AN EXPLORATORY STUDY ON TECHNICAL DEBT INTRODUCTION AND REPAYMENT IN THE GOOGLE ISSUE TRACKER

BACHELORS THESIS
FACULTY OF SCIENCE AND ENGINEERING

Author: Dammes de Zoeten
Primary supervisor: Paris Avgeriou
Secondary supervisor: Mohamed Soliman
Daily superisor: Yikun Li

July 2020

ABSTRACT

Technical debt (TD) is an allegorical term used to describe shortcuts taken by software developers during the development of projects, either deliberately or unintentionally, to achieve a short-term goal. Unlike bugs, TD issues can take a while to make themselves present, which can result in major problems in dealing with them as the rest of a project can be heavily intertwined.

In this paper, a random sample of 1200 issues were taken from two of Googles issue tracker projects, namely, Chromium and Gerrit. These 1200 issues were analysed for different criteria, to help determine the nature of their TD.

Within these issues 8 different types of TD were found: architecture, build, code, defect, design, documentation, requirement, and test Debt. Of these issues, the 3 most common types were code, design and defect debt.

Of the issues that contained TD, half was created as a result of reporting technical debt and the other half was found later on whilst an issue was being resolved.

About 69.6% of issues with TD were resolved by developers which further strengthened the idea that developers are aware of the importance of resolving TD.

On average, it takes projects in Google's issue tracker, 5559.4 hours or 234 days to resolve TD in issues. The majority of developers that resolved the TD in an issue was neither the person that first reported the issue or the developer that identified the TD in the issue.

ACKNOWLEDGMENTS

I would like to thank my supervisors Paris Avgeriou, Mohamed Soliman and Yikun Li for organising this project and giving me the opportunity to work on this project. I want to express special gratitude to Yikun Li for your continuous support and guidance through out the project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

The term Technical debt was first introduced by Ward Cunningham in 1992 to try and describe the short cuts used by developers, either deliberately or unintentionally, taken in software intensive projects. Technical debt was created to try and address the impact that these "quick fixes" can have on the both the maintainability and evolvability of a project [12]. As opposed to say bugs, which are visible to the user, technical debt issues can take a while before they become apparent and problematic.

There is fine balance between permitting a certain amount of technical debt to allow the progress of a project versus an excessive amount of technical debt which can become problematic later on. This build up can become very challenging to deal with as the project has evolved over time and possibly cause the collapse of a project entirely. Almost all projects will incur technical debt at some stage of their development due to some strategic decision that was made to help with the development of a program or project. Due to the dependencies that modules or components can have on these "quick fixes" it can become extremely challenging beyond a certain point to deal with these issues. Therefore it is vital that these issues be resolved regularly when they are still manageable.

Technical debt can occur at almost any stage throughout a project. Quantifying technical debt and knowing how to deal with it are primary focuses in the study of technical debt. As with any relatively new field, the boundaries that define technical debt are ever changing along with the methods and approaches used to deal with it. As technical debt can occur at any stage of the development life cycle, from project demands, to architectural design, implementation, etc. there are already approaches that exist to detect technical debt in any of these areas via the use of analytic tests on source codes. Another key area in the research of technical debt is the study of these analytic reports with the aim of trying to further understand where technical debt occurs and why.

This study will focus on both the introduction and repayment of technical debt in Google issue trackers. A tool was built to collect issue data from Monorail, Googles issue tracker. The types and indicators of technical debt in the issues were annotated and studied to see how developers identify and resolve technical debt.

2

BACKGROUND INFORMATION

This chapter aims to introduce the reader to the origins of how technical debt came about and some of the background information regarding technical debt. There is also a table at the end of the section with that should help introduce the reader to some of the more relevant terminology, definitions and ideas.

The term technical debt was first introduced by Ward Cunningham in 1992 in a board meeting with non-technical people [3]. Cunningham used the idea of financial debt as an analogy for describing this new term. This allowed the members of the meeting to have a clear understanding as they were already familiar with the idea of financial debt.f

Cunningham explains "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise." [3]

Similar to monetary debt, if technical debt is not repaid in time, it can garner "interest", which in turn makes it harder to pay off, i.e implementing changes can be become more difficult.

Technical debt can also be described in the following way: Technical debt is when software development tasks are put aside to make way for higher priority tasks. The tasks form a foundation and support the project in its entirety. As with any structure, if its foundation is poor, it becomes harder and harder to build upon as the structure keeps developing. Eventually, the possibility arises of needing to demolish the entire structure and rebuilding from the ground up as the foundation can no longer support the project.

2.1 QUANTIFYING TECHNICAL DEBT

Since Ward Cunningham first introduced the idea of technical debt, several key developments have been introduced by academics from around the world. Several studies have been conducted to give technical debt a theoretical framework to be universally accepted and adopted. There are three main studies conducted by Krutchen [13], Fowler[5] and McConnell[9] that have made the foundation for both defining and

quantifying technical debt. The following sub section aims to more explicitly state and outline the ways in which technical debt is quantified.

2.1.1 *Krutchen's Technical Debt Landscape*

Philippe Krutchen first noted that the majority of technical debt is only observed by software developers trying to maintain or update systems. Krutchen distinguished between visible and non-visible technical debt, suggesting that the scope of technical debt should encompass just what is visible to the customer. Krutchen introduced the idea of technical debt arising with the evolvement of a product. This can happen when a program simply out grows what it was initially intended for and an overhaul from the ground up is required. Krutchen also argued that issues with Architectural design and Technological gaps require more attention than code-level technical debt as these cannot be detected by technical debt tools.

2.1.2 *Fowler's Technical Debt Quadrant*

Martin Fowler quantified technical debt based on two main factors, poorly written code and badly designed programs, which Fowler termed reckless and prudent respectively. Fowler went on to further break down these groups into deliberate and non-deliberate. This then gives rise to Fowlers technical debt quadrant. For example, well designed code that was quickly written would fall under Deliberate Reckless code. Fowlers motivation for this system of classification is that these types of technical debt are more taxing to pay off.

2.1.3 *McConnell's Technical Debt Taxonomy*

Steve McConnell broke down the idea of technical debt into the following categories shown below in Figure 1. McConnell started categorizing technical debt into either Intentional or Unintentional. Unintentional technical debt can be summarised as anything ranging from substandard code or poor design. McConnell focused primarily at looking at forms of Intentional technical debt.

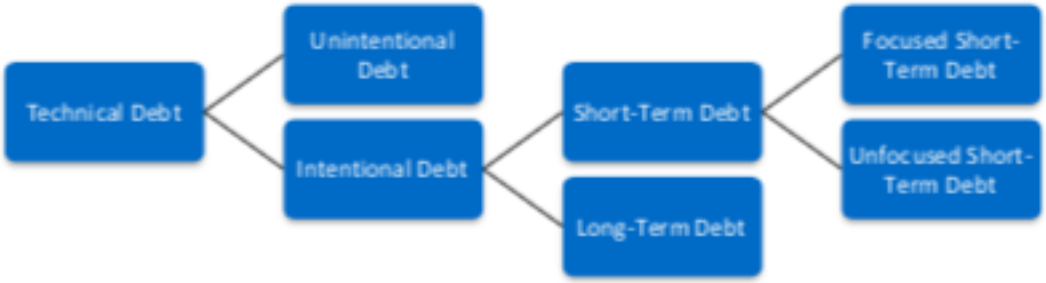


Figure 1: McConnell's Technical Debt Taxonomy

Intentional technical debt is made up of two forms, long term and short term. Long term technical debt is strategically taken on and can last for several years, for example, designing an application to initially only run on Windows and not Mac. Short term technical debt is a strategic choice made by developers which will be quickly payed off. Short term technical debt consists of focused technical debt and unfocused technical debt. Focused technical debt is anything that can be easily tracked and monitored, such as quickly implementing an inefficient algorithm for the sake of completing a project quickly. Unfocused technical debt refers to more difficult to manage issues such as omitting a naming convention for classes for the sake of time. McConnell argued that short term technical debt should be avoided as it can build up very quickly.

Presently, after having established a theoretical framework for technical debt, some of the terms and definitions will be introduced surrounding technical debt will be introduced. It is not required to memorize the following information, it should simply be treated as a point of reference to aid with reading of this paper..

2.2 TECHNICAL DEBT NOMENCLATURE

In practice, technical debt is quite hard to pinpoint in software development. It can occur in many different ways and stages throughout the development and is often challenging to identify. Given that technical debt is still a relatively new research area, there does not yet exist a universally agreed upon system for quantifying and determining technical debt. With this in mind, the following table aims to introduce the reader into the relevant terms and definitions which will be used as metrics for quantifying the data further on in this paper using the information from Alves et al. [6] [11].

See Table 1 Below.

The information given in the aforementioned section(s) will be used throughout the rest of this paper extensively. The terms and definitions will be used fluidly.

Table 1: Definitions of Indicators of Different Types of Technical Debt in Issue Trackers.

Type	Indicator	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.
	UI code	Bad UI code
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.

2.3 ISSUE TRACKERS

An issue tracker is a software system that provides a platform for developers to manage and maintain issues that occur in the development process of software projects. An issue is a unit of work to accomplish and improve in the software development process, an important problem or topic to be discussed. Issues can take many forms ranging from software bugs to feature requests, tasks or the reporting of technical debt. Once a developer reports an issue on the issue tracker, the issue is then archived. At this stage, other developers will then be able to comment and collaborate on resolving the issue.

3

RELATED WORKS

This chapter aims to summarize some related works that have explored the study and understanding of technical debt. There findings have helped in both defining what technical debt is and also analysing it in different areas of software development. This section will focus on detecting technical debt, technical debt in issue trackers and how to manage technical debt.

3.1 DETECTING TECHNICAL DEBT

Several studies have already used static source code analysis techniques to identify code-level technical debt and poor programming (e.g no comments). Marinescu first put forward a metric-based detection system. This was aimed at engineers "to directly localize classes or methods affected by the violation of object-oriented design principles and validated the approach on multiple large industrial case studies." [7]. Munro et al. further developed Marinescu's work by establishing some new metrics to quantify the data and provided justifications for why certain metrics should or should not be present. Munro also tested this updated version of Merinescu's work by identifying two kinds of code smells (lazy class and temporary field) in two case studies [8]. Brondum et al. focused on detecting architectural technical debt by trying to model visual aides based on the analysis of structural code [1]. Li et al. proposed the use of two modularity metrics, Index of Package Changing Impact (ICPI) and Index of Package Goal Focus (IPGF), as indicators of architecture technical debt [18]. Further, they proposed an architecture technical debt identification approach based on architecture decisions and change scenarios [19].

In Ke Dais and Philippe Kruchtens [4] paper the project was split into 5 main stages. Firstly, issue data was exported to allow for easier processing later on. Then the issues were manually categorized into one or more of the relevant types. Following this, key words and phrases were chosen and extracted based on there suspected relevance to technical debt. These words indicate defects or design limitation such as inconsistent UI style, unreasonable design, etc. [4]. Then key features were extracted based on text classification of the previous step. Finally a classifier was established in order to quantify the augmented data. Dai and Kructhen analysed only specific parts of an issue where as this paper will focus on analysing all the comments in an issue.

3.2 TECHNICAL DEBT IN ISSUE TRACKERS

Researchers focused on Technical Debt have the choice of using different Issue trackers, e.g Jira, Clickup. Issue trackers are frequently used in open source projects. They play an important role in facilitating software development teams to manage development and maintenance activities and thus promoting the success of software projects. Some researches have focused on mining issue tracking databases to retrieve valuable information for improved definition, development management, quality evaluation, predictive models, etc.

Runeson et al. developed a prototype tool which detects duplicate defect reports in issue tracking systems using NLP techniques, evaluated the identification capabilities of this approach in a case study and concluded that about 2/3 of the duplicates can possibly be found using this approach [14].

Other work focused on concerned aspects of software quality attributes, e.g security. Cois et al. proposed an approach to detecting security-related text snippets in issue tracking systems using NLP and machine learning techniques [2]

Everton et al. concluded on their paper "Detecting and Quantifying Different Types of Self-Admitted Technical Debt" the following data. Self-admitted technical debt comments were primarily made up of design debt (42% to 84%) and requirement debt (5% to 45%) across their projects. The remaining types of debt have relatively low frequency, comprising of less than 10% of the total quantified debt [15].

Avgeriou et al. focused on self admitted technical in the issue tracker Jira in their article "Identification and Remediation of Self-Admitted Technical Debt in Issue Trackers" [12]. In their findings they found that the issue tracker Jira contained 8 types of technical debt reported: architecture, build, code, defect, design, documentation, requirement, and test debt. In Jira the majority of technical debt was paid off by primarily those who identified or created it. They also found that the median time and average time spent on technical debt repayment was 25.0 and 872.3 hours respectively.

Bellemo et al. gathered the following data with issue trackers Stephany Bellomo and Popec[16] see 2. N.B: Project A and Project B are representatives from the Issue tracker Connect.

Table 2: Issue Tracker Data

Issue Tracker	No. of Issues	No. Of times key words found	Date first occurred
Connect	5,186 since July 2009	15	Jan. 2012
Project A	86	0	N/A
Project B	193	0	N/A
Chromium	>390,000 since Sept. 2008	56	Oct. 2008

As can be seen from the table 2, Key words come up a relatively small amount of time. This in part led Bellemo et al. to the conclusion that "Our data and analysis weakly support that issues where developers discuss certain classes of changes such as refactoring and cleanup are more likely to contain references to accumulation of technical debt." and that "Technical debt conceptually is about conscious design trade-offs. However, the majority of technical debt that developers deal with is a consequence of unintentional design choices. Issue trackers carry information that can assist in uncovering the hidden technical debt." Defining any metric to try and establish a framework for quantifying technical debt can be a tricky and challenging trade-off Stephany Bellomo and Popec[16].

3.3 MANAGING TECHNICAL DEBT

Managing Technical Debt is of course one of the most crucial stages in this process. After data has been collected on technical debt, it can be studied to see in which ways data collection could be improved. Data collection can also help in defining what technical debt is and its scope, i.e, issues rarely or never occurring could be possibly be removed as a metric for defining technical debt.

Bellomo et al. hypothesized that of the issues there tracker provided, the 51 technical debt issues would take longer to solve than the 656 non-technical debt issues. The average amount of days an issues was open for was determined using mean plots. Each day was calculated by subtracting the closed date from the open date. All of the issues in each set of data was split up into either technical debt or non-technical debt. Subgroupings of 100 days (e.g 1 to 100, 101 to 200, etc.) were created. The average of these subgroupings were taking and plotted against the results on a mean plot line charts. These charts enabled easier comparison of the technical and non-technical data sets for each project. Finally the charts were analyzed and it was observed that there

was no consistency or pattern in average days open [16].

4 METHODOLOGY

This chapter aims to outline how to study was conducted by explaining the main steps that were taken during the project. This research project was constructed around answering 5 research questions. These questions are outlined in full below along with further explanations to indicate why they were chosen.

4.1 CASE STUDY DESIGN

The goal of this study, formulated according to the Goal-Question-Metric [6] 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 approached by breaking the goal up into 5 research questions, denoted *RQ*.

- *RQ1: What are the types of technical debt in Google issue trackers?*

  Knowing what types of technical debt can be identified within issue trackers is vital in determining the possible limitations of using issue trackers to analyse technical debt. It is possible to determine which types of technical debt affect Google issue trackers and whether or not specific types of technical debt occur more frequently than others. In determining the types of technical debt that occur in Google issue trackers, all comments in an issue were analysed individually.

- *RQ2: When do developers identify and report technical debt?*

  It is worth noting that issues can be created as a consequence of resolving technical debt but technical debt can also be created as a result of solving issues. It is important to understand whether or not issues are created in order to resolve technical debt or is technical debt is created/acquired as a result of fixing an issue. Understanding when developers identify technical debt in issues can help researchers to more accurately identify the source of technical debt.

- *RQ3: How much of technical debt is resolved?*

  Understanding how much technical debt is resolved can help both software developers and managers in measuring the severity of the technical debt. It also allows an insight to be had of the attitudes of developers towards technical debt. Furthermore it can show when developers are not resolving debt frequently and helps in describing the build up of debt. This is important as a build of debt over

an extended period of time can become problematic and challenging to repay. It also highlights specific developers who have a thorough understanding of debt and manage their own personal technical debt regularly.

- *RQ4: How long does it take on average to resolve technical debt?*

  The length of time it takes to solve technical debt is a good indicator for software developers and managers to effectively plan and manage future issues of technical debt. It helps managers to create a time frame needed to resolve technical debt.

- *RQ5: Who resolves technical debt?*

  Knowing who resolves technical debt will allow a clear understanding to be had in who needs the tools and support to resolve the technical debt more efficiently. Is the person admitting the debt resolving it or is someone else resolving the debt? It is possible that when trying to solve debt admitted by someone else, more information than just the comment itself may be need to resolve the debt. How diverse is the group admitting technical debt? Is it always the same people? Is there a need for change in the structure of development and management of a project?

4.2 METHOD OF APPROACH

1. *Data Source*

   The Google issue tracker Monorail was used for data collection as it contains the majority of Google's mature public source projects. Monorail has been in use since 2013 by Google developers. The projects Chromium and Gerrit were used for this research as both projects have been in development since 2008. Chromium is a web browser that google chrome is built upon and Gerrit is a code review and project management tool for Git based projects. Both of these projects use Monorail as their issue tracker. table 3 shows some details of the Chromium and Gerrit.

Table 3: Project Tracker Data

Project	**No. of Issues**	**Date first occurred**
Gerrit	13,134	2008
Chromium	1,104,611	2008

2. *Filtering Issues*

   This paper contains only data from closed issues, so all open issues were filtered out. This was done in order to avoid conflicts in answering some of the research questions. Open issues can still receive technical debt in future comments which then also need to be resolved. These situations would prevent research questions 3, 4 and 5 from being properly answered. With the closed issue filter applied, the number of issues left to analyse in the two projects were:

Table 4: Project Closed Issue Tracker Data

Project	**No. of Closed Issues**
Gerrit	8,213
Chromium	96,891

3. *Issue Data Collection*

   The collection of data from the issue tracker Monorail was done with a web scrapper tool that was developed in python as the API (Application Program Interface) of monorail is deprecated. Analysing every closed issue from both projects would be too time consuming so a sample size of 600 issues was taken from each project. The web scrapper tool extracted data from 600 randomly chosen closed issues from each project.

4. *Manual annotation of Issues*

   After the relevant data was gathered from these randomly chosen closed issues they were imported into Doccano an online annotation tool. In Doccano each issue was broken up under the following categories: issues title, description and issues comments. Each of these components was then analyzed individually and manually annotated with either the correct label stating which type of technical debt it has or it was labelled debt free if no technical debt was present in the issue. The total numbers of titles, descriptions, and comments of the 1200 issues added up to 6644 components to annotate. Throughout the process of annotation, discussions were held periodically about issues that were annotated in a contradictory manner to overcome the possibility of human error occurring. After completing the annotations, a generated data set was produced containing the technical debt annotations for each of the collected issues.

5. *Manual Analysis of Technical Debt Issues*

   With the generated data set of annotated issues, the issues containing technical debt were manually analysed on Monorail to determine whether or not the issues were resolved. Each issue was analysed using the issue's comments to check if the technical debt had been resolved. If the technical debt in the issue was resolved the owner and date of the resolving comment in the issue is recorded.

6. *Data Analysis*

   Quantitative analysis was conducted on each of the data sets collected. This was done with the aim of trying to understand when and why technical debt occurs, specifically in the projects studied from Google's issue tracker Monorail. Applying the correct statistical procedures on each sample set of data for each project, data was computed and visualised with various charts to accurately answer the research questions with quantified evidence.

5 IMPLEMENTATION

The following chapter aims outline how the study was carried out with in particular, a detailed outline of how the data was gathered, collected, and annotated. Justifications and background to implementation design for the web scrapping tool is also given in this chapter.

5.1 WEB SCRAPPING TOOL

A web scrapping tool was designed and implemented to collect the data required for the analysis of the project. The API of Google's issue tracker, Monorail, was not used for this project as it was deprecated at the time and the newer version was not open to the public at the time this project was conducted.

5.1.1 *Design*

The web scrapper tool was programmed in python due to the ease of implementation. The libraries required for this extraction and manipulation of data where also readily available which further motivated the choice of using python.
The main obstacle in scrapping Monorail's website is that the website is a dynamic website that heavily uses shadow DOM.

"Shadow DOM is a web component that allows encapsulation. This allows the web page to be able to keep the markup structure, style, and behavior hidden and separate from other code on the page." [10]

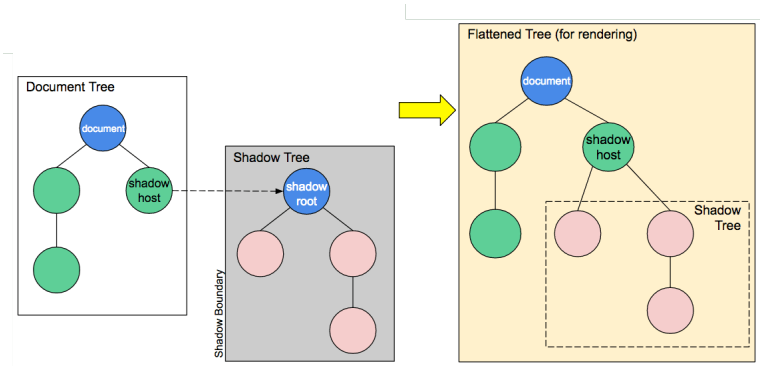
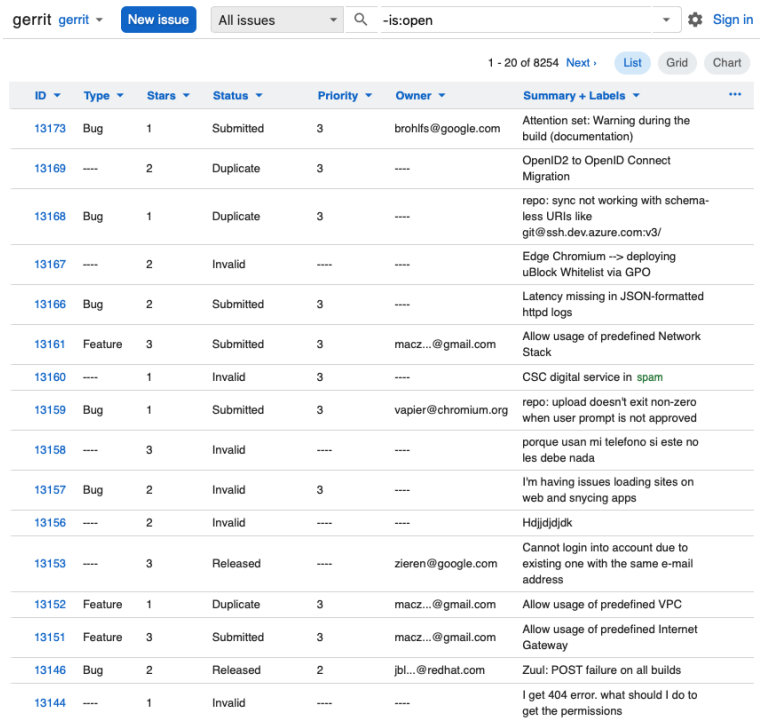


Figure 2: Shadow DOM Structure

"Shadow DOM allows hidden DOM trees to be attached to elements in the regular DOM tree — this shadow DOM tree starts with a shadow root, underneath which can be attached to any elements you want, in the same way as the normal DOM."[10]

To reveal these hidden shadow DOMs, the module *selenium_utilities.py* was created. This was done by rendering these shadow DOMs independently. This had to be done multiple times as Monorails webpages used multiple shadow DOMs on one page and many nested shadow DOMs which all had to be rendered independently.

The *MonorailScraper* class would first create a list of all the ids of closed issues of the chosen projects. This list was saved as json file. This was done by scrapping the webpage that contained the list of issues. This then iterated through multiple pages of the lists of closed issues.



ID	Type	Stars	Status	Priority	Owner	Summary + Labels
13173	Bug	1	Submitted	3	brohlfs@google.com	Attention set: Warning during the build (documentation)
13169	----	2	Duplicate	3	----	OpenID2 to OpenID Connect Migration
13168	Bug	1	Duplicate	3	----	repo: sync not working with schema-less URIs like git@ssh.dev.azure.com:v3/
13167	----	2	Invalid	----	----	Edge Chromium --> deploying uBlock Whitelist via GPO
13166	Bug	2	Submitted	3	----	Latency missing in JSON-formatted httpd logs
13161	Feature	3	Submitted	3	macz...@gmail.com	Allow usage of predefined Network Stack
13160	----	1	Invalid	3	----	CSC digital service in spam
13159	Bug	1	Submitted	3	vapier@chromium.org	repo: upload doesn't exit non-zero when user prompt is not approved
13158	----	3	Invalid	----	----	porque usan mi telefono si este no les debe nada
13157	Bug	2	Invalid	3	----	I'm having issues loading sites on web and snycing apps
13156	----	2	Invalid	----	----	Hdjjdjdjdk
13153	----	3	Released	----	zieren@google.com	Cannot login into account due to existing one with the same e-mail address
13152	Feature	1	Duplicate	3	macz...@gmail.com	Allow usage of predefined VPC
13151	Feature	3	Submitted	3	macz...@gmail.com	Allow usage of predefined Internet Gateway
13146	Bug	2	Released	2	jbl...@redhat.com	Zuul: POST failure on all builds
13144	----	1	Invalid	----	----	I get 404 error. what should I do to get the permissions

Figure 3: Monorail Issue List Page

An older version of the webpage of the list of issue was used to overcome the hindrance of shadow DOM as they were not used in the older version, as they were in the new version of the webpage.

After the creation and collection of the list of closed issue ids, *MonorailScraper* class would create a random list of 600 closed issues of the projects. The function *scrapeIssueList* would iterate through each issue id to scrape.

The class *IssueScraper* was used to scrape all the relevant data of each issue.

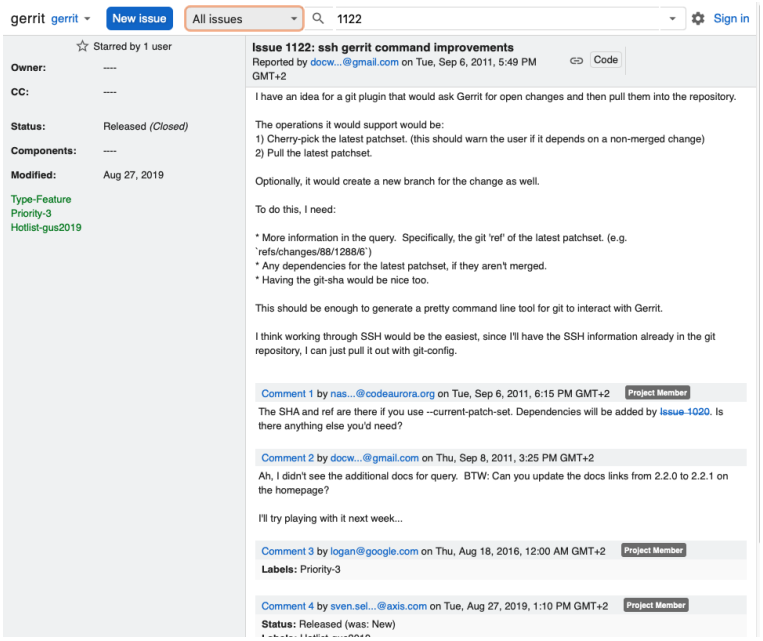


Figure 4: Monorail Issue Page

Each data element collected had to be independently found by first rendering multiple nested shadow roots repeatedly.
The json data below shows all the data that was collect from each issue.

```
{
    "ID": "Issue ID",
    "title": "Title of issue",
    "opened date": "Date issue was opened",
    "closed date": "Date issue was closed",
    "reporter": "The creator of the issue",
    "status": "Status of the issue",
    "description": "description of the issue",
    "comments": [
        {
            "comment": "text of comment",
            "date": "date when comment was made",
            "owner": "owner of comment"
        }
    ]
}
```

Once the program iterated through the whole random list of issues and all the data was collected it was then saved to a json file.

5.1.2 *Evaluation*

Each time a shadow root was rendered a wait time was implemented to give the shadow root time to load the data. Due to this and the need to render multiple nested shadow roots repeatedly, which was needed to find and extract each data element. Therefore, there was a considerate amount of slowdown.

5.2 ANNOTATION

The collected data was imported into, Doccano an online annotation tool. In Doccano each issue was broken up under the following categories: issues title, description and issues comments.

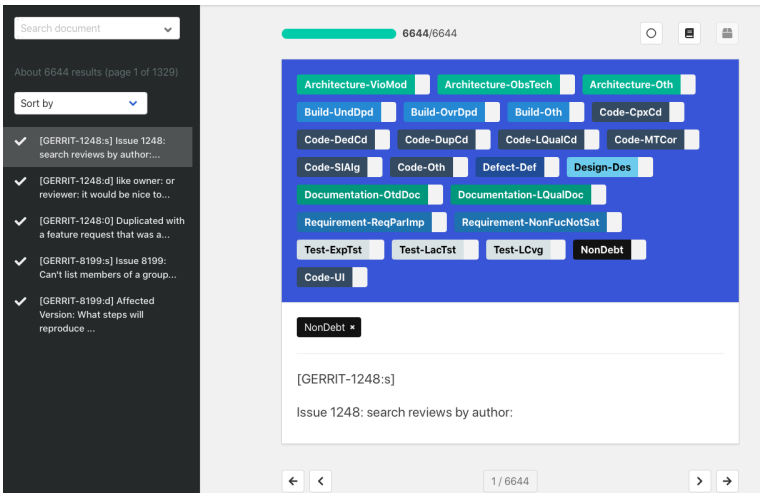


Figure 5: Doccano Page

An Annotation Guideline can be viewed in **Table** 15 of the Appendix. Each of these components was then analyzed individually and manually annotated with either the correct label stating which type of technical debt it has or it was labelled *NonDebt* if no technical debt was present in the issue.

6 RESULTS

This Chapter aims to outline in a clear and concise manner the results gathered. The research questions will be answered in the next section using the results explained in this chapter. It is also worth noting that in certain cases, only the average results across the two projects investigated are displayed here for the sake of ease of reading. The full results for individual projects are however contained in the appendix of this paper in the instances that they are omitted from this chapter.

6.1 TYPES OF TECHNICAL DEBT IN GOOGLE ISSUE TRACKERS

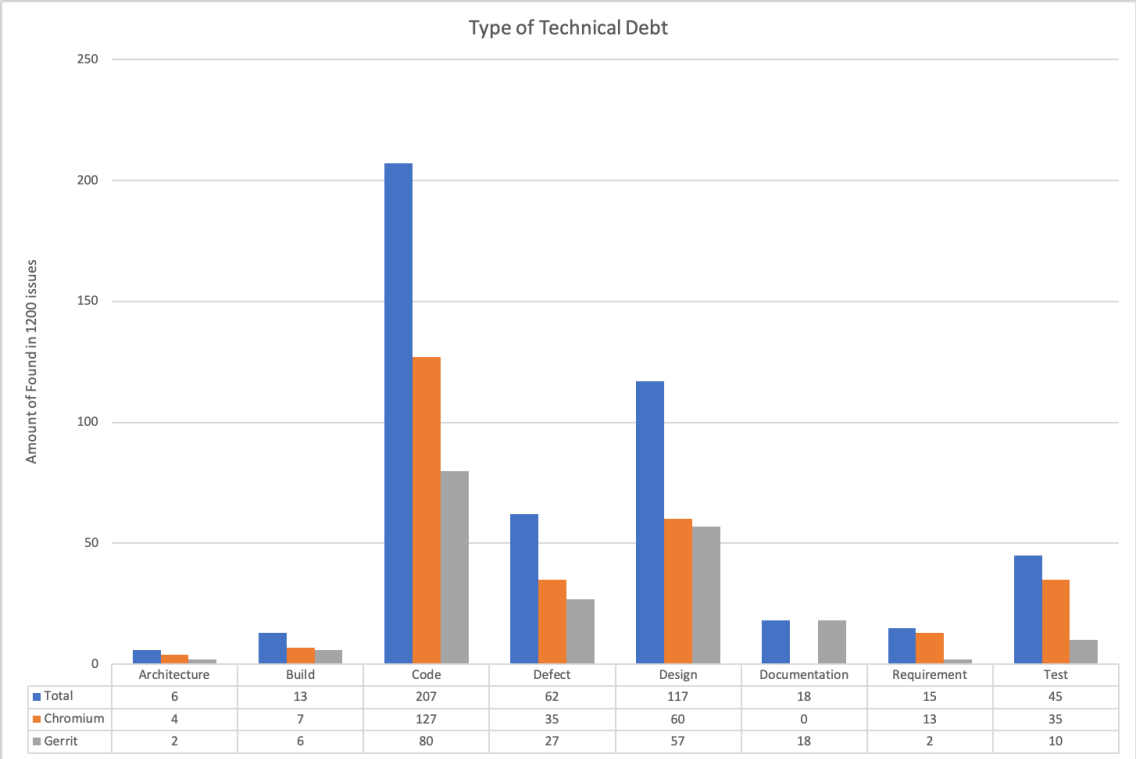
In the Google issue tracker, Monorail, there were 8 different types of technical debt found in the two projects, Chromium and Gerrit and there were 22 different indicators in total for indicating the 8 different types of technical debt.

A list of example issues with the different types of technical debt found:

- **Architecture Debt:** *"Robert, seems you removed support for window.external.AddSearchProvider() on Mac"*-[CHROMIUM-88350]
- **Build Debt:** *"chromium build should not depend on WebKit/WebKit or WebKit/WebKitLibraries"*-[CHROMIUM-4685]
- **Code Debt:** *"Hmmm... did we use a deprecated syntax? If that syntax is illegal(it IS ambiguous) then I'd prefer that it failed rather than worked in some cases..."*-[GERRIT-2507]
- **Defect Debt:** *"We're going to drop GWT after 2.16. It's not worth fixing this on the existing stable branches."*-[GERRIT-2309]
- **Design Debt:** *"Main reason why flush() fixes random things for you is because it creates a timeout before the callback. You could effectively replace it with setTimeout(()=>, 1); in most cases. Which is obviously not a good practice, just to be clear. I think flush() finished debouncers, micro-tasks, and probably data binding propagation. I don't think this is the case here."*-[GERRIT-6524]
- **Documentation Debt:** *"...However. There is an undocumented thing here, we do in fact load $site_path/etc/gitweb_config.perl and allow it to override prior configuration. So I*

*guess I've already done what you asked for, I just didn't document it. *sigh**"-[GERRIT-496]

- **Requirement Debt:** *"The toolbar icons (add / remove / show-in-folder) in the Mac bookmark manager window are just placeholders. I need to replace them with some real ones..."*-[CHROMIUM-32442]
- **Test Debt:** *"Test flakiness on linux due to Omnibox not being fully functional immediately after window creation"*-[CHROMIUM-62783]



	Architecture	Build	Code	Defect	Design	Documentation	Requirement	Test
Total	6	13	207	62	117	18	15	45
Chromium	4	7	127	35	60	0	13	35
Gerrit	2	6	80	27	57	18	2	10

Figure 6: Technical Debt Bar-graph

Table 5: Total Types and Indicators of Technical Debt.

Type	Indicator	No.	No.	%
Architecture debt	Violation of modularity	3	6	1.2
	Using obsolete technology	3		
Build debt	Over-declared dependencies	2	13	2.7
	Under-declared dependencies	2		
	Poor deployment practise	9		
Code debt	Complex Code	9	207	43
	Dead Code	3		
	Duplicated Code	2		
	Low-quality Code	80		
	Multi-thread correctness	5		
	Slow algorithm	7		
	Code UI	98		
	Code other	3		
Defect debt	Uncorrected known defects	62	62	12.8
Design debt	Non-optimal decisions	117	117	24.2
Documentation debt	Low-quality documentation	10	18	3.7
	Outdated documentation	8		
Requirement debt	Requirements partially implemented	8	15	3.1
	Non-functional requirements not being fully satisfied	7		
Test debt	Expensive tests	24	45	9.3
	Lack of tests	13		
	Low coverage	8		

Table 6: Gerrit Types and Indicators of Technical Debt.

Type	Indicator	No.	No.	%
Architecture debt	Violation of modularity	0	2	0.9
	Using obsolete technology	2		
Build debt	Over-declared dependencies	0	6	3
	Under-declared dependencies	2		
	Poor deployment practise	4		
Code debt	Complex Code	0	80	39.7
	Dead Code	0		
	Duplicated Code	2		
	Low-quality Code	35		
	Multi-thread correctness	0		
	Slow algorithm	1		
	Code UI	42		
	Code other	0		
Defect debt	Uncorrected known defects	27	27	13.5
Design debt	Non-optimal decisions	57	57	28.2
Documentation debt	Low-quality documentation	10	18	8.9
	Outdated documentation	8		
Requirement debt	Requirements partially implemented	0	2	0.9
	Non-functional requirements not being fully satisfied	2		
Test debt	Expensive tests	5	10	4.9
	Lack of tests	3		
	Low coverage	2		

Table 7: Chromium Types and Indicators of Technical Debt.

Type	Indicator	No.	No.	%
Architecture debt	Violation of modularity	3	4	1.4
	Using obsolete technology	1		
Build debt	Over-declared dependencies	2	7	2.5
	Under-declared dependencies	0		
	Poor deployment practise	5		
Code debt	Complex Code	9	127	45.1
	Dead Code	3		
	Duplicated Code	0		
	Low-quality Code	45		
	Multi-thread correctness	5		
	Slow algorithm	6		
	Code UI	57		
	Code other	3		
Defect debt	Uncorrected known defects	35	35	12.5
Design debt	Non-optimal decisions	60	60	21.4
Documentation debt	Low-quality documentation	0	0	0
	Outdated documentation	0		
Requirement debt	Requirements partially implemented	8	13	4.6
	Non-functional requirements not being fully satisfied	5		
Test debt	Expensive tests	19	35	12.5
	Lack of tests	10		
	Low coverage	6		

The graph above 6 shows the frequency at which different types of technical debt occurs Chromium, Gerrit and the total. The tables 6 and 7 present the results of different types of technical debt found through their indicators in both projects and 5 shows the combined total different types of technical debt found through their indicators. Table 7 shows that the project Chromium only contained 7 different technical debt types and 19 different indicators for these technical debt. Table 6 shows that the project Gerrit contained all 8 different technical debt types but only 15 different indicators. From table 5 it is observed that three most common types of technical debt are code-debt with 43%, design-debt with 24.2%, and defect debt with 12.8%.

Table 8: Total Number of Types of Technical Debt in Issues.

Type	# Indicator	% Issues
Does not contain technical debt	808	67.3
Contains one type of technical debt	316	26.3
Contains two types of technical debt	63	5.3
Contains three types of technical debt	12	1
Contains five types of technical debt	1	0.1

Table 9: Gerrit Number of Types of Technical Debt in Issues.

Type	# Indicator	% Issues
Does not contain technical debt	426	71
Contains one type of technical debt	150	25
Contains two types of technical debt	20	3.3
Contains three types of technical debt	4	0.7

Table 10: Chromium Numeber of Types of Technical Debt in Issues.

Type	# Indicator	% Issues
Does not contain technical debt	382	63.6
Contains one type of technical debt	166	27.7
Contains two types of technical debt	43	7.2
Contains three types of technical debt	8	1.3
Contains five types of technical debt	1	0.2

Tables 8, 9, and 10 show the number of different types technical debt found in a single issue. An issue can have more than one technical debt instances occurring in its lifetime. From table 8 it is observed that 32.7% of the 1200 issues analysed has technical debt. Of the 600 issues in Gerrit that were analysed, 29% had technical debt and from Chromium's 600 issues, 36.3% had technical debt. Of all the issues with technical debt 19.4% of them had more than one type of technical debt.

6.2 DEVELOPERS IDENTIFYING AND REPORTING TECHNICAL DEBT

Table 11 shows how many issues with technical debt are either *Created* or *Identified*. *Created* means that the issue was created by a developer for the purpose of reporting and resolving technical debt and *Identified* means that the technical debt was identified by a developer later in the process of resolving an issue. Observing the table shows that from the 392 issues with technical debt 54.1% of the issues were created to report technical debt and 45.9% of the issues was technical debt that was identified later in the issue.

Table 11: Technical Debt Reporters.

Project	Created		Identified	
	No.	**%**	**No.**	**%**
Gerrit	84	48.3	90	51.7
Chromium	128	58.7	90	41.3
Total	212	54.1	180	45.9

6.3 AMOUNT OF TECHNICAL DEBT RESOLVED

Table 12 shows the number of issues with technical debt that were resolved and not resolved. In some cases it could not be determined if the issues were resolved or not. In these cases the issues were labelled as Not Determined. Observing the table 12 shows that of the issues that could be determined, 69.6% were resolved. In particular this was 76.8% for Gerrit and 63.7% for Chromium.

Table 12: Amount of Technical Debt that was Repaid.

Project	No. Resolved	No. Not Resolved	No. Not Determined
Gerrit	106	32	36
Chromium	109	62	47
Total	215	94	83

6.4 AVERAGE TIME TO RESOLVE TECHNICAL DEBT

Figure 7 shows the visual distribution for the time in hours it took to resolve the technical debt for the issues in both projects separately and combined. The average time it takes for the two projects to resolve their technical debt is 7246.4 hours for Gerrit and 3918.8 hours for Chromium. To compare the two projects' average time, a Welch Two Sample t-test [17] was preformed on the time samples of the two projects, with the null hypothesis: (that the difference in the average time is equal to 0), gaining the p-value of 0.054. It was calculated that the average time to resolve technical debt for the issue tracker is 5559.4 hours.

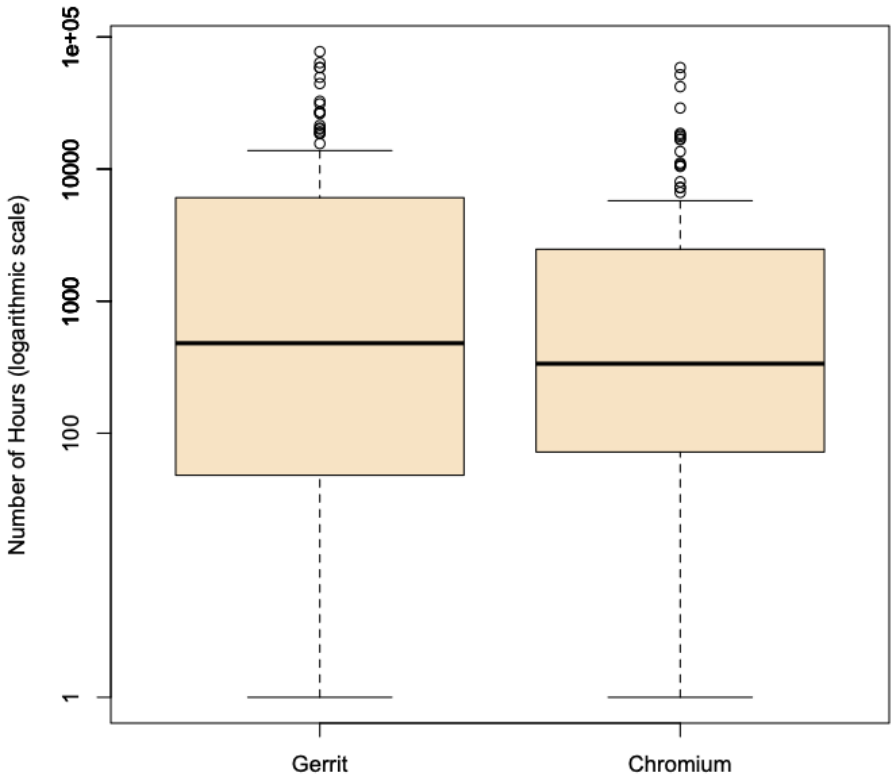


Figure 7: Time to Resole Technical Debt

Table 13: Average Time to Resolve Technical Debt

Project	Mean Time (Hours)	Median Time (Hours)
Gerrit	7246.4	480
Chromium	3918.8	336
Total	5559.405	408

6.5 THE DEVELOPERS THAT RESOLVE TECHNICAL DEBT

Table 14 shows who the developers were that resolved the issues with technical debt. There are three cases: Creator, Identifier, and Other. Creator is when the creator of the issue resolves it. The Identifier is when the person who identifies the issues later in

the project resolves it. Other is when neither the Creator nor the Identifier are the ones who resolve the issue, but rather another developer. The table shows that most of the technical debt is resolved by neither the Creator nor Identifier. From the Table 14 it can be seen that 74% of technical debt issues are resolved by Other, 18.6% of technical debt issues are resolved by the creator of the issue, and 7.4% of technical debt issues are resolved by the identifier of the technical debt.

Table 14: Who Repaid Technical Debt.

Project	**Repaid**	**Repaid by**					
		Creators		**Identifiers**		**Others**	
		No.	**%**	**No.**	**%**	**No.**	**%**
Gerrit	106	16	15.1	5	4.7	85	80.2
Chromium	109	24	22	11	10.1	74	67.9
Total	215	40	18.6	16	7.4	159	74

7 DISCUSSION

This chapter aims to answer the proposed research questions in detail. The results are compared to other papers in order to help give the reader an insight into how the information of this paper correlates with work of a similar nature. Further discussions are made to previous papers to help further support the results of this paper or outline interesting disagreements.

7.1 QUESTION 1: WHAT ARE THE TYPES OF TECHNICAL DEBT IN GOOGLE ISSUE TRACKERS?

It can be seen from the results that the Google issue tracker, Monorail, has 8 different types of technical debt: architecture, build, code, defect, design, documentation, requirement, and test debt. There are 22 different indicators for these types of technical debt. The three most common types of technical debt found in the issue tracker were code, design, and defect debt respectively. In the study[6] it was shown that Jira's three most common types of technical debt were code, documentation, and test debt whereas in the study of S. Maldonado and Shihab[15] they found that design debt was their most common form of technical debt followed by requirement debt. Both Jira and Google's issue trackers have code debt as their most common type of technical debt. Design debt is a major source of technical debt in the two studies cited above and this paper. The other major types of technical debt that occur in Googles issue trackers are not similar to the findings of Everton et al. and to the types that most commonly occur in Jiras issue tracker. Documentation and requirement debt are one of the least common types of technical debt found in this paper. This shows that there are distinct differences in the types of technical debt that are found in different issue trackers.

In comparing the types of technical debt found in Gerrit against Chromium, it can be seen that Chromium had only 7 types of technical debt found compared to Jira which had 8. Chromium did not have any documentation debt unlike Jira, which was the second most common technical debt found in Jira's issue tracker. This clearly shows the importance in these studies as research into the different types of technical debt that occur in issues trackers can highlight many differences between issue trackers. This makes sense as the nature of the projects are different and therefore they do not face the same issues which results in them not producing the same types of technical debt.

7.2 QUESTION2: WHEN DO DEVELOPERS IDENTIFY AND REPORT TECHNICAL DEBT?

Developers use one third of the issues created to identify and report technical debt. This shows that issue tackers are important tools in keeping track of technical debt in projects. In the results section two categories were used to analyse when technical debt is identified and reported by developers. The results show that the data is split evenly between developers reporting technical debt through creating an issue and developers identifying technical debt later in an issue.

7.3 QUESTION 3: HOW MUCH OF TECHNICAL DEBT IS RESOLVED?

It can be seen from the results that most of the technical debt that is identified and reported by developers was resolved. This total came to about 69.6% of the issues in Google's issue tracker that were eligible candidates for determining a result of resolved or not. This is inline with results of the study done on Jira's issue tracker[6], as their result was 71.3% and 72.5% for their two projects. This strengthens the idea that developers are aware of the importance of resolving technical debt.

7.4 QUESTION 4: HOW LONG DOES IT TAKE ON AVERAGE TO RESOLVE TECHNICAL DEBT?

On Average it takes projects in Google's issue tracker, 5559.4 hours to resolve technical debt in issues. For Gerrit this is 7246.4 hours and for Chromium it is 3918.8 hours. It was shown in the Welch Two Sample t-test that there is a significant difference in the average time taken to resolve technical debt for the two projects Comparing the average times it takes to resolve technical debt in Google's issues tracker and Jira's issue tracker, it can clearly be seen that the average time to resolve technical debt is significantly higher in Google's issues tracker, as in the study [6], compared to Jira's average time of repaying technical debt of about 872.3 hours. What is similar though is that in the study [6] it was also observed that there was a significant difference between their average times to resolve in their two projects. This can be due to the fact that these projects have different developers with different technical skill or due to the fact of that different types of technical debt occur in each project. Like in [16] there was no consistency or pattern in average hours it took to resolve technical debt

7.5 QUESTION 5: WHO RESOLVES TECHNICAL DEBT?

From the results it can be seen that technical debt in Google issue trackers are mostly resolved by developers who are neither the reporter nor identifier of the technical debt

in an issue. This comes out to about 74% of the technical debt issues. This could be due to the fact that the developers that report or identify technical debt in an issue rely other developers that are highly skilled in resolving technical debt. The opposite however can be said about Jira's issue tracker, where it was found that only 8.3% of all technical debt is resolved by other developers. This might indicate why the average time taken to resolve technical debt is significantly lower in Jira's issue trackers as they do not wait for other developers.

8 THREATS TO VALIDITY

This chapter aims to explain the nature and types of threats to validity in the undertaking of this research. The information in this paper is subject to errors due to the approaches taken in both methodology and the collection of data. As with any research topic concerning the analysis of data using subjective methods, both the analysis of results and conclusions garnered are affected by threats to validity. The two primary sources of threats to validity concerning this particular paper are internal and external threats to validity which concerns the actions taken by the researcher in regards to the analysis of data and how the results of this particular paper can correspond to topics or applications of a similar nature respectively.

8.1 INTERNAL VALIDITY

Threats to the internal validity of this paper stem from the decisions and measures that were taken in a subjective aspect. Threats to internal validity in this study arises in the manual annotations of comments on issues of technical debt in Google issue trackers. The way in which these comments can be annotated can differ from researcher to researcher, i.e one researcher may label an issue as solved in the case of unclear comments whereas another could label it unresolved. The threats to internal validity can still arise however when the sample taken does not accurately reflect the entire project. This was mitigated by taking a random sample of data.

8.2 EXTERNAL VALIDITY

The understanding of threats to external validity in this paper is important for further research in the study of technical debt as it allows others researchers to see how relevant this paper may be to a similar topic. The data garnered in this paper may not accurately reflect data in other issues trackers. This is because different projects may use different labels and comments for issues than in this paper as each developer or team may have their own unique system for issue classification. Threats to external validity were minimized by focusing on only Googles issue trackers. From the two projects chosen in the issue tracker, Chromium and Gerrit, equal sample sizes were chosen in order to allow for fair comparisons to be made. These projects were also chosen due there longevity, which insured the samples taken would be significantly less likely to be skewed with erratic issues that may occur in newer projects.

9 CONCLUSION

The goal of this study is to analyze issues in Google 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.

In this paper, eight types of technical debt were found in two projects, Chromium and Gerrit, found on Monorail, a Googles issue tracker. There were 22 different indicators that highlighted the presence of one of these forms of technical debt. The types of technical debt that were present were architecture, build, code, defect, design, documentation, requirement, and test debt. From the sample set taken of 1200 issues, 32.7% contained at least one indicator that showed the presence of some form of technical debt. The three most common types of technical debt found in the issue tracker were code, design, and defect debt (with 43%, 24.2%, and 12.8% respectively).

Furthermore, from the technical debt reported, an average 54.1% of issues were reported by their creator across the two projects. The remaining 45.9% of issues were reported by a developer other than the creator of the issue. Of the 392 cases were technical debt was identified across the two projects studied in this paper, 215 cases were resolved, 94 cases were not resolved and 83 of these cases were inconclusive, i.e it was not clear if the technical debt was repaid or not.

The average time it took for technical debt to be resolved for the projects studied in Monorail was 5559.4 hours. It took on average 7246.4 hours for Gerrit to resolve their issues with technical debt and 3918.8 hours for chromium. This significant statistical difference is most likely a result of the different nature of these two projects.

Lastly, across the two projects, of the technical debt that was repaid, 18.6% of it was repaid by the creator of the issue and 7.4% was repaid by the identifier of the technical debt. The remaining 74% of resolved technical debt was resolved by developers who neither created nor identified the debt.

10

FURTHER WORKS

This study can act as good basis for many different research projects in technical debt. The structure created in this paper by outlining how technical debt can be quantified and how the results can be analysed can aid further research even when different sample sets are chosen.

Further works in an exploratory study of technical debt in Google issue trackers can be improved by taking a larger sample size of issues from the projects Chromium and Gerrit. Despite the sample set being randomly selected, the size of the sample set selected still limits how accurately the results of this study can reflect the total spread of issues with technical debt across Chomium and Gerrit. A larger sample set would greatly strengthen the findings of this thesis should this larger data set lead to results that are inline with that of this paper.

As reflection of Googles issue trackers, this study can only be representative of the projects Chromium and Gerrit. Other projects that differ in nature could likely produce different statistical results. Given that 27 different projects use Monorail as their issue tracker, it would be interesting to repeat this study with a larger number of projects. This would allow one to compare and determine with a greater certainty if the findings of this study are a reflection of Googles issue tracker or to the selected projects chosen inside Googles issue tracker.

Lastly, given that all the annotations were deduced on a subjective manner, there will always be discrepancies in how this study will relate to further works. Each study will by nature, have its own system and methodology for labeling issues and some studies will also quantify issues differently than how this study has. A standard guideline or a standard metric for detecting and annotating technical debt in issues would strengthen and enrich future data in further studies in technical debt.

APPENDIX A

A.1 DOCCANO ANNOTATION GUIDELINE

LABEL	TECHNICAL DEBT INDICATOR
Architecture-VioMod	Architecture debt - Violation of modularity
Architecture-ObsTech	Architecture debt - Using obsolete technology
Architecture-Oth	Architecture debt - Others
Build-UndDpd	Build debt - Under-declared dependencies
Build-OvrDpd	Build debt - Over-declared dependencies
Build-Oth	Build debt - Others
Code-CpxCd	Code Debt - Complex code
Code-DedCd	Code Debt - Dead code
Code-DupCd	Code Debt - Duplicated code
Code-LQualCd	Code Debt - Low-quality code
Code-MTCor	Code Debt - Multi-thread correctness
Code-SlAlg	Code Debt - Slow algorithm
Code-UI	Code Debt - Wrong UI code
Code-Oth	Code debt - Others
Defect-Def	Defect debt - Uncorrected known defects
Design-Des	Design debt - Non-optimal decisions
Documentation-OtdDoc	Documentation debt - Outdated documentation
Documentation-LQualDoc	Documentation debt - Low-quality documentation
Requirement-ReqParImp	Requirement debt - Requirement partially implemented
Requirement-NonFucNotSat	Requirement debt - Non-functional requirements not fully satisfied
Test-ExpTst	Test debt - Expensive tests
Test-LacTst	Test debt - Lack of tests
Test-LCvg	Test debt - Low coverage
NonDebt	No technical debt

Table 15: Annotation Guideline.

BIBLIOGRAPHY

[1] John Brondum and Liming Zhu. "Visualising architectural dependencies. In Proceedings of the Third International Workshop on Managing Technical Debt." In: (2012).

[2] C.A. Cois and R. Kazman. "Natural Language Processing to Quantify Security Effort in the Software Development Lifecycle." In: (2015).

[3] Ward Cunningham. "The wycash portforlio management system." In: (1993).

[4] Ke Dai and Philippe Kruchten. "Detecting Technical Debt through Issue Trackers." In: (2016).

[5] Martin Fowler. *Technical debt quadrant*. URL: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html. accessed: 20.04.2020.

[6] Yikun Li, Mohamed Soliman, and Paris Avgeriou. "Identification and Remediation of Self-Admitted Technical Debt in Issue Trackers." In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. unpublished. 2020.

[7] R Marinescu. "strategies: Metrics-based rules for detecting design flaws in Software Maintenance." In: (2004).

[8] R. Marinescu. "M.J. Product metrics for automatic identification of" bad smell"design problems in java source-code." In: (2005).

[9] Steve McConnell. *Managing technical debt. The Construx Whitepaper*. URL: https://www.construx.com/developer-resources/whitepaper-managing-technical-debt. accessed: 20.04.2020.

[10] Chris Mills. *Using shadow DOM*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM. accessed: 03.05.2020.

[11] Vivyane Caires Thiago S. Mendes Rodrigo O. Spínola Nicolli S. R. Alves Leilane F. Ribeiro. "Towards an Ontology on Terms of Technical Debt." In: (2014).

[12] Ipek Ozkaya Paris Avgeriou Philippe Kruchten and Carolyn B. Seaman. "Managing technical debt in software engineering." In: (2016).

[13] Robert L. Nord Philippe Kruchten and Ipek Ozkaya. "Technical debt: From metaphor to theory and practice." In: (2012).

[14] M. Alexandersson Runeson P. and O. Nyholm. "Detection of duplicate defect reports using natural language processing." In: (2007).

[15] Everton da S. Maldonado and Emad Shihab. "Detecting and Quantifying Different Types of Self-Admitted Technical Debt." In: (2015).

[16] Ipek Ozkaya Stephany Bellomo Robert L. Nord and Mary Popec. “Got Technical Debt? Surfacing Elusive Technical Debt in Issue Trackers.” In: (2016).

[17] B. L. Welch. “The generalization of "Student’s" problem when several different population variances are involved.” In: (1947).

[18] Paris Avgeriou Nicolas Guelfi Zengyang Li Peng Liang and Apostolos Ampatzoglou. “An empirical investigation of modularity metrics for indicating architectural technical debt.” In: (2014).

[19] Paris Avgeriou Zengyang Li Peng Liang. “Architectural technical debt identification based on architecture decisions and change scenarios.” In: (2015).