* (Victoria, BC, Canada, Sep. 29-Oct. 3, 2014). ICSME '14. IEEE Press, Piscataway, NJ, 91-100.

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

[31] Spínola, R. O., Zazworka, N., Vetro, A., Seaman, C., and Shull, F. 2012. Investigating technical debt folklore: shedding some light on technical debt opinion. *Third International Workshop on Managing Technical Debt* (Zurich, Switzerland, June 5, 2012). IEEE Press, Piscataway, NJ, 1-7.

[32] Technical Debt Classification Approach Document and Technical Debt Issue Examples. 2016 Sample Data Set http://sei.cmu.edu/architecture/research/arch_tech_debt/got-technical-debt.cfm

[33] Tom, E., Aurum, A., and Vidgen, R. T. 2013. An exploration of technical debt. *J. Syst. Softw.* 86, 6 (June 2013), 1498-1516.

[34] Zazvorka, N., Spínola, R., Vetro', A., Shull F., and Seaman C. 2013. A case study on effectively identifying technical debt. In *17th International Conference on Evaluation and Assessment in Software Engineering* (Porto de Galinhas, Brazil, April 14-16, 2013). EASE '13. ACM, New York, NY, 42-47.

[35] Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schröter, A., and Weiss, C. 2010. What makes a good bug report? *IEEE Trans. Software Eng.* 36, 5 (Sep. 2010), 618-643.