university of groningen

Mining Technical Debt in Commit Messages and Commit Linked Issues

Ai Deng

Supervisors:
Paris Avgeriou
Mohamed Soliman
Yikun Li

University of Groningen, 2020
Faculty of Engineering and Science

Acknowledgements

I would like to thank all my supervisors for making this thesis possible, especially my daily supervisor, Y. Li, for guiding me with my thesis during the past months, and giving me advice on progress and follow-up decisions during this study. It has been a tough time with the corona outbreak and numerous setbacks, but I am very grateful for the opportunity to be able to conduct this thesis study.

Abstract

Technical Debt(TD) is a metaphor describing the sub-optimal shortcuts taken during the software development life cycle. These shortcuts result in an easy approach or immediate solution in the short run but negative impacts and greater maintenance efforts in the long run. There are a number of studies which focused on the identification and detection of TD in publications, source code, source code comments and issue trackers. However rarely any study has explored TD in commit messages and commit linked issues.

This paper proposed a case study on identifying TD in commit messages and commit linked issues from open source industry. 1,000 commit linked issues as well as 847 commit messages from five open source Apache projects (*i.e.* Thrift, Camel, Hbase, Impala and Hadoop) were extracted and manually analyzed. Each issue consists of different sections, namely: a summary, a description and individual comments. The 1,000 issues consist of 6,034 sections in all. All these sections were analyzed independently. In total 1,338 TD items were found among the issue sections and 247 TD items were found among related commit messages.

The result shows:

1. Eight different types of TD are identified in commit messages and commit linked issues, including architecture, build, code, defect, design, documentation, requirement, and test debt.

2. Design, code, and documentation debt are the three most common TD types.

3. When a TD is found in a commit message, the likelihood to find the same TD in the commit linked issue is 33.2%.

4. Out of all commits with identified TD, 92.3% of commits resolved an existing TD while 8.9% of commits introduced new TD. There were 1.2% of them were found resolving TD and introducing new TD at the same time.

The results obtained from this study can be used to motivate future studies on TD in issue tracking systems as well as commit messages in the context of open source projects. The findings can also help developers to be more aware of the TD concept in any future software development project.

Contents

Chapter 1

Introduction

In a software development life-cycle, developers sometimes tend to take decision-making shortcuts. When these shortcuts result in immediate solutions or easy approaches in the short run but cause greater maintenance efforts and negative impacts to the development process in the long run, these shortcuts are known as Technical Debt(TD). TD is not necessarily unfavorable. It pushes the development process forward, reaching the phase of minimum viable product faster. However, as the size and complexity of the project grows bigger, an excessive amount of TD would make it more difficult to maintain and continue the project. It also creates extra work and frustration for the developers. Hence, it is crucial to find a balance between minimizing existing TDs and moving the project forward. In order to achieve this goal, it is necessary to conduct studies on identification and characterizing TD to obtain a comprehensive understanding on the TD concept. A number of studies have been conducted on detecting TD in literature, source code, source code comments, and issue tracking systems. However, TD in commit messages and commit linked issues is relatively unexplored.

Nowadays, most software development projects use a version control system to track the changes to the source code repository. A commit adds the most recent changes to the source code and makes these changes part of the latest revision of the repository. Git[1] is the most popular version control system used by most of software development projects. A new commit consists of the current contents of the index and the given log message describing the changes (Chacon & Straub, 2014). This log message is known as commit massage. On the other hand, in software development, the issue tracking system provides a ticketing system to record and follow the progress of every issue identified by the stakeholders, *e.g.* the development team, the customer etc. until the issue is resolved. An issue can be anything, *e.g.* a bug fix, a new feature implementation, code refactoring etc. JIRA[2] from Atlassian is one of the most popular issue tracking systems used by the software development community. In a traditional software development workflow, a new ticket is created when a new issue has been raised. This ticket would become a task assigned to the appropriate people. When developers make corresponding changes to the code or project, a commit is submitted. This commit would then be linked back to the corresponding issue. This research study mainly focuses on exploring TD in commit messages and commit linked issues.

The main goal for this study is to identify and quantify the different types of TD in commit messages and commit linked issues, and to find if there is a correlation between them. The following research questions(RQs) are addressed:

[1] `https://git-scm.com/`

[2] `https://www.atlassian.com/software/jira`

RQ1: *What types of TD are identified in commit messages and commit linked issues?*

RQ2: *When a TD is detected in a commit message, what is the likelihood of the same TD being found in the corresponding issue?*

RQ3: *How often does a commit message introduce or resolve TD?*

Five open source Apache projects are selected for this study. These projects use Git as their version control system and JIRA as their issue tracker. A case study was designed to examine and quantify the different types of TD in these projects. 1,000 commit linked issues and 849 related commit messages were extracted and manually examined. A refined classification method from an existing framework (Alves, Ribeiro, Caires, Mendes, & Spinola, 2014) (Y. Li, Soliman, & Avgeriou, 2020) was used to classify the messages and issues. In total, eight different types of TD were found. They are architecture, build, code, defect, design, documentation, requirement and test debt. The result also shows that design, code and documentation debt are the 3 most common TD types. Furthermore, the author examined the correlation between TD identified in commit messages and commit linked issues. It turns out when a TD was detected in a commit message, there are 33.2% of cases that the same TD was found in the commit linked issue. Finally, the author discovered that out of all commits with identified TD, 92.3% of commits resolved an existing TD while 8.9% of commits introduced new TD.

The results provide a number of discussion points. First of all, the TD detected in commit messages corresponds to the TD found in commit linked issues. However, greater corresponding rate can be achieved. When the developer addresses certain TD in a commit, he should address the same TD in the commit linked issue as well for greater consistency and formality. Also, most commits are found that they have resolved an existing TD in the system. Hence, when a commit is detected as introducing TD, it should get the attention and not be merged by anyone. After all, the findings from this study can be used to further motivate or compare with any future study on TD in commit messages and commit linked issues.

The remainder of this paper is structured as follows: Chapter 2 discusses the related work. Chapter 3 elaborates the method of case study design. Chapter 4 presents the results. Chapter 5 evaluates the threats to validity. Chapter 6 brings up couple of discussion points. Finally, Chapter 7 draws the conclusion.

Chapter 2

Related Work

This chapter summarises all related background studies regarding mining TD in commit messages and commit linked issues. The related work are divided into 3 major components: work related to identification of TD in general, work related to mining TD in issue trackers, and work related to commit messages.

2.1 Identification of TD in General

A number of studies have focused on the identification and classification of TD. For example, S. McConnell (McConnell, 2007) categorized TD into two groups: intentional TD and unintentional TD. In another related work, N. Alves *et al.* (Alves et al., 2014) proposed a thorough ontology on TD terms. In their work, they studied publications and gathered information related to TD. They organized the different types of TD based on their nature as a classification criterion. They mapped TD into 13 types: architecture, build, code, defect, design, documentation, infrastructure, people, process, requirement, service, test automation, as well as test debt. Associated with each TD type, indicators were used to identify certain type of TD. For example, 'duplicated code' and 'slow algorithm' are two examples of found indicators for code debt. This paper uses a similar ontology to define the basic types of TD, with high-level definitions and a list of indicators for each type.

Similar to N. Alves *et al.* (Alves et al., 2014), Z. Li *et al.* (Z. Li, Avgeriou, & Liang, 2015) provided a systematic mapping to identify and analyze TD. They studied the concept of TD in 94 selected primary studies from seven main publications and Google Scholar. They classified TD into 10 basic types: requirement, architectural, design, code, test, build, documentation, infrastructure, versioning, and defect debt. Moreover, they further classified these TD into sub-types based on the causes of TD. For example, 'duplicated code' is a sub-type of code debt. Among all these debts, Z. Li *et al.* found that code debt is the most frequently mentioned TD type in selected studies. They also explicitly mentioned that what should not be regarded as Non-TD. Non-TD has not received much attention from researchers and developers. However, understanding and knowing what is not considered as TD is helpful in some ambiguous cases. For example, should defects be considered as debts? There are four studies claiming that defects are not TD while defect debt is identified as TD by other 11 studies (Z. Li et al., 2015). In this research, only when a defect has been explicitly mentioned to be ignored in commit messages or commit linked issues, it is marked as a defect debt. Otherwise it will not be considered as a debt.

Alternatively to identifying TD in publications, there are studies that focused on identifying TD in source code comments. A. Potdar and E. Shihab (Potdar & Shihab, 2014)

proposed a method to manually identify and inspect TD in code comments from four different open source projects. They found that 2.4% - 31.0% of source files contained TD items. However, their method does not consider the different types of TD. In a follow-up study, E. Maldonado and E. Shihab (da S Maldonado & Shihab, 2015) examined and quantified different types of TD in source code comments from five open source projects. They detected five different types of TD, including design, defect, documentation, requirement and test debt. Compared to the works from N. Alves *et al.* (Alves et al., 2014) and Z. Li *et al.* (Z. Li et al., 2015), less types of TD are identified in source code comments.

There are also studies which mainly focused on one specific TD type. For example, in the work of N. Zazworka *et al.* (Zazworka, Shaw, Shull, & Seaman, 2011), the main focus was design debt. The study revealed that the existence of god classes can be seen as a strong technical design debt indicator. God classes refer to classes that are too complex, less coherent and overly coupling to other classes. God classes often have high refactoring potentials. N. Zazworka *et al.* studied the code from two commercial software projects and analyzed the change likelihood and defect likelihood of god classes. They discovered that god classes are more likely to be part of a change/defect fix in contrast to non-god classes. Hence, there is a strong correlation between the existence of god classes and TD. N. Zazworka *et al.* 's focus of design debt is expected because in this paper, the author discovered that design debt is one of the most frequently occurring TD in commit messages and commit linked issues. Therefore, paying more attention to design debt would improve the software development life-cycle progress.

All in all, this paper separates itself from the works described above. Instead of publications, source code or source code comments, the focus is mainly on commit messages and commit linked issues. In this paper, TD is manually inspected and identified in commit messages and commit linked issues. However, the ontology proposed by N. Alves *et al.* (Alves et al., 2014) provides the basic understanding and framework of identifying TD in this paper.

2.2 Mining TD in Issue Trackers

Issue tracking systems are widely used in collaborative projects in the software industry to track, prioritize and file different types of issues happened during the life-cycle of software development (Dai & Kruchten, 2017). These issues can be implementations of new features, bug fixings, code refactoring, documentation updates etc. Issue trackers are very crucial in assisting software developing teams to manage development and maintenance activities in order to achieve success and efficiency in software projects (Dai & Kruchten, 2017). Each issue has its particular life cycle, from the timestamp of being created to the timestamp of being closed, including the following steps: creating the issue, discussing and creating a patch, code reviews, updating the patch, code reviews again, and a final code commit (Y. Li et al., 2020). Useful information can be retrieved from these issues to improve the development management and quality of evaluation (Dai & Kruchten, 2017).

A couple of studies focus on the identification and detection of TD specifically in issue trackers. S. Bellomo *et al.* (Bellomo, Nord, Ozkaya, & Popeck, 2016) manually examined 1,264 issues from four projects, including the Chromium and Connect Health IT Exchange open source projects and two government IT projects. They proposed a classification method to identify TD in issue tracking systems. According to this method, issues related to new feature implementations and fixing of incorrect functionality are not considered as technical debt issues. Only issues that are design related and have shown negative side effects are

considered as technical debt issues. For example, the following issue

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

is not considered as a technical debt issue as the evidence of side effects is not clear. However, this issue would be considered as a low-quality code debt issue in this study, evidently due to the clearly mentioned 'refactor' that implies the existence of non-well-written code. Out of the 1,264 issues, S. Bellomo *et al.* identified 51 issues in which TD was detected. However, they did not classify TD into different types based on the nature or cause. Besides the classification approach, S. Bellomo *et al.* also examined the number of issues where the developers explicitly used the term '*technical debt*' and related concepts, *e.g.* taking on, accumulating, and paying back debt. It turns out 58 issues were found, which is seven more than the implicit technical debt issues. This shows that, in present days, developers have become more aware of the TD metaphor and pay more attention on reducing and eliminating TD accumulation in software development projects.

In another study, K. Dai and P. Kruchten (Dai & Kruchten, 2017) analyzed 8,149 issues from a Chinese commercial software project. Similar to S. Bellomo *et al.* , they first tagged the issues manually according to a classification criterion. Issues related to requirement change, new features, insufficient description and critical defects are not considered as technical debt issues. Being different from S. Bellomo *et al.* , they further classified TD into 6 main types, including defect, requirement, design, code, UI, and architecture debt. Out of the 8,149 issues, 331 technical debt issues were found. Design and requirement debt are the two most common TD types, including 141 and 105 instances respectively. Subsequently, they extracted key words, phrases and features from existing issues and trained a classifier by using natural language processing and machine learning techniques to detect potential technical debt issues automatically. Their experimental results show that text patterns indicating TD exist and can be used to identify TD. Similar to S. Bellomo *et al.* , K. Dai and P. Kruchten (Dai & Kruchten, 2017) also searched for the term '*technical debt*' in the issues. However, the result differs from S. Bellomo *et al.* . No positive result was found. This revealed that the developers from this selected project were not aware of TD metaphor. K. Dai and P. Kruchten recommended the developers to use TD as an issue type in the issue tracking system to communicate TD explicitly (Dai & Kruchten, 2017).

In another issue tracker related study, Y. Li *et al.* (Y. Li et al., 2020) manually analyzed 5,00 issues from two open source projects. They classified the sample issues using framework from N. Alves *et al.* (Alves et al., 2014), which was discussed previously in chapter 2.1. They annotated sentences within these issues according to the definition of different types of TD in N. Alves *et al.* (Alves et al., 2014). However, they made refinements to the types and indicators of TD in the original framework. For example, they added the indicator *"Requirements Partially Implemented"* to the requirement debt type. Out of the 500 issues, they found 152 technical debt issues, i.e. 30.4% of the issues are related to TD, which is more than previous mentioned studies. An explanation for this is that instead of only looking at the summary and description of each issue, Y. Li *et al.* annotated the issues on a sentence level. Hence, an issue might consist of more than one type of TD. They found that out of all the technical debt issues, 20% of them contained more than one type of TD. Y. Li *et al.* detected 8 different types of TD, namely architecture, build, code, defect, design, documentation, requirement and test debt. Code, documentation and test debt are the most frequently found TD types, making up of 38.8%, 21.7% and 18.4% respectively. Additionally, they also mentioned that only 15.9% of TD is reported by the issue creators while other TD is reported by other developers in the comment sections. Moreover, on average, 71.1% of identified TD is repaid.

This paper differs from the aforementioned works. First of all, unlike S. Bellomo *et al.* (Bellomo et al., 2016), K. Dai and P. Kruchten (Dai & Kruchten, 2017) who analyzed each issue as a whole by looking at the description and summary of each issue, or Yikun Li *et al.* (Y. Li et al., 2020) who annotated each issue on a sentence level, this study dissected each issue into different sections: a summary, a description and every individual comment. Each section was analyzed and annotated completely independently. The reason for doing this is to make sure any necessary information related to TD from an issue would not be missed out. Also, it provides convenience for future follow-up studies that want to develop machine learning tools to analyze each issue. It makes more sense for machine to learn each section separately. Even though this study used the same classification method in Y. Li *et al.* (Y. Li et al., 2020), the author made minor refinement to the types and indicators of TD to suit the data-set best. This will be discussed in details in Chapter 3. Another big difference between this study and the aforementioned works is that this study also analyzed and classified commit messages. It focuses on the result comparison between commit messages and commit linked issues. To the best of the author's knowledge, so far, not many studies have focused on the detection and comprehension of TD in commit messages yet.

2.3 Commit Messages

A well-written commit message is an effective way to communicate context about a change to fellow developers, and to their future selves (Beams, 2014). Commit messages contain substantial amount of useful information about the current change. This includes the origin of the problem, why a change was made, and any limitations of the current code. Different studies examined and analyzed the commit messages. For example, C. Rosen *et al.* (Rosen, Grawi, & Shihab, 2015) developed a tool, *Commit Guru*, which enables commit-level analytic and predictions by using a number of metrics. It automatically identifies TD-inducing commits and builds a prediction model that estimates the probability of a commit inducing potential bug. In their work, they used a list of associated words to classify commits into six categories: corrective, feature addition, merge, non-functional, perfective, and preventive. Then they performed *diff* to link a fixing commit to a bug-inducing commit. Once the fixing and the risky commits are labeled, *Commit Guru* is able to identify commits that are more likely prone to bugs. Hence, C. Rosen *et al.* have not only analyzed the commit messages, but they also analyzed the commits themselves to better predict potential risky commits. However, in this study, the author mainly focuses on the detection and quantification of different types of TD in commit messages.

Chapter 3

Approach

The main goal of this research study is formulated according to the Goal-Question-Metric approach proposed by Dana Sulistiyo Kusumo *et al.* (Kusumo, Sabariah, & Wiharja, 2017):

> ***Analyze*** *selected commit messages and commit linked issues* ***for the purpose of*** *identifying the TD* ***with respect to*** *the types, the correlation between TD found in messages and issues, and the repayment of TD in commit messages* ***from the point of view*** *of software developers and researchers* ***in the context of*** *open source projects.*

The goal is broken down into the following RQs:

- **RQ1**: *What types of TD are identified in commit messages and commit linked issues?.* This can be broken down into two sub-questions:
  - **RQ1.1**: *What are the different types of TD identified in commit linked issues?* Rationale: Identifying different types of TD in commit linked issues could help us comprehend TD concepts in the context of issue tracking systems. For example, certain type of TD might be only identified in issues. This information may lead to future research directions.
  - **RQ1.2**: *What are the different types of TD identified in commit messages?* Rationale: Identifying different types of TD in commit messages could help us comprehend TD concepts in the context of commit messages. For example, certain types of TD might be only identified in commit messages. To the best of the author's knowledge, so far, not many studies have focused on the detection and comprehension of TD in commit messages.
- **RQ2**: *When a TD was detected in a commit message, what is the likelihood of the same TD being found in the corresponding issue?* Rationale: Finding the correlation between TD in commit messages and commit linked issues could help the researchers understand the strengths and limitations of TD in different contexts. This could also help in proposing case studies for identifying TD that combine different sources.
- **RQ3**: *How often does a commit message introduce or resolve TD?* Rationale: Quantifying how much TD is paid off or induced in commit messages helps us understand developers' attitudes towards TD in the context of open source projects. It can also help to increase the awareness of TD concepts among developers.

The flow of the approach can ideally be described using a flowchart displaying the different phases of the project (Figure 3.1).

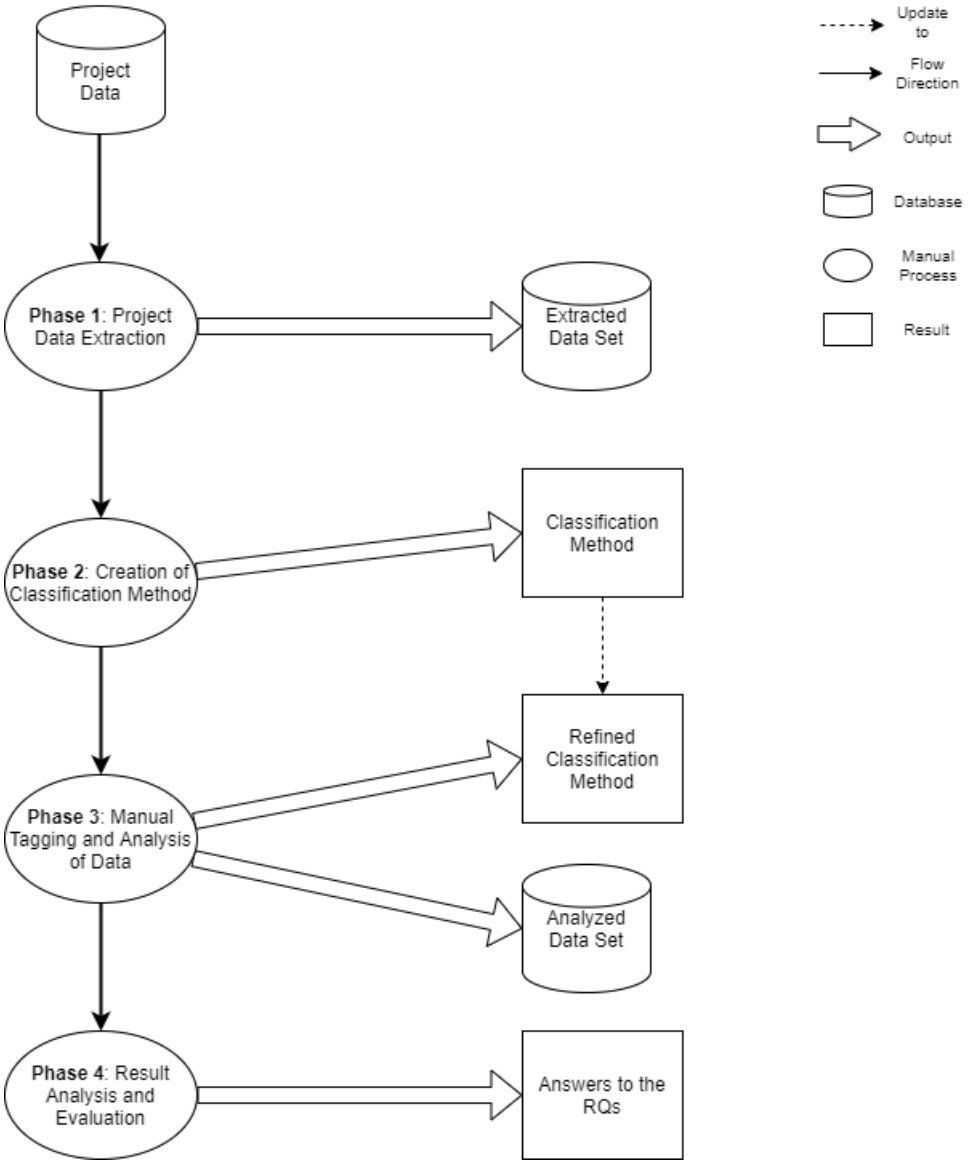

Figure 3.1: Overview of the approach

The approach contains four different phases: project data extraction, creation of classification method, manual tagging and analysis of data, and result analysis and evaluation. Each phase either outputs a result or an extracted/analyzed database. The following subsections provide details of each phase.

3.1 Phase 1: Project Data Extraction

To perform this research study, the supervisor Y. Li selected five open source Apache projects for the author to study and analyze. The reason for this is that when the author later manually tagged and analyzed the issues and messages in phase 3, she did not have any prior knowledge about these selected projects. Hence, biased decision making was successfully avoided.

In order for projects to be applicable to the context of studying TD in commit messages and commit linked issues, the project selection criteria are the same as in the work from Y. Li *et al.* (Y. Li et al., 2020). These criteria are as follows:

- All selected projects need to be publicly available, well-maintained and well-

documented. They need to use commits and issue tracker during the project development.

- All selected projects need to be middle to large sized projects, *i.e.* at least 2,000,000 source lines of code (SLOC), 1,000 issues, 10,000 commits involved in the projects. This is to ensure sufficient complexity.

Based on these criteria, five projects were selected: Thrift[1], Impala[2], HBase[3], Camel[4] and Hadoop[5]. These five projects are well-documented and highly active. They all use Git as their version control system and JIRA as their issue tracker.

After selecting these five projects, issues and commit messages were extracted and stored in a local database. Next, issues and commit messages were filtered. For issues, all *open* or *pending closed* issues were filtered out as this study is only interested in issues that have completed an entire life cycle, *i.e. closed* and *resolved* issues. Moreover, issues without corresponding commits were filtered out as well, as the research needed to compare the results obtained from issues and commit messages. Considering the large amount of filtered issues, for this research study, a subset of this data-set was selected for the author to do manual analysis. A random sample of 250 issues respectively from each project was selected. Afterwards, all the commits that were related to the 1,000 issues were extracted and manually analyzed as well. Thus, in total, this phase gave an output of an extracted data-set, which contains 1,000 commit linked issues and 849 commit messages. This data-set will be manually analyzed in the future phase 3.

3.2 Phase 2: Creation of Classification Method

In phase 2, after studying and extracting all relevant concepts and research studies related to TD, and the identification of TD in different contexts, the author created the classification method that suits best in the context of this research study. In cases of TD, the author used the same classification method from Y. Li *et al.* (Y. Li et al., 2020). This classification method is based on an existing framework proposed by N. Alves *et al.* (Alves et al., 2014).

N. Alves *et al.* (Alves et al., 2014) proposed an ontology to identify different types of TD and their respective indicators. According to this ontology, the types of TD were defined considering their nature and cause. N. Alves *et al.* performed a systematic literature mapping to come up with the definitions of thirteen different types of TD, namely: architecture, build, code, defect, design, documentation, infrastructure, people, process, requirement, service, test automation, as well as test debt. Along with the definition of each TD type, indicators are used to identify the TD. Although, for some types, *i.e.* process, infrastructure, test automation, and people debt, no indicator has yet been found. Among all the types, some of them can overlap with each other while some of them have disjoint classes restriction. For example, a problem related to design can be classified as design debt as well as architecture debt at the same time. However, it seems unlikely for a problem to be a process and test debt at the same time. N. Alves *et al.* came up with an exhaustive disjoint classes table in their research.

Even though N. Alves *et al.* have provided comprehensive concepts related to TD and an

[1] `https://thrift.apache.org/`
[2] `http://impala.apache.org/`
[3] `https://hbase.apache.org/`
[4] `https://camel.apache.org/`
[5] `https://hadoop.apache.org/`

exhaustive classification method of TD, the different types of TD they came up with were identified in the context of technical literature, which differs from the context of this study. Y. Li *et al.* (Y. Li et al., 2020) have refined the classification guidance proposed by N. Alves *et al.* . They proposed a classification method which is more applicable in the context of detecting TD in issue trackers. In their method, infrastructure, people, process, service, and test automation debt are not relevant anymore. Also, most of the indicators for each type have been redefined. For example, *betweeness centrality* which is a found indicator for architecture debt (Alves et al., 2014) does not apply in the context of issue trackers and hence became obsolete. As this research mainly focused on the detection of TD in issue trackers as well as commit messages, the classification method that used by Y. Li *et al.* was used. However, the author also did minor refinement on their classification method to suit the best with the data-set during this study in phase 3. For example, the indicator *flaky test* is later added to the test debt. Also, some indicators, like *non-functional requirements not fully satisfied* just does not apply to the data-set and hence became obsolete. Moreover, the indicator *multi-thread correctness* has been renamed to *multi-thread incorrectness* as the author claims the word *incorrectness* suits the context of debt more.

Besides different types of TD, it is also crucial to define types that are considered as Non-TD. Couple of studies explicitly mentioned a list of scenarios that should not be regarded as TD. In the cases of Non-TD, the author used the existing classification method proposed by K. Dai and P. Kruchten (Dai & Kruchten, 2017) with some refinements. K. Dai and P. Kruchten also did the detection of TD in issue trackers, hence their method suits the context of this study. They came up with four different types of Non-TD issues based on their nature and cause: requirement change, new features, critical defects and insufficient description. However, for the data-set of this study, the requirement change is not applicable, because issues were broken down into small sections and all these sections were analyzed and annotated independently. Usually these sections are very small, consisting of just a couple of sentences. It is challenging for the author to judge if such a small section is related to a requirement change. This also counts for commit analysis, as normally a commit message can be even shorter which hardly contains any information where one can decide if it is a requirement change. Furthermore, in the study of (Dai & Kruchten, 2017), when a defect comes from not correctly implemented functions or features which are important and critical, it is considered as a Non-TD type. When a defect is not critical from the client's perspective but weakens the performance and capabilities of the system and will be fixed in the future, it is considered as a TD type(Dai & Kruchten, 2017). This differs from this paper. In this study, when it comes to defect case, only when a defect happened and it explicitly stated that this defect was ignored for now, this defect is considered as a defect debt. Otherwise it is a Non-TD type. Hence in this study, requirement change became an obsolete Non-TD type, and critical defect was refined to defect as one of the Non-TD types.

The following Table 3.1 presents the classification method of Non-TD and TD used by this research study.

Label	Type	Indicator	Definition
Non-TD	Requirement Change *	—-	The request for requirement change from the client.
	New Features		Tasks to add new function or introduce new features.
	Insufficient Description		The description is insufficient to make a decision.
	Defects **		Known defects that induce bugs in the system.
TD	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.
		Containing outdated or empty files ***	Containing outdated or empty files causes messy architecture structure.
	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 incorrectness **	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.
		Memory leak ***	Code that has memory leak.
	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.
		Flaky test ***	Tests that are flaky, i.e. it would fail or pass for the same configuration.

Table 3.1: The Refined Classification Method.
Note. Adapted [reprinted] from (Y. Li et al., 2020) and (Dai & Kruchten, 2017)

Any type with * means it is a type from an existing framework (Dai & Kruchten, 2017) and (Y. Li et al., 2020) but became obsolete for this research, mainly because this type is not applicable for the selected data set. This also applies for any indicator with *, which means it is an indicator from the study of Y. Li *et al.* (Y. Li et al., 2020) but became obsolete for this research. Any type with ** means a type from an existing framework (Dai & Kruchten, 2017) that gets refined to a new type in the context of this study. Any indicator with *** means it is a newly created indicator during the manual analysis in phase 3. The final version of this table is an output of phase 3. During phase 3, the classification guidance was refined to suit the best to the data-set.

3.3 Phase 3: Manual Tagging and Analysis of the Data

In phase 3, the 1,000 issues and 849 commit messages obtained from phase 1 were manually annotated and tagged according to the classification method obtained in phase 2. At the point of this phase, the author did not have any domain knowledge about the selected projects, which makes it possible to avoid any biased decision making.

To analyze the selected issues, each issue was dissected into different sections: a summary, a description and every individual comment. Figure 3.2[6] shows an incomplete snap-shoot of one example issue *THRIFT-418* on JIRA interface.

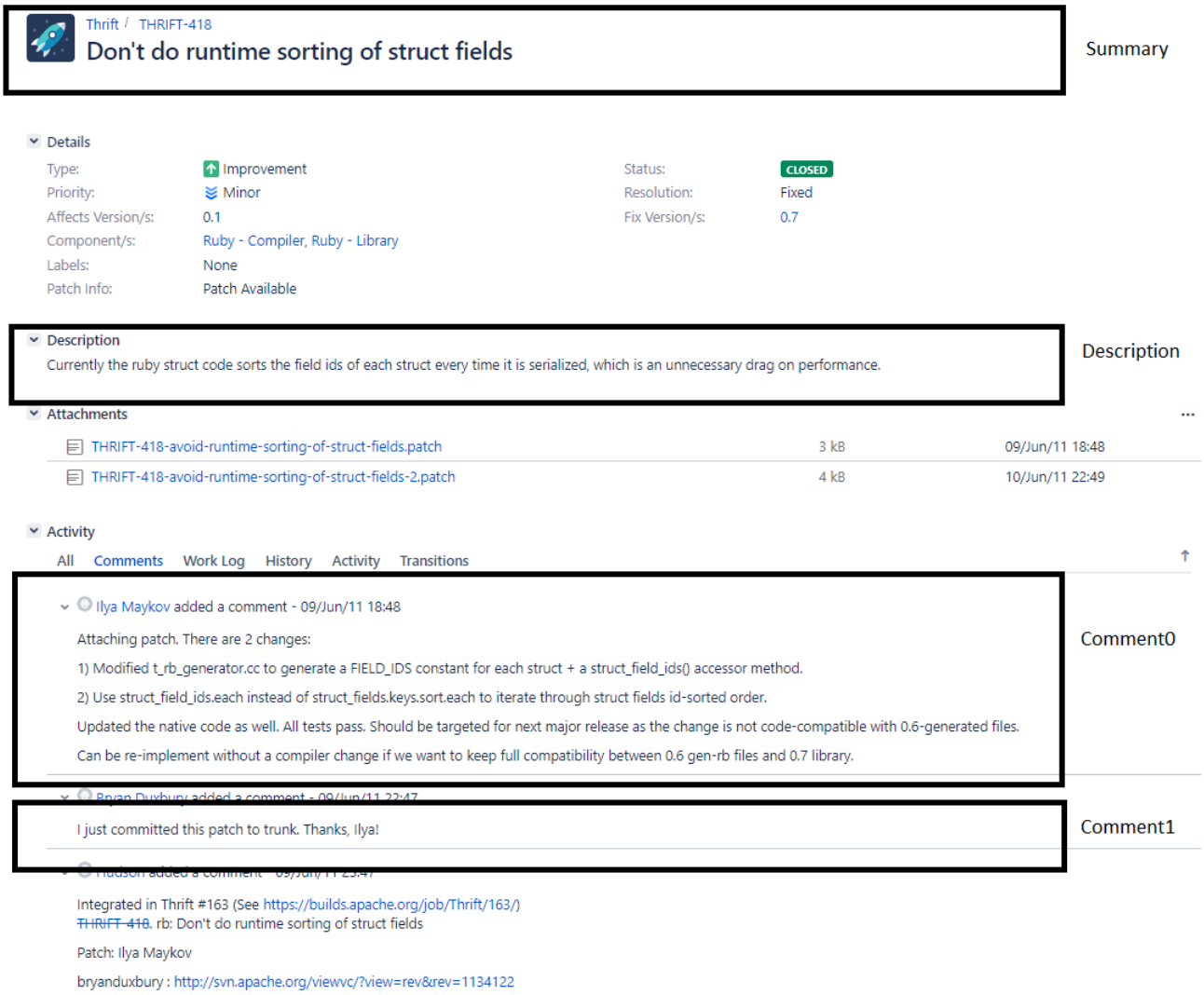

Figure 3.2: Example of issue *THRIFT-418*

For this issue which has issue key *THRIFT-418*, the summary is

> *"Don't do runtime sorting of struct fields"* - [THRIFT-418:s]

the description is

> *"Currently the ruby struct code sorts the field ids of each struct every time it is serialized, which is an unnecessary drag on performance."* - [THRIFT-418:d]

and with following 6 comments. One of them(the 2nd one) is

> *"I just committed this patch to trunk. Thanks, Ilya!"* - [THRIFT-418:1]

All these sections were extracted and were later were analyzed and annotated independently. The reasons for separating one issue into sections and studying them separately are the following:

[6] `https://issues.apache.org/jira/browse/THRIFT-418`

1. Summary and description contain essential information about an issue. They are normally written by the issue creator. They give the reader an initial idea about what the issue is and what type of TD might be involved. They should be analyzed separately from the rest of comments.

2. Each comment might come from a different code reviewer. They might contain a different perspective and opinion on the issue, which means different types of TD might be found in different comments. Hence, they should be analyzed individually.

3. To further divide each section into sentences does not make much sense as normally each section just contains one idea. And most likely multiple sentences in one section just describe one related thing.

4. Dividing an issue into these sections provides convenience to future follow-up studies to develop machine learning tools that analyze and detect TD automatically, as such tools would treat each section independently.

This separation of issues resulted in 6,034 sections to analyze. An open source text annotation tool **doccano**(Nakayama, Kubo, Kamura, Taniguchi, & Liang, 2018) was used to annotate each section. All tags for different indicators of TD as well as non-TD were created to annotate each section. **Doccano** shows each section at a time and gives the list of all created tags that can be manually assigned to the issue section.

The same has been done with all related commit messages as well. The commit messages from the Git(the source code repository) that are related with the selected issues from above were extracted. For example, the following Figure 3.3[7] shows a snap-shoot of one commit message on Github interface, which is related to issue *THRIFT-418*.

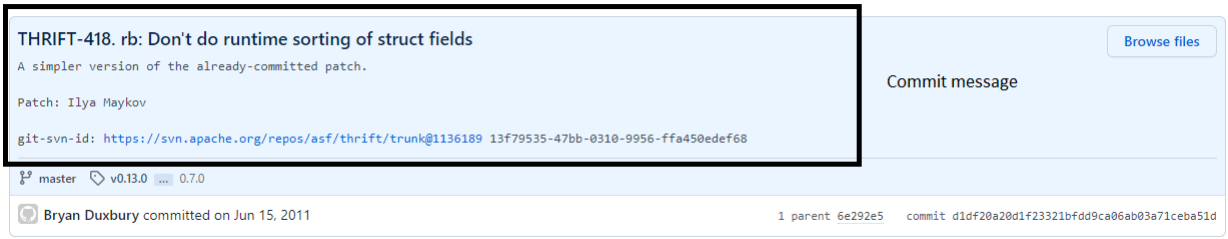

Figure 3.3: Example of commit message

A commit message contains a summary as well as a message body. However, unlike an issue being split apart into multiple sections, one commit message was annotated as a whole unit. This resulted in 849 commit messages being extracted. The same annotation tool doccano was used. Although, in addition to tags of different indicators of TD and non-TD, a couple of tags like *debt-introduced* and *debt-resolved* were added as well. This was done for the purpose of answering RQ3 in next phase.

This phase gave an output of a set of items classified as TD, 471 from all the issue sections and 247 from the commit messages.

3.4 Phase 4: Result Analysis and Evaluation

In phase 4, all identified issue sections and commit messages from phase 3 were revised again. The results were analyzed and compared in this phase. The incentive for this phase

[7] `https://github.com/apache/thrift/commit/d1df20a20d1f23321bfdd9ca06ab03a71ceba51d`

was to see if any correlation and conclusion can be drawn from the results, which might be helpful for any future open source projects. All the RQs mentioned in the beginning of this chapter were answered in this phase. For RQ1, the author looked at the results for issues and commit messages separately where a conclusion could be drawn for each of them. For RQ2, the author investigated if there was correlation between the results of the two. The author took all commit messages in which TD is identified, and looked up if the same TD has been identified in the commit linked issues. For RQ3, the author looked at the amount of commits with the tags *debt-introduced* and *debt-resolved* to quantify how much TD is paid off. The output of this phase will be exhaustively discussed in the next chapter.

Chapter 4

Results and Analysis

In this chapter, the author answers all RQs mentioned in previous chapter.

4.1 RQ1: What types of TD are identified in commit messages and commit linked issues??

4.1.1 RQ1.1: What are the different types of TD identified in commit linked issues?

Before discussing what types of TD are found, it is fundamental to set solid boundaries for scenarios in which Non-TD is defined.

4.1.1.1 Non-TD issues

In this study, the author found 3 types of Non-TD from selected data-set. For each type, one example has been listed below, with quote and the issue key included.

1. **New features**: when the issue is about adding new functions or features to the product, it is considered as Non-TD (Dai & Kruchten, 2017). For example:

   "Add Async implementation via IFuture" - [THRIFT-4422:s]

   In the above issue summary, the developer pointed out that this issue was about an implementation of a new feature.

2. **Defects**: when it is a defect that induces bugs in the system, it is considered as Non-TD. For example:

   "Thrift's D library/test: parts of "make check" code do not compile with recent dmd-2.062 through dmd-2.064alpha" - [THRIFT-2130:s]

   In the above issue summary, the developer pointed out that this issue was about a bug fixing.

3. **Insufficient description**: when the description of the issue or commit is too short or insufficient to decide whether it is a TD item, it is considered as Non-TD (Dai & Kruchten, 2017). For example:

   "please provide a patch" - [THRIFT-972:0]

   In the above issue comment, no useful information can be retrieved about if it is related to any type of TD.

4.1.1.2 TD issues

There were eight different types of TD identified in commit linked issues, namely architecture, build, code, defect, design, documentation, requirement and test debt. For each type, there were one or more indicators. For each indicator, one example is provided below, with quote and the issue key included.

1. **Architecture debt**

   - *Violation of modularity*: bad architecture design caused by wrong modularity. For example:

     > *"...fixed that, the wrong libthrift.jar file location and updated the changelog. => committed"* - [THRIFT-1141:0]

     In the above issue section, the developer stated that a file had been placed in the wrong module.

   - *Using obsolete technology*: bad architecture design caused by not upgrading the old technology stack. For example:

     > *"Gem::InstallError: byebug requires Ruby version >= 2.4.0."* - [THRIFT-5100:s]

     In the above issue section, the developer stated that some newer updated version of a certain technology stack is required.

   - *Containing outdated or empty files*: bad architecture structure caused by non-deleted outdated or empty files.

     > *"Thrift library has some empty files that haven't really been deleted"* - [THRIFT-2020:s]

     In the above issue section, the developer stated that there were some empty files to be deleted.

2. **Build debt** - *Under- or over-declared dependencies*: build failure caused by missing dependencies or overly complicated dependencies.

   - *Under-declared dependencies*:

     > *"...As a result, the code generated by libthrift no longer builds as it misses this dependency. The dependency needs to be restored..."* - [THRIFT-3168:d]

     The developer stated that some necessary dependency was missing.

   - *Over-declared dependencies*:

     > *"oro is a transitive dependency from commons-net. Feel free to remove it from ivy.xml though, as it'll..."* - [HADOOP-7161:0]

     The developer pointed out that some unnecessary dependency could be removed.

3. **Code debt**

   - *Complex code*: source code that is too complicated.

     > *"Another suggestion... This would simplify the code base quite a bit and remove the need to maintain this abstraction layer."* - [THRIFT-2028:4]

     This is an example of a code debt that is related to complex code. The developer stated that the code could be simplified.

- *Dead code*: unused code including declared but unused variables, methods, imports etc.

  > *"The following variables appear to no longer be used..."* - [HADOOP-12733:d]

  The developer stated that there were some unused variables, which should be removed.

- *Duplicate code*: code that occurs more than once or which achieves the same functionality.

  > *"Reduce duplicate code in thrift CMakeLists.txt"* - [IMPALA-9543:s]

  The developer stated that there was duplicate code in certain file that should be removed.

- *Low-quality code*: code that has low quality, for example: incorrect code, typo, malformed format, and variables or methods with wrong types.

  > *"...I think the added line {{ENUM = 16}} in {{types.rb}} is incorrect. If you compare with any other language, there is no such entry for enums."* - [THRIFT-2143:1]

  As mentioned in the issue section, certain lines of code were incorrect.

- *Multi thread incorrectness*: code is not thread-safely implemented.

  > *"MemTracker::EnableReservationReporting() is not thread-safe"* - [IMPALA-5130:s]

  The above example expressed thread-safe code that was not correctly implemented.

- *Slow algorithm*: a sub-optimal algorithm, data structure, or library is used.

  > *"Perl client library for Thrift is bit slow.This implementation can make 24.75 QPS.*
  > *I wrote a I/O buffering patch for this implementation.This makes 147x faster the echo server..."* - [THRIFT-593:d]

  This above example mentioned a faster algorithm replaced the old one.

- *Memory leak*: code that does not free all the memory.

  > *"Memory leak with ExprCtxs not free"* - [IMPALA-4027:s]

  The developer pointed out that a certain pointer had not been freed, which resulted in memory leaks.

4. **Defect debt** - *Uncorrected known defects*: bugs that are ignored or deferred to be fixed in the future.

   > *"I am disabling haskell in windows cmake builds until someone can resolve this."* - [THRIFT-4545:1]

   In the above example, instead of resolving an existing issue, the developer chose to disable the builds.

5. **Design debt** - *Non-optimal decision*: poor or non-optimal implementation.

   > *"The performance of the named pipes Delphi server is sub-optimal. Furthermore, BYTE message modes should be used (instead of MESSAGE)"* - [THRIFT-2415:d]

In the above example, the developer stated that sub-optimal performance exists. Hence, better design was needed to improve the performance.

6. **Documentation debt**

   - *Outdated documentation*: the documentation is not up to date.

     "DOC: Update docs to reflect that Deflate is supported for text files" - [IMPALA-9431:s]

     The developer stated that an updated docs was needed when some new functionality was added.

   - *Low quality documentation*: the documentation is incorrect or unclear.

     "...Thought it was strange the docs were wrong for this long. Was going to ask for a sanity check on this JIRA ..." - [HADOOP-16291:1]

     The developer pointed out there was some mistakes in the documentation for a prolonged time.

7. **Requirement debt** - *Requirements partially implemented*: requirements are not fully implemented.

   "Dfs needs a validation operation similiar to fsck, so that we get to know the files that are corrupted and which data blocks are missing.
   Dfs namenode also needs to log more specific information such as which block is replication or is deleted. So when something goes wrong, we have a clue what has happened." - [THRIFT-240:s]

   The developer pointed out that some requirements had not been implemented yet.

8. **Test debt**

   - *Expensive tests*: tests are slow.

     "The unit tests for the camel-hazelcast component use real HazelcastInstance objects, which is very slow..." - [CAMEL-6826:d]

     The developer pointed out that some unit tests took a long time.

   - *Lack of tests*: tests are missing.

     "Would you be able to build an unit test of this sample code so we can take that and add to the tests of camel-cxf and work on a fix." - [CAMEL-12104:0]

     The developer stated that some tests needed to be added.

   - *Low coverage*: code that has low test coverage.

     "Improve Ranger test coverage" - [IMPALA-8248:s]

     The developer stated that the test coverage needed to be improved.

   - *Flaky test*: tests are flaky.

     "This test is flaky, I can easily make it fail:..." - [THRIFT-2431:d]

     The developer pointed out that a certain test is a flaky test.

In total, eight different types and 20 different indicators of TD were identified in all the commit linked issues. It is worth to notice that in some cases, more than one TD were identified. For example, in the following issue comment section:

"Added missing import.
Could not find any test class for HTTPServer2. I will try to add tests for this class" - [HADOOP-11677:3]

Both "low-quality" code and "lack of tests" indicators were detected in this issue section. Some missing import makes it low-quality code and some missing tests makes it lack of tests.

Out of 6,034 issue sections (970 from Thrift, 793 from Impala, 1685 from Hbase, 1625 from Hadoop and 961 from Camel), there were 1,338 TD items that have been identified (293 from Thrift, 322 from Impala, 295 from Hbase, 321 from Hadoop and 107 from Camel). Considering each project has different contributors, coding standards, qualities, review processes and development cycles, the reader might be interested in the differences for each project. Hence an overview of TD types and indicators in every project is presented below. Afterwards, an overview table is presented.

Type	**Indicator**	#[1]	#	%[2]
Architecture Debt	Violation of modularity	8	16	5.4
	Using obsolete technology	4		
	Containing outdated or empty files	4		
Build Debt	Under- or over-declared dependencies	9	9	3.0
Code Debt	Complex code	10	125	42.7
	Dead code	13		
	Duplicated code	2		
	Low-quality code	96		
	Multi-thread incorrectness	2		
	Slow algorithm	1		
	Memory leak	1		
Defect Debt	Uncorrected known defects	6	6	2.0
Design Debt	Non-optimal Decisions	105	105	35.8
Documentation Debt	Outdated documentation	1	8	2.7
	Low-quality documentation	7		
Requirement Debt	Requirements partially implemented	4	4	1.4
Test Debt	Expensive tests	2	20	6.8
	Lack of tests	15		
	Low coverage	2		
	Flaky test	1		

Table 4.1: Types and Indicators of TD in the Thrift issue

As shown in the table, code debt(42.7%), design debt(35.8%) and test debt(6.8%) are the three most common types in the Thrift issues. Moreover, non-optimal decisions, low-quality code and lack of tests are the three most common indicators in the Thrift issues. Below is the result summary table for the Camel issues:

[1]The symbol # refers to the number of instances. The same applies to all other tables in this chapter.

[2]The symbol % refers to the percentage of total instances. The same applies to all other tables in this chapter.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	1	1	0.9
	Using obsolete technology	0		
	Containing outdated or empty files	0		
Build Debt	Under- or over-declared dependencies	2	2	1.9
Code Debt	Complex code	0	16	15.0
	Dead code	2		
	Duplicated code	0		
	Low-quality code	12		
	Multi-thread incorrectness	2		
	Slow algorithm	0		
	Memory leak	0		
Defect Debt	Uncorrected known defects	0	0	0.0
Design Debt	Non-optimal Decisions	51	51	47.7
Documentation Debt	Outdated documentation	17	26	24.3
	Low-quality documentation	9		
Requirement Debt	Requirements partially implemented	2	2	1.9
Test Debt	Expensive tests	4	9	8.4
	Lack of tests	4		
	Low coverage	1		
	Flaky test	0		

Table 4.2: Types and Indicators of TD in the Camel issues

Design debt(47.7%), documentation debt(24.3%) and code debt(15.0%) are the three most popular types in Camel while non-optimal decisions, outdated documentation and low-quality code are the three most common indicators.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	5	7	2.4
	Using obsolete technology	1		
	Containing outdated or empty files	1		
Build Debt	Under- or over-declared dependencies	1	1	0.3
Code Debt	Complex code	0	70	23.7
	Dead code	10		
	Duplicated code	2		
	Low-quality code	52		
	Multi-thread incorrectness	1		
	Slow algorithm	5		
	Memory leak	0		
Defect Debt	Uncorrected known defects	1	1	0.3
Design Debt	Non-optimal Decisions	153	153	51.9
Documentation Debt	Outdated documentation	11	40	13.6
	Low-quality documentation	29		
Requirement Debt	Requirements partially implemented	1	1	0.3
Test Debt	Expensive tests	0	22	7.5
	Lack of tests	10		
	Low coverage	1		
	Flaky test	11		

Table 4.3: Types and Indicators of TD in the Hbase issues

The three most common types in the Hbase issues are: design debt(51.9%), code debt(23.7%) and documentation debt(13.6%). The three most common indicators are non-optimal decisions, low-quality code and low-quality documentation.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	6	10	3.1
	Using obsolete technology	1		
	Containing outdated or empty files	3		
Build Debt	Under- or over-declared dependencies	3	3	0.9
Code Debt	Complex code	5	52	16.1
	Dead code	1		
	Duplicated code	3		
	Low-quality code	26		
	Multi-thread incorrectness	5		
	Slow algorithm	10		
	Memory leak	2		
Defect Debt	Uncorrected known defects	5	5	1.6
Design Debt	Non-optimal Decisions	195	195	60.1
Documentation Debt	Outdated documentation	12	37	11.5
	Low-quality documentation	25		
Requirement Debt	Requirements partially implemented	6	6	1.9
Test Debt	Expensive tests	0	14	4.3
	Lack of tests	3		
	Low coverage	7		
	Flaky test	4		

Table 4.4: Types and Indicators of TD in the Impala issues

Once again, design debt(60.1%) is the most common TD in the Impala issues, following with code debt(16.1%) and documentation debt(11.5%). The most common indicators in the Impala are non-optimal decision, low-quality code and low-quality documentaion.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	1	1	3.1
	Using obsolete technology	0		
	Containing outdated or empty files	0		
Build Debt	Under- or over-declared dependencies	8	8	2.5
Code Debt	Complex code	3	79	24.6
	Dead code	13		
	Duplicated code	1		
	Low-quality code	57		
	Multi-thread incorrectness	5		
	Slow algorithm	0		
	Memory leak	0		
Defect Debt	Uncorrected known defects	0	0	0
Design Debt	Non-optimal Decisions	142	142	44.2
Documentation Debt	Outdated documentation	26	66	20.6
	Low-quality documentation	40		
Requirement Debt	Requirements partially implemented	2	2	0.6
Test Debt	Expensive tests	0	23	7.2
	Lack of tests	21		
	Low coverage	1		
	Flaky test	1		

Table 4.5: Types and Indicators of TD in the Hadoop issues

Design debt(44.2%), code debt(24.6%) and documentation debt(20.6%) are the three most popular types in the Hadoop issues while non-optimal decision, low-quality code and low-quality documentation are the three most common indicators.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	21	35	2.6
	Using obsolete technology	6		
	Containing outdated or empty files	8		
Build Debt	Under- or over-declared dependencies	23	23	1.7
Code Debt	Complex code	18	342	25.6
	Dead code	39		
	Duplicated code	8		
	Low-quality code	243		
	Multi-thread incorrectness	15		
	Slow algorithm	16		
	Memory leak	3		
Defect Debt	Uncorrected known defects	12	12	0.9
Design Debt	Non-optimal Decisions	646	646	48.3
Documentation Debt	Outdated documentation	67	177	13.2
	Low-quality documentation	110		
Requirement Debt	Requirements partially implemented	15	15	1.1
Test Debt	Expensive tests	6	88	6.6
	Lack of tests	53		
	Low coverage	12		
	Flaky test	17		

Table 4.6: Types and Indicators of TD in Issues Overall

Overall, design debt(48.3%), code debt(25.6%) and documentation debt(13.2%) are the three most common types while non-optimal decisions, low-quality code, and low-quality documentation are the three most common indicators.

4.1.2 RQ1.2: What are the different types of TD identified in commit messages?

4.1.2.1 Non-TD commit messages

Similar to RQ1.1, before listing all different types of TD found in commit messages, examples for Non-TD types were listed below to show the reader that in three different scenarios, a commit message would be annotated as a Non-TD. As the definitions and explanations are already given in RQ1.1, only the examples are shown as below.

1. **New features**:

   THRIFT-3637 Implement compact protocol for dart
   This closes #916

2. **Defect**:

   THRIFT-4545: fix haskell build on windows, fix appveyor stale packages

3. **Insufficient description**:

   THRIFT-4416: additional CPAN packaging changes
   Client: perl

4.1.2.2 TD commit messages

The same number of types and indicators were identified in commit messages. For each of them an example is provided below. As the definition and explanation is similar to RQ1.1, only the examples are shown as below.

1. **Architecture Debt**

- *Violation of modularity*:

  CAMEL-4357 Move Main class to package .main as it is not part of the camel API

- *Using obsolete technology*:

  HBASE-10779 Doc hadoop1 deprecated in 0.98 and NOT supported in hbase 1.0

- *Containing outdated or empty Files*:

  THRIFT-2020: Thrift library has some empty files that haven't really been
  deleted
  Client: cpp
  Patch: Ben Craig

2. **Build debt** - *Under- or over-declared dependencies*:

   - Under-declared dependency:

     THRIFT-2602 fix missing dist files
     - add automake 1.13 dependency to configure.ac and doc
     - use serial-tests instead of .NOTPARALLEL (introduced by THRIFT-1829)
     Patch: Roger Meier

   - Over-declared dependency:

     HADOOP-7161. Remove unnecessary oro package from dependency management section. Contributed by Sean Busbey.

3. **Code debt**

   - *Complex code*:

     Custom MetricRegistry bean has become unecessary given CAMEL-9616

   - *Dead code*:

     HBASE-1396 Remove unused sequencefile and mapfile config. from hbase-default.xml – fixed number in CHANGES.txt

   - *Duplicate code*:

     HBASE-5591 ThiftServerRunner.HBaseHandler.toBytes() is identical to Bytes.getBytes() (Scott Chen)

   - *Low-quality code*:

     HBASE-417 Factor TableOperation and subclasses into separate files from HMaster

   - *Multi thread incorrectness*:

     HADOOP-10427. KeyProvider implementations should be thread safe. (tucu)

   - *Slow algorithm*:

     IMPALA-5042: Use a HashSet instead of ArrayList for O(1) look ups

   - *Memory leak*:

     IMPALA-4027:Memory leak with ExprCtxs not free

4. **Defect debt** - *Uncorrected known defects*:

   CAMEL-8029: For the time being need to @Ignore the failing test

5. **Design Debt** - *Non-optimal decision*:

   HBASE-19969: Improve fault tolerance in backup Merge operation

6. **Documentation debt**

   - *Outdated documentation*:

     HBASE-13973 Update documentation for 10070 Phase 2 changes

   - *Low-quality documentation*:

     HADOOP-11786. Fix Javadoc typos in org.apache.hadoop.fs.FileSystem. Contributed by Andras Bokor.

7. **Requirement debt** - *Requirements partially implemented*:

   THRIFT-2590 C++ Visual Studio solution doesn't include Multiplexing support
   Client: C++
   Patch: Jens Geyer, based on a patch proposal from Thomas Lazar

8. **Test debt**

   - *Expensive tests*:

     CAMEL-12104: Reduce unit test time

   - *Lack of tests*:

     CAMEL-4007: Added test for mail file attachments with non ascii names.

   - *Low coverage*:

     IMPALA-8248: Improve Ranger test coverage
     This patch adds increased coverage for Apache Ranger integration. Specifically, tests were added that interact directly with Apache Ranger via the REST API and then assertions were made against Impala to test proper behavior.

   - *Flaky Test*:

     HBASE-10008 TestNamespaceCommands is flakey on jenkins

In total, eight different types and 21 different indicators of TD were found in commit messages. Out of 849 related commit messages (170 from Thrift, 168 from Impala, 153 from Hbase, 90 from Hadoop and 268 from Camel), there were 247 TD items(66 from Thrift, 62 from Impala, 29 from Hbase, 24 from Hadoop, and 66 from Camel) identified. Similar to commit linked issues, the results for each project are listed first for interested readers and an overall result is listed afterwards. These results could be compared with the results from the commit linked issues.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	1	6	9.1
	Using obsolete technology	3		
	Existence of outdated or empty files	1		
	Non-optimal entity relationship	1		
Build Debt	Under- or over-declared dependencies	2	2	3.0
Code Debt	Complex code	0	27	40.9
	Dead code	4		
	Duplicated code	0		
	Low-quality code	22		
	Multi-thread correctness	0		
	Slow algorithm	0		
	Memory leak	1		
Defect Debt	Uncorrected known defects	2	2	3.0
Design Debt	Non-optimal Decisions	18	18	27.2
Documentation Debt	Outdated documentation	1	4	6.0
	Low-quality documentation	3		
Requirement Debt	Requirements partially implemented	4	4	6.0
Test Debt	Expensive tests	0	3	4.5
	Lack of tests	2		
	Low coverage	1		
	Flaky test	0		

Table 4.7: Types and Indicators of TD in Thrift commit messages

As shown in the table, code debt(40.9%), design debt(27.3%), and architecture debt(9.1%) are the three most common types in the Thrift commit messages. Low-quality code, non-optimal decision and dead code/requirements partially implemented are the most popular indicators in the Thrift commit messages. The results are similar to the results from the Thrift issues.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	2	12	18.2
	Using obsolete technology	9		
	Existence of outdated or empty files	1		
	Non-optimal entity relationship	0		
Build Debt	Under- or over-declared dependencies	5	5	7.6
Code Debt	Complex code	1	17	25.8
	Dead code	6		
	Duplicated code	0		
	Low-quality code	10		
	Multi-thread correctness	0		
	Slow algorithm	0		
	Memory leak	0		
Defect Debt	Uncorrected known defects	1	1	1.5
Design Debt	Non-optimal Decisions	11	11	16.7
Documentation Debt	Outdated documentation	1	4	6.0
	Low-quality documentation	3		
Requirement Debt	Requirements partially implemented	7	7	10.6
Test Debt	Expensive tests	1	9	13.6
	Lack of tests	8		
	Low coverage	0		
	Flaky test	0		

Table 4.8: Types and Indicators of TD in Camel commit messages

In the Camel commit messages, code debt(25.8%), architecture debt(18.2%) and design debt(16.7%) are the three most popular types while non-optimal decision, low-quality code, using obsolete technology are the three most common indicators. This result deviates from what was acquired from the Cammel issues.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	0	1	3.4
	Using obsolete technology	1		
	Existence of outdated or empty files	0		
	Non-optimal entity relationship	0		
Build Debt	Under- or over-declared dependencies	1	1	3.4
Code Debt	Complex code	0	9	31.0
	Dead code	2		
	Duplicated code	1		
	Low-quality code	6		
	Multi-thread correctness	0		
	Slow algorithm	0		
	Memory leak	0		
Defect Debt	Uncorrected known defects	0	0	0
Design Debt	Non-optimal Decisions	5	5	17.2
Documentation Debt	Outdated documentation	4	8	27.6
	Low-quality documentation	4		
Requirement Debt	Requirements partially implemented	0	0	0
Test Debt	Expensive tests	0	5	17.2
	Lack of tests	1		
	Low coverage	0		
	Flaky test	4		

Table 4.9: Types and Indicators of TD in Hbase commit messages

In the Hbase commit messages, code debt(31.0%), documentation debt(27.6%), design debt/test debt(17.2%) are the most common types while low-quality code, non-optimal decision and outdated/low-quality documentation/flaky test are the most common indicators. The results are similar to the results from the Hbase issues. Below is a result summary table for the Impala commit messages.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	0	3	4.8
	Using obsolete technology	2		
	Existence of outdated or empty files	0		
	Non-optimal entity relationship	1		
Build Debt	Under- or over-declared dependencies	2	2	3.2
Code Debt	Complex code	0	22	35.5
	Dead code	1		
	Duplicated code	1		
	Low-quality code	16		
	Multi-thread correctness	1		
	Slow algorithm	2		
	Memory leak	1		
Defect Debt	Uncorrected known defects	1	1	1.6
Design Debt	Non-optimal Decisions	14	14	22.6
Documentation Debt	Outdated documentation	6	12	19.4
	Low-quality documentation	6		
Requirement Debt	Requirements partially implemented	2	2	3.2
Test Debt	Expensive tests	0	6	9.7
	Lack of tests	2		
	Low coverage	1		
	Flaky test	3		

Table 4.10: Types and Indicators of TD in Impala commit messages

In the Impala commit messages, the three most common types are: code debt(35.5%), design debt(22.6%) and documentation debt(19.4%). The three most common indicators are low quality code, non-optimal decisions and outdated/low-quality documentation. When comparing these with what was acquired from Impala issues, the results are very similar.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	0	1	4.2
	Using obsolete technology	0		
	Existence of outdated or empty files	0		
	Non-optimal entity relationship	1		
Build Debt	Under- or over-declared dependencies	2	2	8.3
Code Debt	Complex code	0	11	45.8
	Dead code	0		
	Duplicated code	0		
	Low-quality code	9		
	Multi-thread correctness	2		
	Slow algorithm	0		
	Memory leak	0		
Defect Debt	Uncorrected known defects	0	0	0
Design Debt	Non-optimal Decisions	2	2	8.3
Documentation Debt	Outdated documentation	3	8	33.3
	Low-quality documentation	5		
Requirement Debt	Requirements partially implemented	0	0	0
Test Debt	Expensive tests	0	0	0
	Lack of tests	0		
	Low coverage	0		
	Flaky test	0		

Table 4.11: Types and Indicators of TD in Hadoop commit messages

In the Hadoop commit messages, the three most common TD types are: code debt(45.8%), documentation debt(33.3%) and build/design debt(8.3%). The three most common indicators are: low-quality code, low-quality documentation and outdated documentation. The result is comparable to what we got from Hadoop issues. Below is the result summary table for TD types and indicators found in commit messages.

Type	Indicator	#	#	%
Architecture Debt	Violation of modularity	3	23	9.3
	Using obsolete technology	15		
	Existence of outdated or empty files	1		
	Non-optimal entity relationship	4		
Build Debt	Under- or over-declared dependencies	12	12	4.9
Code Debt	Complex code	1	86	34.8
	Dead code	13		
	Duplicated code	2		
	Low-quality code	63		
	Multi-thread correctness	3		
	Slow algorithm	2		
	Memory leak	2		
Defect Debt	Uncorrected known defects	4	4	1.6
Design Debt	Non-optimal Decisions	50	50	20.2
Documentation Debt	Outdated documentation	15	36	14.6
	Low-quality documentation	21		
Requirement Debt	Requirements partially implemented	13	13	5.3
Test Debt	Expensive tests	1	23	9.3
	Lack of tests	13		
	Low coverage	2		
	Flaky test	7		

Table 4.12: Types and Indicators of TD overall commit messages

Overall, in commit messages, code debt(34.8%), design debt(20.2%) and documentation debt(14.6%) are the three most common types. If these are compared to what was acquired from commit linked issues in RQ1.1:

Overall in issues, design debt(48.3%), code debt(25.6%) and documentation debt(13.2%) are the three most common types.

Thus, in both issues and commit messages, the author identified exactly the same three most common TD types, only the percentages and the order have differences.

Comparing indicators, in commit messages, low-quality code, non-optimal decisions and low-quality documentation are the three most common indicators while in issues,

Non-optimal decisions, low-quality code, and low-quality documentation are the three most common indicators in issues.

Hence the three most common indicators are the same in issues as well as in commit messages. Only the order differs.

To sum up the answer to RQ1:

1. In total, eight different types of TD and 21 different indicators were found in both commit massages and commit linked issues. These 8 types are: architecture, build, code, defect, design, documentation, requirement and test debt.
2. Code debt, design debt and documentation debt are the three most common TD types found in both commit massages and commit linked issues. Low-quality code, non-optimal decision and low-quality documentation are the three most common indicators found in both issues and commit messages.

4.2 RQ2: When a TD was detected in a commit message, what is the likelihood of the same TD been found in the commit linked issue?

In RQ1, the author analyzed the issue sections and commit messages separately and then compared the results. In RQ2, the author tried to find the correlation between TD found in commit linked issues and commit messages. For example, in the following commit message:

THRIFT-4545: disable haskell on windows builds until language-specific build issue can be resolved

Defect debt is identified in this commit message as the developer pointed out there was some incorrect implementation deferred to be fixed. In the corresponding commit linked issue, in one of the issue sections, the same TD has been identified:

"I am disabling haskell in windows cmake builds until someone can resolve this."
- [THRIFT-4545:1]

In this case, the same defect TD has been identified in the commit message as well as in the commit linked issue. The author went through all the commit messages with TD identified, and for each commit, all linked issue sections were went through again to see if the same TD has been identified in the commit linked issue. The process was done automatically by a python script written by the author. The following Table 4.13 shows the result.

	# of commit messages with TD	# of match cases	matching rate
Thrift	66	31	46.9%
Impala	62	30	48.4%
Hbase	29	8	27.9%
Hadoop	24	10	41.7%
Camel	66	3	4.5%
In Total	247	82	33.2%

Table 4.13: Matching Rate between Issues and commit messages for All Projects

As could be seen from the table, there is a high conformity rate between TD found in commit messages and commit linked issues. All the Thrift, Impala, Hadoop has more than 40.0% matching rate while overall, the average matching rate is 33.2%.

To sum up the answer to RQ2: When a TD is identified in a commit message, there is a significantly high chance the same TD would be identified in the corresponding commit linked issue. The chance on average is around 33.2% for the five projects selected for this study.

4.3 RQ3: How often does a commit message introduce or resolve TD?

In order to answer this RQ, two extra tags were created during annotation of commit messages: *debt-introduced* and *debt-resolved*. For example, in the following commit message:

> *THRIFT-4545: disable haskell on windows builds until language-specific build issue can be resolved*

This commit message was tagged with *debt-introduced* because there was a new TD, i.e. defect debt, introduced by this commit to the system. In the following commit message:

> *THRIFT-3197: keepAliveTime is hard coded as 60 sec in TThreadPoolServer*
> *Client: java*
> *Patch: Pankaj Kumar*
>
> *Creating ThreadPoolExecutor in TThreadPoolServer, keepAliveTime is hard coded as 60 sec.*

This commit message was tagged with *debt-resolved* because the current commit fixed a low-quality code TD, as there was a hard coded value in the system before. During annotation, the author also found that there were commit messages that were resolving TD but introducing new TD at the same time. For example, in the following commit:

> *THRIFT-166. java: Java tests should be in lib/java/test/*
> *THRIFT-221. java: Make java build classpath more dynamic and configurable*
>
> *This issue moves all the tests from test/java to lib/java/test/src and combines the build files. In addition, rather than continue on with the same busted approach to finding dependent jars for the tests, THRIFT-221 has been implemented, allowing the user to specify a .thrift-build.properties file in their home directory that contains additional classpath entries.*
>
> *As a result of this patch, "make check" does not currently work as it is expected to. This will be resolved in a follow-up commit.*

In this commit, the developer solved some module violation problem. Hence this commit has resolved a violation of modularity TD. However, the new change resulted in some new bug in the system, and the developer deferred to fix it in the future. Hence a new TD, defect debt, has been introduced in this commit. This commit message was tagged with *debt-introduced* and **debt-resolved**. As a result, within 247 commit messages with TD identified, there were 19 commit messages tagged with *debt-introduced*, 225 commit messages were tagged with *debt-resolved*. There were three cases tagged with both *debt-introduced* and **debt-resolved** at the same time.

To sum up the answer to RQ3: when a commit is submitted, there is a significantly high chance that this commit has resolved a TD in the system. For the five projects, out of all commit messages with identified TD, 92.3% of commit messages were resolving TD while 8.9% of commit messages were introducing TD. To notice, there was also a small amount of commit messages that were resolving TD but introducing new TD at the same time.

Chapter 5

Discussion

From the results, couple of discussion points can be brought up.

First of all, the results revealed that the types of TD identified in commit messages correspond to the types of TD found in commit linked issues. Eight different types of TD and 21 indicators were detected in both commit messages and commit linked issues. Comparing the types, in both issues and commit messages, the author identified exactly the same three most common TD types. They are code debt, design debt and documentation debt. Comparing the indicators, the three most common indicators are the same in both issues and commit messages as well. They are low-quality code, non-optimal decision and low-quality documentation. This overlapping is expected, as in software development projects, developers use commits to address the issues. This result can be also compared with results from studying TD in other sources, for example source code and source code comments. It turns out that more types of TD are identified in commit messages and commit linked issues. This shows that developers tend to explicitly or implicitly discuss about the problems and disadvantages of the current implementation or design through issues or commit messages. Another reason for this is that these problems and disadvantages become more clear through code reviews. Code reviews are reflected through issue tracker systems. Hence, it is reasonable to argue that commit message and commit linked issues can reflect TD more than other sources.

Secondly, the results revealed that there are 33.2% cases that the same TD is addressed in both commit messages and commit linked issues. The percentage is significantly high but can be improved more. Hence, during a software development lify-cycle, for greater consistency and formality, when a TD is addressed in an issue, developers could address the same TD in the corresponding commit messages. Also, as the developers become more aware of the concept of TD, TD can be created as an issue type in the issue tracking system to communicate TD explicitly and to prevent any further TD accumulation. This would bring extra work in the beginning of the development but would make the development process faster and more efficient in the long run.

Lastly, around 92.3% commit messages are found resolving existing TD in the system. This proves that most of TD are repaid. However, there are also 8.9% of commit messages are found incurring TD in the system. The author suggests that when a commit message is incurring TD, it should be reported in the issue tracker. This would raise attentions from other developers and accelerate the repayment of these debt items.

Chapter 6

Threats to Validity

Validity refers to the degree of accuracy to which a study reflects or assesses the specific concept that the researcher is attempting to measure (Pelham & Blanton, 2013). There are two major threats to the validity for this research : threats to internal validity and threats to external validity. This chapter will discuss both of them in detail.

6.1 Internal validity

Internal validity reflects the accuracy of the results of this research. For this study, threats to internal validity can be caused by the manual process of tagging and analysis of the data, because this process is highly subject to personal bias and subjectivity. To counter the threats, first of all, the project data selection and extraction phase was done by the supervisor instead of the researcher. Thus, during the manual analysis phase, the researcher did not have any prior domain knowledge about the data. In this way, when the researcher did the tagging and classification, decision-making would be less biased. Also, as the result highly depends on the data, multiple projects were selected for this study and random sampling was done to obtain the data-set that needs to be analyzed. Lastly, during the manual tagging and classification phase, when the commit message or issue is ambiguous or not clear to classify, the author always discussed the case with one of the supervisors, Y. Li, to try to reach an agreement. If the agreement cannot be reached, Y. Li would discuss the case with the second supervisor, M. Soliman. In this way, some discrepancies can be avoided.

6.2 External validity

External validity reflects the generalisability of the results of this research study. For this study, 5 open source projects were selected. They all use Git commit and JIRA issue tracking system. The findings have shown that the results among these projects are similar and comparable. Hence, the results from this study can be generalized to any similar size open source project which uses Git and JIRA. However, no further generalization can be claimed, especially for projects which do not use commits as version control or issue trackers to record issues.

Chapter 7

Conclusion

The term TD is being used in software development community nowadays by the developers and researchers. It refers to the shortcut decision-making in the software development life-cycle, which is normally immediate solution and easy approach in the short run, but causes negative impacts and greater maintenance efforts in the long run. In general, excess amount of TD would harass and impede the development process and cause extra work for the developers. As the size and complexity of the project grows, excess amount of TD would even make it impossible to maintain or continue the project anymore. A number of studies have focused on the identification and detection of TD in context of literature, source code, source code comment etc. This study has focused on mining TD in commit messages and commit linked issues.

For this study, 5 open source Apache projects were selected. They are Thrift, Camel, Hbase, Impala and Hadoop. All these projects use Git commit as their version control system, and JIRA issue tracking system to record their issues. They are all of high quality and medium to large size, with significant amount of codes, comments, contributors etc. All these projects are still active as the contributors are still constantly updating the projects. Hence, these projects are good indicators to study and understand the concept of TD in commit messages and commit linked issues in the context of open source projects. From these projects, 1,000 commit linked issues and 849 corresponding commit messages were extracted and analyzed. A refined classification method from an existing framework (Alves et al., 2014) (Y. Li et al., 2020) was used to annotate and classify the data.

As the result, 8 different types of TD were detected in both commit linked issues and commit messages. They are architecture, build, code, defect, design, documentation, requirement,and test debt. For each type of TD, there might be one or more causes of it, which was referred as indicator in this paper. There are 21 indicators found in total. For each of them, a concrete example was provided in chapter 4. Furthermore, the author found that, in both commit messages and issues, the 3 most common TD types are code debt, design debt and documentation debt. The 3 most common indicators are low quality code, non-optimal decision and low quality documentation. There is also a correlation between TD found in issues and commit messages, *i.e.* When a TD is identified in a commit message, there is 33.2% probability that the same TD would be found in the commit linked issue. Also, 92.3% of commits with identified TD were found resolving existing TD in the system and 8.9% were found introducing new TD in the system. There were also small amount of commits that were resolving TD but introducing new TD at the same time.

Based on the results, the findings from commit messages correspond to the findings from commit lined issues. The software development community nowadays follows a good work-

flow, pays attention to the concept of TD and addresses TD in the commit messages and issues. This paper can encourage and motivate future research in studying TD in commit messages and commit linked issues. The result can be compared with larger data-set or other types of selected projects. Also, the author hopes that this paper can raise more awareness of the concept of TD among developers during software development to help to achieve a more efficient and faster development process.

References

Alves, N. S. R., Ribeiro, L. F., Caires, V., Mendes, T. S., & Spinola, R. O. (2014). Towards an ontology of terms on technical debt. *IEEE international workshop on managing technical debt*(6).

Beams, C. (2014). *How to write a git commit message.* `https://chris.beams.io/posts/git-commit/`.

Bellomo, S., Nord, R. L., Ozkaya, I., & Popeck, M. (2016). Got technical debt? surfacing elusive technical debt in issue trackers. *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories*, 327-338.

Chacon, S., & Straub, B. (2014). *Pro Git.*

Dai, K., & Kruchten, P. (2017). Detecting technical debt through issue trackers. *5th International Workshop on Quantitative Approaches to Software Quality*, 59-65.

da S Maldonado, E., & Shihab, E. (2015). Detecting and quantifying different types of self-admitted technical debt. *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD) Bremen, Germany.*

Kusumo, D. S., Sabariah, M. K., & Wiharja, K. R. S. (2017). A goal question metric (gqm) approach for evaluating interaction design patterns in drawing games for preschool children. *Jurnal Ilmu Komputer Dan Informasi.*

Li, Y., Soliman, M., & Avgeriou, P. (2020). Identification and remediation of self-admitted technical debt in issue trackers. In *2020 46th euromicro conference on software engineering and advanced applications (seaa).* (unpublished)

Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *The Journal of Systems and Software*(101), 193-220.

McConnell, S. (2007). *Technical debt. 10x software development.* `http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx`.

Nakayama, H., Kubo, T., Kamura, J., Taniguchi, Y., & Liang, X. (2018). *doccano: Text annotation tool for human.* Retrieved from `https://github.com/doccano/doccano` (Software available from https://github.com/doccano/doccano)

Pelham, B. W., & Blanton, H. (2013). *Conducting research in psychology : measuring the weight of smoke.* Wadsworth, Cengage Learning.

Potdar, A., & Shihab, E. (2014). An exploratory study on self-admitted technical debt. *IEEE International Conference on Software Maintenance and Evolution*, 91-100.

Rosen, C., Grawi, B., & Shihab, E. (2015). Commit guru: Analytics and risk prediction of software commits. *ESEC/FSE.*

Zazworka, N., Shaw, M. A., Shull, F., & Seaman, C. (2011). Investigating the impact of

design debt on software quality. *2nd International Workshop on Managing Technical Debt, MTD*, 17-23.