Identification and Remediation of Self-Admitted Technical Debt in Issue Trackers

Yikun Li, Mohamed Soliman, Paris Avgeriou
Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence
University of Groningen
Groningen, The Netherlands
{yikun.li, m.a.m.soliman, p.avgeriou}@rug.nl

***Abstract*—Technical debt refers to taking shortcuts to achieve short-term goals, which might negatively influence software maintenance in the long-term. There is increasing attention on technical debt that is admitted by developers in source code comments (termed as self-admitted technical debt or SATD). But SATD in issue trackers is relatively unexplored. We performed a case study, where we manually examined 500 issues from two open source projects (i.e. Hadoop and Camel), which contained 152 SATD items. We found that: 1) eight types of technical debt are identified in issues, namely architecture, build, code, defect, design, documentation, requirement, and test debt; 2) developers identify technical debt in issues in three different points in time, and a small part is identified by its creators; 3) the majority of technical debt is paid off, 4) mostly by those who identified it or created it; 5) the median time and average time to repay technical debt are 872.3 and 25.0 hours respectively.**

***Index Terms*—mining software repositories, self-admitted technical debt, technical debt introduction, technical debt repayment, issue tracking system**

I. Introduction

Technical debt (TD) refers to taking shortcuts, either deliberately or inadvertently, to achieve short-term goals, which might negatively influence the maintenance and evolution of software in the long term [1]. Technical debt can be incurred in activities throughout the whole development life cycle, from requirements, to design, implementation, testing, etc. There have been several approaches supporting the identification of technical debt in almost all of these activities [2]. For example, there are approaches detecting code debt by analyzing source code [2], and test debt by analyzing test reports [2].

A part of technical debt is declared as such by the developers themselves; for example when developers state in source code comments, that something is not right and should be fixed. This has been termed *"Self-Admitted Technical Debt" (SATD)* [3]. SATD is often complementary to other types of technical debt items, as it provides information that cannot be uncovered through other means of technical debt identification. For example deciding to use a sub-optimal library is likely to be captured in a source code comment but it cannot be detected from source code. Maldonado and Shihab [3] detected five types of SATD (i.e. requirement, code, design, defect, and documentation debt) from source code comments.

This work was supported by ITEA3 and RVO under grant agreement No. 17038 VISDOM (https://visdom-project.github.io/website).

While current work on SATD has focused on source code comments, there are other potentially rich sources of information containing SATD. In this paper we focus on SATD in issue trackers, as developers often discuss about technical debt when working on issues. There has been some research work exploring technical debt in issue tracking systems [4], [5], showing the possibility of detecting TD through issue trackers, and analyzing the characteristics of technical debt issues, such as opening time, and number of watchers. However, SATD in issue tracking systems is still relatively unexplored.

The main **goal** of this paper is to *analyze the types of SATD in issue tracking systems, and to determine how software engineers identify and resolve them*. To achieve our goal, we conducted a case study where we performed a qualitative analysis on a sample of 500 issues. Specifically, we identified and analyzed sentences in issues that refer to SATD. Our findings indicate that: 1) eight types of technical debt are found in issues, namely architecture, build, code, defect, design, documentation, requirement, and test debt; 2) there are three distinct cases of identifying technical debt in issue trackers, while only a small part (13.1%) of technical debt is identified by its creators; 3) the majority of technical debt is paid off, mostly by those who identified or created it (47.7% and 44.0% respectively); 4) the median time and average time spent on technical debt repayment are 25.0 and 872.3 hours.

Our findings provide a number of implications to practitioners and researchers, including: 1) using issue trackers as complementary sources to source code comments for debt detection; 2) developing approaches to detect technical debt, depending on the time that the debt is identified; 3) reporting urgent technical debt in issue trackers, rather than in source code comments, for quicker repayment.

The remainder of this paper is organized as follows. In Section II, related work is discussed. Section III presents a typical issue life cycle, accompanied with an example. The case study design is then elaborated in Section IV, while the results are presented and discussed in Section V and Section VI respectively. Finally, threats to validity are evaluated in Section VII and conclusions are drawn in Section VIII.

II. Related Work

In this study, we investigate technical debt in issue trackers, which is a type of SATD. Thus, we organize the related work

into two parts: work related to SATD in general and work related to technical debt in issue trackers.

Self-admitted Technical debt: Potdar and Shihab [6] studied self-admitted technical debt in source code comments within four open source projects. They found that a range of 2.4% to 31.0% of source files contain SATD and 26.3% to 63.5% of debt is eventually removed. In a follow-up study, Maldonado and Shihab [3] studied five open source projects and discovered the following five types of SATD: design, defect, documentation, requirement, and test debt.

There has also been work related to paying back SATD. Maldonado *et al.* [7] analyzed five Apache projects to study the removal of SATD. They found that most of SATD is removed by the same person that introduced it, and on median, it takes 18 to 172 days to remove SATD comments. Zampetti *et al.* [8] also analyzed the removal of SATD in five Java open source projects. The findings showed that 20% to 50% of SATD is removed unintentionally, and 8% of debt removal is recorded in commit messages. Our work differs from the work described above, as we look into SATD within issue trackers, instead of source code comments.

Technical debt in issue trackers: To the best of our knowledge, only two studies have focused on the detection and comprehension of technical debt in issue trackers. The first, by Bellomo *et al.* [4] presents a classification method for technical debt issues. They manually examined 1,264 issues in four issue trackers from two government projects and two open source industry projects. From this set, they classified 109 issues as technical debt issues and derived generic characteristics for these issues. The second study, by Dai and Kruchten [5] analyzed issues from a commercial software issue tracker by reading issue summaries and descriptions. From 8,149 analyzed issues, they classified 331 as TD issues, and categorized them into six types - defect, requirement, design, code, UI, and architecture debt. Subsequently, by using machine learning techniques, they trained a classifier with the analyzed issues to automatically classify TD issues.

Our study also classifies issues into types of technical debt (RQ1). But it differs, as it also focuses on how technical debt items are identified (RQ2), and how technical debt items are repaid by developers (RQ3). Moreover, we analyze issues on the sentence level by reading each sentence in the issue summary, description, and comments. If a sentence or a group of sentences indicates technical debt, we tag it as a technical debt statement. This is different from the aforementioned related studies [4], [5] as they both classified whole issues as technical debt issues or non-technical debt issues. Treating a whole issue as a single type of technical debt may be inaccurate, because software engineers might discuss several types of technical debt in the same issue. For example, in issue HADOOP-6730[1], software engineers discuss both code debt and test debt.

III. Background - Issue Life Cycle

In general, an issue tracker is a system for issue management. A managed issue is not only limited to defects but also new features or refactoring. An issue has its own life cycle, from the time it is created until the time it is resolved. The typical steps of this life cycle and an example of each step are shown in Table I.

TABLE I
AN EXAMPLE OF AN ISSUE LIFE CYCLE.

No.	Step	Description	Example (Hadoop-11074[2])
1	Create Issue	Usually, software developers create an issue when they find bugs or have new requirements. They first create an issue, which is assigned a unique issue key and describe that issue in detail.	*"Now that hadoop-aws has been created, we should actually move the relevant code into that module, similar to what was done with hadoop-openstack, etc."* (unique key is Hadoop-11074)
2	Discuss and Create Patch	At a later stage, developers start working on it: they comment inside the issue analyzing the problem and sharing their ideas about the solution, and then create a patch to address the issue.	*"HADOOP-11074.patch is attached. This patch does the following: Move the s3 and s3native FS connector code from hadoop- common to hadoop-aws..."*
3	Code Review	The proposed patch is reviewed by other developers, and feedback is given. If no problem is found in the patch, they proceed to step No.6, otherwise to the next step.	*"Can you add an @Ignore on the tests which are failing, so that we can have a green upstream build? +1 once that's addressed."*
4	Update Patch	According to the code review feedback, developers refine the patch and submit it again for another round of code review.	*"HADOOP-11074.patch.2 is attached. Change the original patch to... This should get Jenkins passing."*
5	Code Review	The code is reviewed once more. If it passes, developers proceed to the next step; otherwise, they go back to step No.4.	*"+1, will commit in an hour or two if there are no more comments."*
6	Final Code Commit	The approved patch is committed to the repository with the issue key included in the commit message, and then the issue status is changed to *Resolved*.	*"Patch is committed. Commit message: HADOOP-11074. Move s3-related FS connector code to hadoop-aws."*

IV. Case Study Design

The goal of this study, formulated according to the Goal-Question-Metric [9] template is to "***analyze*** *issues in issue tracking systems* ***for the purpose of*** *characterizing the technical debt within the issues* ***with respect to*** *the types, the introduction, and the repayment of technical debt* ***from the point of view of*** *software developers* ***in the context of*** *open source software*". This goal is refined into three research questions (RQs):

- **(RQ1)** *What types of technical debt are reported in issues?* Having knowledge of the types of technical debt could help us understand the strengths and limitations of detecting technical debt in issue trackers. For example, we may find that a specific type of technical debt is only detected in issues and not in other sources, or that it is mostly detected in issues. That can help in proposing

[1]https://jira.apache.org/jira/browse/HADOOP-6730
[2]https://jira.apache.org/jira/browse/HADOOP-11074

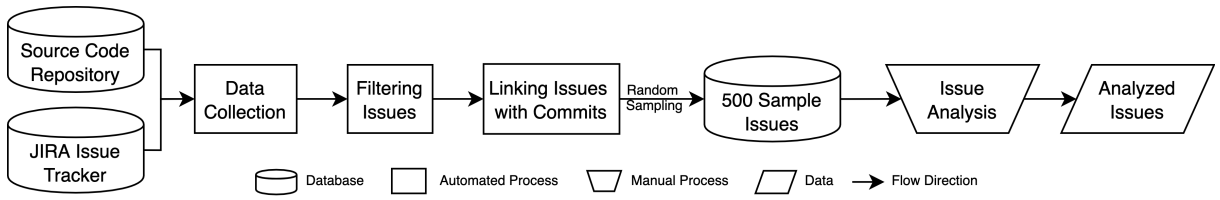

Fig. 1. The framework of our approach.

approaches for detecting technical debt that combine different sources. Although Dai and Kructhen [5] also studied types of debt in issues, they only analyzed the issue summary and description. In contrast, we analyze entire issues (including the comments) at the level of sentences.

- **(RQ2)** *When do software developers identify technical debt in issues?* This RQ aims at understanding the point in time that debt is identified in issue trackers. For example, technical debt can be incurred when working on an issue, or it can exist beforehand and the issue is created to address it. This can help researchers to tune their TD detection approaches depending on when it is identified. For example, if the technical debt is added to a patch and eventually the patch is rejected (not committed), the debt is not added to the system. In this case, an approach may falsely detect this debt item in a code review statement regarding that (rejected) patch.
- **(RQ3)** *How do software engineers resolve technical debt in issues?* This is further refined into 3 sub-questions:
  - **(RQ3.1)** *How much technical debt is resolved?* Quantifying how much technical debt is paid off, helps us understand developers' attitudes towards technical debt and of course the magnitude of the problem. For instance, if most of the debt is discussed and resolved, it would imply that developers are aware of the harmfulness of technical debt and take action resolving it. It would also imply that technical debt in issues does not pose a critical threat.
  - **(RQ3.2)** *Who resolves technical debt?* Technical debt can be resolved by those who created it, those who discovered it, or by others. This aids in understanding the practices of developers, e.g. if those that incur debt take the responsibility to resolve it. It can also be used to assist with debt repayment; for example if the debt creator did not resolve it, another developer may need more documentation to understand the problem well enough in order to solve it.
  - **(RQ3.3)** *How long does it take to resolve technical debt?* Knowing how long it normally takes to repay technical debt after discovering it, is helpful for technical debt management. Technical debt that is long-lived causes extra maintenance effort and should thus be prioritized for remediation.

Fig. 1 shows the approach we follow to answer the research questions. The four individual processes (automated and manual) are explained in the following sub-sections.

A. Data collection

To answer the research questions, we looked into Apache Java projects since they are of high quality and supported by mature communities. To select Apache projects pertinent to our study goal, we set the following criteria:

1) Both the issue tracking project and the source code repository are publicly available and well-maintained.
2) They have at least 1,000,000 source lines of code (SLOC) and 10,000 issues in the issue tracker. This is to ensure sufficient complexity.
3) Source code commits involve their associated issue keys within their comments. This is important to support linking commits (in the source code repository) with issues (in the issue tracker). This is further motivated in Section IV-C.
4) They are commonly used in other SATD studies (e.g. [7]). This allows us to compare the results between our study and other SATD studies.

Based on these criteria, we selected Hadoop[3] and Camel[4]. Both projects were studied for SATD [7], were developed in Java, used Git as a source code repository and JIRA[5] as an issue tracker. We analyzed the latest released versions on Jan 16, 2020. Table II shows some details for the two projects. The number of Java files and SLOC are calculated using the LOC tool[6]. The number of contributors is obtained from GitHub. We used the JIRA Python package to extract all Hadoop and Camel issues from the online server and stored them in a local database; then we counted the number of issues.

TABLE II
DETAILS OF CHOSEN PROJECTS.

Project	# Java files	SLOC	# Contributors	# Issues	# Filtered issues
Hadoop	10,918	1,700,501	259	16,808	6,685
Camel	17,585	1,196,790	583	14,411	12,259

B. Filtering issues

To ensure that we study issues with a complete life cycle (as shown in Table I), we applied two filtering criteria:

1) **Issue status:** Since we are aiming at studying technical debt items that were resolved, we focus on issues that

[3]https://hadoop.apache.org
[4]https://camle.apache.org
[5]https://jira.apache.org
[6]https://github.com/cgag/loc

TABLE III
DEFINITIONS OF INDICATORS OF DIFFERENT TYPES OF TECHNICAL DEBT IN ISSUE TRACKERS.

Type	Indicator	Reused	Definition
Architecture debt	Violation of modularity	•	Because shortcuts were taken, multiple modules became inter-dependent, while they should be independent.
	Using obsolete technology	○	Architecturally-significant technology has become obsolete.
Build debt	Under- or over-declared dependencies	•	Under-declared dependencies: dependencies in upstream libraries are not declared and rely on dependencies in lower level libraries. Over-declared dependencies: unneeded dependencies are declared.
	Poor deployment practice	○	The quality of deployment is low that compile flags or build targets are not well organized.
Code debt	Complex code	○	Code has accidental complexity and requires extra refactoring action to reduce this complexity.
	Dead code	○	Code is no longer used and needs to be removed.
	Duplicated code	•	Code that occurs more than once instead of as a single reusable function.
	Low-quality code	○	Code quality is low, for example because it is unreadable, inconsistent, or violating coding conventions.
	Multi-thread correctness	•	Thread-safe code is not correct and may potentially result in synchronization problems or efficiency problems.
	Slow algorithm	•	A non-optimal algorithm is utilized that runs slowly.
Defect debt	Uncorrected known defects	•	Defects are found by developers but ignored or deferred to be fixed.
Design debt	Non-optimal decisions	○	Non-optimal design decisions are adopted.
Documentation debt	Outdated documentation	•	A function or class is added, removed, or modified in the system, but the documentation has not been updated to reflect the change.
	Low-quality documentation	○	The documentation has been updated reflecting the changes in the system, but quality of updated documentation is low.
Requirement debt	Requirements partially implemented	○	Requirements are implemented, but some are not fully implemented.
	Non-functional requirements not fully satisfied	○	Non-functional requirements (e.g. availability, capacity, concurrency, extensibility), as described by scenarios, are not fully satisfied.
Test debt	Expensive tests	○	Tests are expensive, resulting in slowing down testing activities. Extra refactoring actions are needed to simplify tests.
	Lack of tests	○	A function is added, but no tests are added to cover the new function.
	Low coverage	•	Only part of the source code is executed during testing.

are done or closed. Thus, we removed all issues with status *Open* or *Pending Closed*.

2) **Availability of issue key in commits:** Although some issues have their status set to *Resolved* and developers commented that the patches are successfully committed to the repositories, we cannot find the related commits in Git. This is mostly because developers did not include the issue key in the corresponding commit messages. We also exclude these issues, since we need the commit information to be able to answer RQ3 on debt repayment.

The final number of issues after filtering is listed in the rightmost column of Table II.

C. Linking issues with commits

In order to determine how software engineers actually resolve technical debt (i.e. answering RQ3), we have to capture the code commits associated with an issue. This information is needed to determine the software developers responsible for repaying technical debt (RQ3.2) and the time for this repayment (RQ3.3).

Since in the previous step, we ensured that the commit messages contain the related issue keys, we use those keys to link issues with commits. In practice, we first output the Git commit log, and match the issue key by applying a regular expression to the commit log. Then all matched commits (including commit date, commit message, and commit author) are inserted into the issue holding the issue key ordered by time, and then the issue with commit information is stored in a local database.

D. Issue manual analysis

The filtering step resulted in 18944 issues that fulfill our criteria (see Section IV-B): 6685 for Hadoop and 12259 for Camel. Since manually analyzing issues is extremely time-consuming, we are only able to analyze a subset. From this set, we randomly selected a sample of 500 issues for analysis: 250 issues from each project (i.e. Hadoop and Camel). The size of our sample is in line with similar studies, e.g. Zaman *et al.* analyzed 400 issues to study performance bugs [10]. To analyze issues for technical debt, we followed the instructions for qualitative analysis proposed by Runeson *et al.* [11]. We used a professional qualitative content analysis tool (ATLAS.ti[7]) to annotate relevant sentences within the sample issues.

To answer RQ1, we performed a classification using an existing framework from Alves *et al.* [12]. This framework

[7]https://atlasti.com

provides basic types of technical debt, with high-level definitions and a list of indicators per type. Using these types, we annotated sentences within issues, referring to existing debt or resolving debt. We read each sentence in issue summary, description, and comments. If a sentence or a group of sentences indicated a certain type of technical debt, we tagged it with that type and relevant indicators.

The issues were independently annotated by the first and second author. The differences between the two authors supported refining the types and indicators of technical debt from the original framework of Alves *et al.* [12]. For example, we added the indicator *Requirements Partially Implemented* to the requirement debt type. The refined classification framework that resulted from this step is presented in Table III. The *Reused* column refers to whether the indicators are reused directly from the study of Alves *et al.* ("•" symbol) or they were created inductively during the qualitative analysis ("○" symbol). The original framework of Alves *et al.*, can be found in the replication package[8]. The classification resulted in 152 annotated statements with different technical debt types and indicators, which are also available in the replication package.

To mitigate the risk of bias, we evaluated the level of agreement between the classifications of the two authors using Cohen's kappa coefficient [13]; this is commonly used to measure inter-rater reliability. The calculated level of agreement between the two authors is 0.757 based on a sample consisting of 15% of all technical debt statements, which is considered excellent according to the work of Fleiss *et al.* [13].

Next, we revisited all identified technical debt to obtain information to answer RQ2 and RQ3. More specifically, for RQ2, we annotated text with information regarding the identification of technical debt items within the issue life cycle. Regarding RQ3, for each technical debt item, we read the related issue comments and the corresponding commit messages (see Section IV-C) to identify information on debt remediation. If indeed there was such information, we noted it down, as well as the person who resolved the item and the time between reporting and resolving it.

V. Results

A. (RQ1) What types of technical debt are reported?

We found eight types of technical debt in issues: architecture, build, code, defect, design, documentation, requirement, and test debt. For each type we found one or more indicators. In the following paragraphs, we report on the associated indicators for each type, also providing a quote from actual issues to exemplify each indicator.

Architecture debt: problems that are architecturally significant, i.e. they are hard to change. Most of the debt in this type relates to the indicator *Violation of Modularity*.

> *"It would be good if these were moved into their own module..."* - [Camel-4543]

Some architecture debt is caused by *Using Obsolete Technology*.

> *"The camel-atom component is using an ancient incubator version of abdera which will make it hard to work with camel-cxf."* - [Camel-4132]

Build debt: issues that make building (i.e. source code compilation to artifacts) harder or more time-consuming. Most of the identified build debt is caused by *Over- or Under-Declared Dependencies*.

> *"Avoid the redundant direct dependency on log4j by the components."* - [Camel-4331]
>
> *"Compiling for Fedora revels a missing declaration for javax.annotation.Nullable. This is the result of a missing explicit dependency on..."* - [Hadoop-10067]

The rest of build debt is caused by *Poor Deployment Practice*.

> *"Rationalize the way architecture-specific subcomponents are built with ant in branch-1. This is a matter of maintainability and understandability, and therefore robustness under future changes in build.xml."* - [Hadoop-8364]

Code debt: issues in source code, which negatively influence the maintenance of software. Most of the code debt is caused by *Low-Quality Code*.

> *"This will lead to very unmaintainable code. We absolutely do not want to have nested retries for different contexts."* - [Hadoop-3198]

A few code debt items result from *Slow Algorithm*.

> *"#query() does O(N) calls LinkedList#get() in a loop, rather than using an iterator. This makes query O(N^2), rather than O(N)."* - [Hadoop-8866]

Multi-Thread Correctness is another factor causing code debt.

> *"EnsureInitialized() forced many frequently called methods to unconditionally acquire the class lock."* - [Hadoop-9748]

The rest of the code debt is caused by *Dead Code*, *Duplicated Code*, and *Complex Code*.

> *"As we don't use the CxfSoap component any more, it's time to clean it up."* - [Camel-2535]
>
> *"I am concerned about the code duplication this brings."* - [Hadoop-6381]
>
> *"...can be simplified to the following so there aren't so many return statements to track."* - [Hadoop-10169]

Defect debt: known defects that are deferred to be fixed. All defect debt items are caused by *Uncorrected Known Defects*.

> *"This works in 2.12.x onwards. Hunting this down on 2.11.x is low priority. End users is encourage to upgrade if they really need this."* - [Camel-6735]

Design debt: shortcuts or non-optimal decisions taken in detailed design. All design debt results from *Non-Optimal Decisions*.

[8]http://www.cs.rug.nl/search/uploads/Resources/li_soliman_avgeriou_seaa2020.zip

"Instead of passing a long[] you should pass a struct that implements Writable." - [Hadoop-481]

'Extending the Trash API might be ok in the short term but does not sound too appealing from a long-term perspective." - [Hadoop-2815]

Documentation debt: when the software is modified, the documentation is not updated to reflect the changes or the quality of updated documentation is low. Most of this type of debt is caused by *Outdated Documentation*.

"The maven reports is just getting to old and intermixed with 1.x and trunk releases." - [Camel-1846]

The second indicator is *Low-Quality Documentation*.

"I agree to improve documentation to make it clear that..." - [Hadoop-12672]

Requirement debt: when the requirements specification is not in line with the actual implementation. Some requirement debt is caused by *Requirements Partially Implemented*.

"The only feature which we don't support is correlated message groups. That requires a bit more work and also may complicated..." - [Camel-1669]

Another common cause concerns *Non-Functional Requirements Not Being Fully Satisfied*. In the example below, concurrency is not fully satisfied.

"Definition requires the implementations for its interfaces should be thread-safe. HarFsInputStream doesn't implement these interfaces with tread-safe, this JIRA is to fix this." - [Camel-5587]

Test debt: shortcuts or non-optimal decisions taken in testing that negatively affect maintainability. Most test debt is caused by *Lack of Tests*.

"There are no XQuery specific tests." - [Camel-201]

The other major cause of test debt is *Low Coverage*.

"Some of the test code doesn't check for correct error codes to correspond with the wrapped exception type." - [Hadoop-11103]

Finally, some test debt results from *Expensive Tests*.

"I see recent hadoop-hdfs test runs have been taking 2.5 hours. This one (new patch) was 45 minutes." - [Hadoop-11670]

Table IV presents an overview of technical debt types and indicators in the examined issues. We observe that code, documentation, and test debt are the three most common types (with 38.8%, 21.7%, and 18.4% respectively). Furthermore, the three most common indicators are *Low-quality Code*, *Lack of Tests*, and *Outdated Documentation*.

Finally, since we annotated technical debt on the sentence level (instead of the issue level), an issue may contain more than one types of technical debt. Table V presents how many issues contain zero, one or more types of technical debt in issues. As we can see, 24 out of 117 issues (20%) that contain technical debt, contain more than one type. This validates our choice to analyze issues at the level of sentences; if we had performed the analysis at the level of issues, we would have missed the additional technical debt types per issue.

[9]The symbol # refers to the number of instances.

TABLE IV
TYPES AND INDICATORS OF TECHNICAL DEBT.

Type	Indicator	#[9]	#[9]	%
Architecture debt	Violation of modularity	8	10	6.6
	Using obsolete technology	2		
Build debt	Over- or under-declared dependencies	5	6	3.9
	Poor deployment practice	1		
Code debt	Complex code	2	59	38.8
	Dead code	12		
	Duplicated code	6		
	Low-quality code	36		
	Multi-thread correctness	1		
	Slow algorithm	2		
Defect debt	Uncorrected known defects	4	4	2.6
Design debt	Non-optimal decisions	8	8	5.3
Documentation debt	Low-quality documentation	16	33	21.7
	Outdated documentation	17		
Requirement debt	Requirements partially implemented	3	4	2.6
	Non-functional requirements not being fully satisfied	1		
Test debt	Expensive tests	1	28	18.4
	Lack of tests	20		
	Low coverage	7		

TABLE V
NUMBERS OF TYPES OF TECHNICAL DEBT IN ISSUES.

Issue description	# Issues	% Issues
Does not contain technical debt	383	76.6
Contains one type of technical debt	93	18.6
Contains two types of technical debt	21	4.2
Contains three types of technical debt	2	0.4
Contains four types of technical debt	1	0.2

Eight types of technical debt are found in issue trackers: architecture, build, code, defect, design, documentation, requirement, and test debt. The three most common types are code, documentation, and test debt (i.e. 38.8%, 21.7%, and 18.4%). About one fifth of the issues that contain technical debt, contain more than one type.

B. (RQ2) When do software engineers identify technical debt?

We observed three distinct cases of technical debt being identified in issue trackers:

1) *Identifying technical debt before creating an issue (i.e. debt is the reason for creating the issue):* When developers spot an existing technical debt item in the system, they report it in an issue tracker to be resolved. For instance, a developer found low-quality code, which complicates debugging; thus, he/she created a new issue:

"If the user doesn't setup the right camel context for the context component. The exception we got is misleading, we need to throw more meaningful exception for it." - [Camel-5714]

2) *Identifying technical debt during code review:* As explained in Section III, software engineers perform code reviews by creating and reviewing code patches in issue trackers. When a code reviewer identifies technical debt items in a code patch, he/she discusses it with other developers to determine, if the identified technical debt should be resolved or committed to the system. For example, during a code review, a developer found that a shortcut was taken. Thus, he/she commented on a patch:

   "The patch looks good to me... It would be better if we can add an upper limit for the size of the GSet." - [Hadoop-9763]

3) *Identifying technical debt after a patch is committed:* Technical debt can exist in a patch but go undetected through the code review; after the patch is committed, a developer may notice the debt in the commit and report it. For instance, after a command patch is committed to the repository, a developer noticed that documentation is not updated accordingly:

   "We need to update the documentation with the new command." - [Camel-8101]

TABLE VI
TECHNICAL DEBT IDENTIFICATION CASES.

Project	# Identified	Case 1		Case 2		Case 3	
		#[9]	%	#[9]	%	#[9]	%
Hadoop	101	41	40.6	57	56.4	3	3.0
Camel	51	27	52.9	13	25.5	11	21.6
Total	152	68	44.7	70	46.1	14	9.2

To gain a better understanding of how technical debt is identified, Table VI presents the count of technical debt items for the three aforementioned cases. Clearly, the first and second cases represent the majority (44.7% and 46.1% respectively) in these projects. Compared with Camel, there is 30.9% more debt introduced in Hadoop with the second case and 18.6% less debt introduced with the third case. This means that more technical debt is identified during code reviews (on patches) than after the patch is committed in the system in Hadoop compared with Camel.

TABLE VII
TECHNICAL DEBT REPORTERS.

Project	Reported by creators		Reported by others	
	#[9]	%	#[9]	%
Hadoop	4	6.7	56	93.3
Camel	7	29.2	17	70.8
Total	11	13.1	73	86.9

Moreover, we also investigate who reported the debt: the developers who created it in the first place or those who discovered it. Since technical debt identified in the first case already exists in the system, information on who created it is not contained in issue trackers; thus, such information is obtained only for technical debt identified in the second and third cases. Table VII presents an overview on who reported the technical debt. We find that on average most of the debt is reported by other developers (i.e. 86.9%), and a small part is self-reported (reported by those that created it). Camel has a higher percentage of self-reported debt than Hadoop, but the vast majority of its debt is still reported by others (i.e. 70.8% versus 29.2%). This may mean that most developers create technical debt unintentionally.

> *There are three cases of identifying technical debt in issue trackers: discovering existing debt and creating an issue for it, identifying debt in a patch during code review, or after the patch is committed in the system. Most of the technical debt is identified in the first and second cases. A small part of the debt is reported by its creators, while most is reported by other developers.*

C. (RQ3) How do software engineers resolve technical debt?

1) ***(RQ3.1) How much technical debt is paid off?*** Table VIII presents the amounts and percentages of technical debt items that are identified and resolved. We can see that most of the identified technical debt is actually resolved in both Hadoop and Camel (i.e. 71.3% and 72.5%, respectively). This indicates that, when technical debt is reported in issue trackers, it will likely be resolved. In other words, most software developers are conscious of the importance of paying off technical debt items.

TABLE VIII
AMOUNT OF TECHNICAL DEBT THAT WAS REPAID.

Project	# Identified	# Repaid	% Repaid	% Remaining
Hadoop	101	72	71.3	28.7
Camel	51	37	72.5	27.5
Total	152	109	71.7	28.3

2) ***(RQ3.2) Who repays technical debt?*** As shown in Table IX, we distinguish between developers who create technical debt, those who identify it and other developers who participate in resolving it. We can see that most of the technical debt is repaid by those who identified it (i.e. 47.7%), and those who created it (i.e. 44.0%); while only 8.3% debt is resolved by other developers. This shows that developers take the responsibility to pay off most of the technical debt they identified or created themselves.

TABLE IX
WHO REPAID TECHNICAL DEBT.

Project	# Repaid	# Repaid by					
		Creators		Identifiers		Others	
		#[9]	%	#[9]	%	#[9]	%
Hadoop	72	36	50.0	33	45.8	3	4.2
Camel	37	12	32.4	19	51.4	6	16.2
Total	109	48	44.0	52	47.7	9	8.3

*3) **(RQ3.3) How long does it take to fix technical debt?*** Fig. 2 shows the mean times, the median times, and the time distributions of technical debt repayment for the two projects. With a visual inspection, we see that the time spent to fix technical debt in Hadoop and Camel varies. We also observe that after the technical debt is reported (point zero in the y axis), most of the fixes happened in a short time compared to the average (67.0% of the debt is repaid in the first 100 hours).

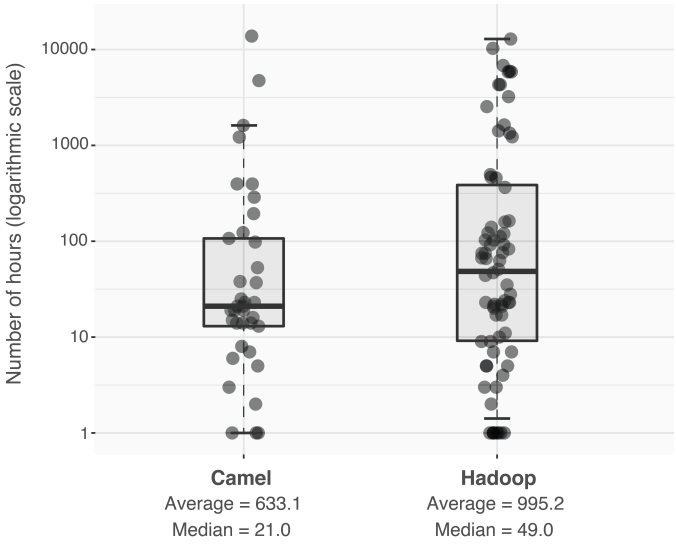

Fig. 2. The time distribution of technical debt repayment in issue trackers.

Furthermore, we compare the time spent on resolving technical debt by different developers (Creators, Identifiers, and Others as discussed in Section V-C2). More specifically, we compare repayment time distributions between pairs of developers (e.g. between creators and identifiers) using the Mann-Whitney test [14] and Cliff's delta [15] to determine the significance level and the effect size of the differences. The result is demonstrated in Table X. There are notable differences between Hadoop and Camel. In Hadoop, the repayment time of identifiers and others is longer than creators with statistical significance (p-values are 0.031 and 0.028 respectively). Moreover, the time difference between identifiers and others is at the margin of statistical significance (p-value is 0.080). According to the effect size, we observe that the difference between creators and identifiers is small, while the difference between identifiers and others is large. Thus, technical debt in Hadoop is paid back the quickest by creators, followed with a small margin by identifiers, followed with a large margin by others. In Camel, the situation is different as none of the time differences is statistically significant. We only observe that the repayment time of others is much longer (on average) than creators and identifiers.

TABLE X
REPAYMENT TIME COMPARISON BETWEEN DIFFERENT DEVELOPERS

Project	Average time spent on debt repayment (h)		p-value	Cliff's delta
	Creators	Identifiers		
Hadoop	128.0	1510.8	0.031	-0.303 (small)
Camel	174.5	142.3	0.935	0.021 (negligible)
	Creators	**Others**		
Hadoop	128.0	5730.3	0.028	-0.777 (large)
Camel	174.5	3104.3	0.851	-0.069 (negligible)
	Identifiers	**Others**		
Hadoop	1510.8	5730.3	0.080	-0.626 (large)
Camel	142.3	3104.3	0.463	-0.210 (small)

Most of the identified technical debt in issue trackers is resolved (on average 71.7%). Debt identifiers and creators pay off most of the technical debt (47.7% and 44.0% respectively). The median time and average time to repay technical debt are 872.3 and 25.0 hours. In Hadoop, technical debt creators resolve it quicker than those who identify it or others.

VI. DISCUSSION

Various types of technical debt are detected in issues, and they are complementary to those detected in source code comments. Comparing the types of technical debt we identified in issues (RQ1) against those types detected in source code comments by Potdat and Shihab [3], we find requirement, defect, design, test, and documentation debt appearing in both. However, although documentation and test debt are among the three most common types in issues, they are the two least common types in source code comments. Meanwhile, design debt is the most common type in source code comments, but it is rather uncommon in issues. Finally, code, build, and architecture debt are only detected within issues. This means that the types of technical debt detected through issue trackers and source code comments have some overlap but they are also sufficiently distinct. Thus, using each source (issues or source code comments) has its strengths and weaknesses. Therefore, we argue that the two sources are complementary in detecting different types of technical debt. Researchers should take both sources into account to increase the completeness and accuracy of their detection tools.

Approaches are required to identify technical debt in all three different cases (existing debt, during code review or after committing a patch). Researchers should customize the identification approaches according to the characteristics of each case (see results of RQ1). For example the approach proposed by Dai and Kruchten [5] can work for the first case but not for the other two cases. Furthermore, the findings show that only 13.1% of technical debt is reported by those that created it. We suggest that researchers look into this phenomenon and interview practitioners to find out why they usually do not report their own debt. Furthermore, we advise practitioners who deliberately incur technical debt, to report it in the issue tracker. This would accelerate the repayment of these debt items, even if that is performed by other developers.

Technical debt admitted in issues is resolved faster than in source code comments. Considering the results obtained from RQ3 in comparison with the study of Maldonado *et al.* [7], we find that most of the technical debt is repaid or removed (71.7% for debt within issues and 76.7% for debt in code comments). Furthermore, a great percentage of technical debt is repaid or removed by debt creators (44.0% for issues and 54.4% for source code comments). This indicates that developers consistently take care of SATD in both issues and source code comments, and debt creators often take the responsibility to resolve it.

Moreover, in Hadoop, it is noteworthy that debt creators repaid technical debt the fastest, followed by identifiers, and other developers. This is consistent with the intuition that certain people are better able to resolve TD depending on their familiarity with the problem at hand; creators being the most familiar, followed by debt identifiers, and others. Therefore, we suggest that, in order to pay off TD faster, the repayment task should be assigned to debt creators. In addition, comparing the TD repayment between issues and source code comments [7], we observe that debt within issues is resolved much quicker than in comments (i.e. for Hadoop, median of 2.0 days versus 159.0 days; for Camel, 0.9 days versus 18.2 days). Therefore, we suggest that developers report TD that needs to be resolved immediately in issue trackers instead of commenting it in the source code.

VII. Threats to Validity

Threats to Construct Validity concern the correctness of operational measures for the studied subjects. Since only a small subset of issues in issue trackers contain technical debt statements [4], the sample (500 analyzed issues) may not represent the population (issues containing technical debt in general). To minimize this threat, the analyzed sample was obtained randomly from all collected issues.

Threats to Reliability concern potential bias from the researchers in data collection or data analysis. Since issues are written in natural language, they were identified and categorized manually. To counter the threat of researchers biasing the manual analysis, the first and second author annotated the issue sample independently, and then discussed any differences to reach consensus on the classification. Additionally, the level of agreement (Cohen's kappa) was 0.757, which indicates high inter-rater agreement. Finally, as aforementioned all data are publicly available in the replication package.

Threats to External Validity concern the generalization of findings. In this study, we analyzed issues from two large open source projects, which both use JIRA as the issue tracker. Thus, our findings may be generalized to other open source Java projects of similar size and complexity that use JIRA; we can not claim any further generalization.

VIII. Conclusion

In this paper, we explored SATD in issue trackers. We found eight types of technical debt: architecture, build, code, defect, design, documentation, requirement, and test debt. Code, documentation, and test debt are the three most common technical debt found in issue trackers. Furthermore, there are three cases of identifying technical debt in issue trackers: discovering existing debt and creating an issue for it, identifying debt in a patch during code review, or after the patch is committed in the system. Most of the technical debt is identified in the first and second cases. Only 13.1% of technical debt is reported by debt creators.

For technical debt repayment, we found that on average 71.7% of identified debt is repaid, and most of it is paid by debt identifiers and creators (i.e. 47.7% and 44.0%). The median time and average time of debt repayment are 25.0 and 872.3 hours respectively. Our results show that in Hadoop, the repayment time by creators is statistically significantly shorter than that of identifiers and others. However, in Camel, we did not observe statistically significant differences between different types.

References

[1] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)," *Dagstuhl Reports*, vol. 6, no. 4, pp. 110–138, 2016.

[2] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, 2016.

[3] E. d. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2015, pp. 9–15.

[4] S. Bellomo, R. L. Nord, I. Ozkaya, and M. Popeck, "Got technical debt? surfacing elusive technical debt in issue trackers," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 327–338.

[5] K. Dai and P. Kruchten, "Detecting technical debt through issue trackers." in *QuASoQ@ APSEC*, 2017, pp. 59–65.

[6] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 91–100.

[7] E. d. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 238–248.

[8] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal? an in-depth perspective," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 526–536.

[9] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, "Goal Question Metric (GQM) approach," in *Encyclopedia of Software Eng.* Hoboken, NJ, USA: John Wiley & Sons, Inc., jan 2002, pp. 528–532.

[10] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *2012 9th IEEE working conference on mining software repositories (MSR)*. IEEE, 2012, pp. 199–208.

[11] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[12] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *2014 Sixth International Workshop on Managing Technical Debt*. IEEE, 2014, pp. 1–7.

[13] J. L. Fleiss, B. Levin, M. C. Paik *et al.*, "The measurement of interrater agreement," *Statistical methods for rates and proportions*, vol. 2, no. 212-236, pp. 22–23, 1981.

[14] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[15] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.