9-2020

Self-Admitted Technical Debt - An Investigation from Farm to Table to Refactoring

Ben Christians
bbc7909@rit.edu

Self-Admitted Technical Debt - An Investigation from Farm to Table to Refactoring

by

Ben Christians

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Software Engineering

Supervised by
Dr. Mohamed Wiem Mkaouer

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

August 2020

The thesis "Self-Admitted Technical Debt - An Investigation from Farm to Table to Refactoring" by Ben Christians has been examined and approved by the following Examination Committee:

Dr. Mohamed Wiem Mkaouer
Assistant Professor
Thesis Committee Chair

Dr. Christian D. Newman
Assistant Professor

Dr. Ali Ouni
Associate Professor

Dr. J. Scott Hawker
Associate Professor
Graduate Program Director

Abstract

Self-Admitted Technical Debt (SATD) is a metaphorical concept which describes the self-documented contribution of technical debt to a software project in the manner of source-code comments. SATD can linger in projects and degrade source-code quality, but its palpable visibility draws a peculiar sort of attention from developers. There is a need to understand the significance of engineering SATD within a software project, as these debts may have lurking repercussions.

While the oft-performed action of refactoring may work against a generalized volume of source code degradation, there exists only slight evidence suggesting that the act of refactoring has a distinct impact on SATD. In fact, refactoring is better understood to convalesce the measurable quality of source code which may very well remain unimpressed by the preponderance of SATD instances. In observation of the cross-section of these two concepts, it would seem logical to presume some magnitude of correlation between refactorings and SATD removals. In this thesis, we will address the extent of such concurrence, while also seeking to develop a dependable tool to promote the empirical studies of SATD. Using this tool, we mined data from 5 open source Java projects, from which associations between SATD removals and refactoring actions were drawn to show that developers tend to refactor SATD-containing code differently than they do code elsewhere in their projects. We also concluded that design-related SATD is more likely to entail a refactoring than non-design SATD.

Contents

Chapter 1

Introduction

Technical Debt is a metaphorical concept which has been used since 1992 to describe the trade-off between quality and schedule of software projects [8]. This metaphor is used to describe the intention of developers to contribute a substandard or incomplete implementation to a project with the intention of making later contributions to "pay back" the debt of quality [18]. Much like with monetary lenders, software project owners need to effectively manage their technical debts to assure that they are not left with low-quality projects that are expensive later on during project maintenance. The tech debt metaphor is far reaching within most development teams, as it provides an easily conceivable problem to both technically- and non-technically minded individuals.

As the idea of technical debt has grown, a need to break technical debt down into categories has provided motivation for several studies [2, 16]. These studies classify technical debt in how it impacts difference aspects of the project such as its design, implementation, or requirements. The common finding among these studies determines that technical debt can exist in all areas of a project and can be contributed during any phase of development. This finding goes far to describe the far-reaching implication that technical debt has on the overall quality of a project.

The modern importance of technical debt has assembled the need for a myriad of measurement and management solutions for all project types. Measuring the amount of technical debt in a project is a hefty task, with a plethora automated and manual solutions having shown to be effective in the past [17, 21]. It is far beyond the scope

of this thesis to identify and compare these methods, as this work instead seeks to improve our understandings of 'why' technical debt is contributed, and 'how' it is addressed.

Understanding how technical debt is addressed in practice can be accomplished by observing refactorings. Refactoring is an important concept to align with the idea of technical debt, as it metaphorically appears within a project as an attempt to repay its technical debt. While refactoring may not be a perfect fit into the metaphor, as the context of the refactoring determines whether or not it is effective at reducing the overall balance of the technical debt, refactoring serves as the base-line standard for identifying actions that can improve project quality without impacting the functionality of the project.

This thesis will observe several large open-source projects to identify instances of refactoring and a subsection of technical debt, Self-Admitted Technical Debt (SATD), to determine what impacts refactoring may have on technical debt and visa versa. This is done with a goal of achieving a better understanding of the effectiveness of contributing SATD to a project, and to determine the qualities of the most effective instances of SATD.

The paper is structured as follows. Chapter 2 will discuss the background knowledge necessary to understand these studies. Chapter 3 will state the two objectives of this research project and detail their motivations. Chapter 4 will discuss similar studies and studies from which understandings will be utilized within this research. Chapter 5 will lay out the methodology that will be used to draw conclusions. Chapter 6 will discuss the findings of the study as they relate to the three research questions. Chapter 7 will identify any known threats to validity. Chapter 8 will conclude the thesis, and identify any future work that can be done within this domain.

Chapter 2

Background

This section will detail all necessary information on topics discussed in this thesis.

2.1 Self-Admitted Technical Debt (SATD)

Self-Admitted Technical Debt is a candid form of technical debt in which the contributor of the debt includes a self-admission in the form of additional documentation. This admission is typically accompanied with a description of a known or potential defect or a statement detailing what remaining work must be done. Well-known and frequently used examples of SATD include comments beginning with TODO, FIXME, BUG, XXX, or HACK [26]. SATD can also take other forms of more complex language void of any of the previously mentioned keywords. Any comment detailing a not-quite-right implementation present in the surrounding code can be classified as SATD.

SATD was first classified as so by Potdar and Shihab in 2014 [26] as the intentional documentation of the contribution of technical debt. Since then, SATD has gained traction within technical-debt-studying community for serving as a quantifiable and measurable way of identifying developer's intentions when contributing technical debt to a project. Prior to Potdar and Shihab, efforts to identify the motivations behind technical debt had relied solely on commit messages and other commit metadata. However, these data points were often too general to accurately extract any related information on the tech debt instances contributed during the commit to the project.

There are different types of technical debt as shown in an ontology proposed by Alves et al. [2]. This ontology encompasses all areas of the software development process and methodologies such as debt that impacts design, implementation, architecture, requirements, and process. Maldonado and Shihab translated to classifiers for SATD in a 2015 study, where the classification were reduced to a more concrete classification of design, requirement, defect, test, and documentation debt [9]. Important for this thesis is the understanding of design-classified SATD, which are source code comments that indicate a potential issue with the design of the surrounding code. These SATD instances can detail misplaced code, a lack of abstraction, long methods, poor implementation, workarounds, or temporary solutions [9]. A common theme among these instances is the portrayal of doubt or uncertainty which offer some insight into the original motivations of contributing technical debt to the overall design of the system.

Modern Integrated Development Environments (IDEs) have begun recognizing the utility of SATD [29]. It is common for them to highlight comments containing the aforementioned keywords, or for them to add SATD to a project when automatically generating unimplemented stubbed functionality to be implemented manually at a later time. Developers and IDEs both contribute SATD with the common assumption that including a self-admission will make their technical debt easier to pay back, or at least reduce the likelihood of it being forgotten. The effectiveness of this strategy needs to be brought into question, as it may have significant impacts on development practices. Understanding the implications of this assumption is vital to assure high-quality performance in software development teams.

2.2 Detecting Removals With Edit Scripts

It has been shown that the primary dataset used by most SATD-related studies has been plagued by issues of accurate comment removal detection [32]. However, the issue of detecting accurate removals is not specific to the study of SATD. In Java,

there are two primary ways of detecting source code changes: through differencing abstract syntax trees (ASTs), or through mapping source code elements between two versions of a file. GumTree is a well renowned tool for differencing ASTs, and is shown to accomplish the task with high precision and accuracy [11]. Upon GumTree exists a Java language specific implementation called IJM, which benefits from using some Java-specific tokens when differencing ASTs [12]. However, some attempts to increase differencing tool's performance rely on a hybrid approach combining both mapping and AST matching [20].

Unfortunately, all of the aforementioned techniques, which happen to exhaust the list of well-regarded Java differencing tools, do not preserve source code comments in their parsing process[1]. This is due to source code being removed as a pre-processing step, which means there is no simple modification to any of these tools that would allow them to be useful during this study. This explains why no accurate source comment detection tools have been created to assist in the large-scale mining of SATD instances. It also poses a serious threat to any potential tool's accuracy that hopes to detect and track SATD instances effectively within Java projects.

Since we will not be allowed the luxury of relying on a proven solution for detecting removals, we must look elsewhere for a reliable means of detecting source code changes. Software developers parse source code changes primarily using Edit Scripts. An edit script details a algorithmic "best guess" as to what a developer's intentions were when mutating a source file from one version to the other. These are commonly displayed by green and red highlighted text as shown in Figure 2.1. Edit scripts have inaccuracies [12], but always detail all changes between two versions of a file exhaustively.

There are several algorithms that produce edit scripts. The Myers algorithm dates back to 1986, and was widely regarded as the "bread and butter" of differencing algorithms [24]. The algorithm has been consistently used by Git as the default algorithm

[1] https://github.com/GumTreeDiff/gumtree/issues/39

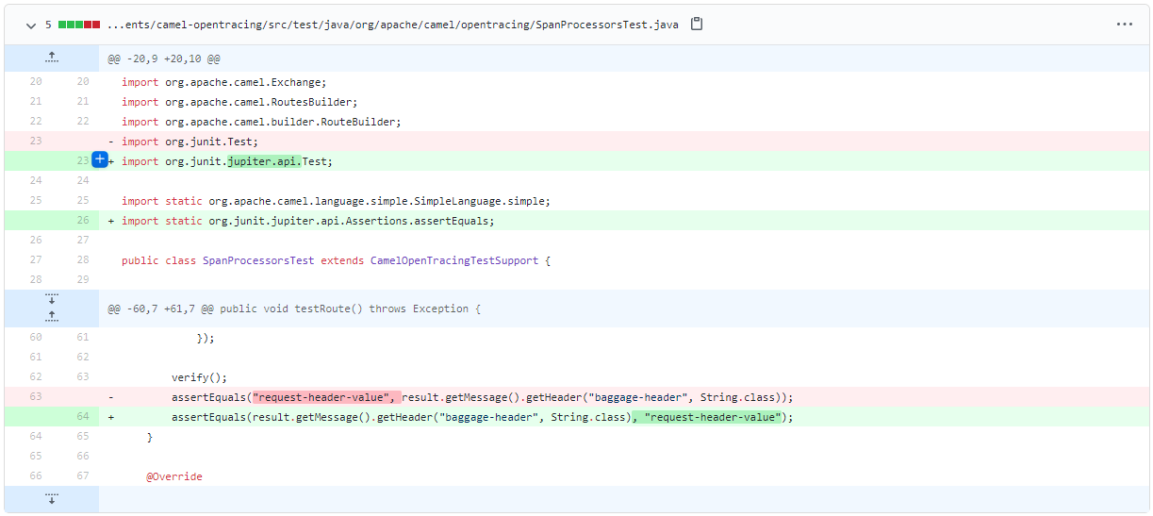

Figure 2.1: Edit Script from the Apache Camel Project (Myers)

for displaying differences between source files [25]. The histogram differencing algorithm serves as another important tool for differencing files as implicitly provided by Git since Git 1.7.7 in 2011[2]. The histogram algorithm is a modified version of Git's Patience algorithm, but offers increased speed[3]. When comparing the histogram algorithm and the Myers algorithm, it was found that the histogram algorithm provided a more accurate representation of developer's intentions when manually classified [25].

2.3 Refactoring

Refactorings are widely regarded as being the blueprints for improving the quality of source code, as refactoring operations are shown to yield direct improvements to the projects when they are performed [27, 1, 13]. Refactorings have also been shown to decrease the number of errors that result from the refactored code [30]. However, refactorings can have a negative impact on source code, as only proper refactorings will have a positive impact on source quality [27]. Due to these important

[2]https://github.com/git/git/blob/77bd3ea9f54f1584147b594abc04c26ca516d987/Documentation/RelNotes/1.7.7.txt#L68-L70

[3]https://marc.info/?l=git&m=133103975225142&w=2

qualities, it makes sense that refactoring is an extremely important focus for the software engineering community. As the practice of engineering software becomes more and more efficient, so must the management of efficient and effective refactoring methods.

Refactoring typically involves re-arranging source code elements such as attributes, methods, or classes, or the renaming of source elements. Refactoring also commonly includes the merging or splitting of code elements. Regardless of the operation, it is done with the goal of making the source code more manageable for other developer to understand. This understandability is shown to directly impact the rate in which developers can make changes [22].

One of the most influential factors of a software project's quality is its design. A poorly designed software system is much more likely to have defects, and is also likely to require more time to change [5, 3]. Software design quality has been effectively represented by many easily obtainable quality metrics [6]; most of which refactorings can have a positive impact upon [7].

Chapter 3

Research Objectives

The objective of this thesis is twofold. The primary objective of this research is to determine the extent of which the presence of SATD impacts software developers' intent to refactor source code. This motivation stems from a need to understand whether including a self-admission to tech debt impacts whether it is eventually paid back.

The secondary objective of this thesis is to contribute a new tool for obtaining larger-scale datasets of SATD instances.

3.1 Contributing a Mining Tool

It can be difficult to identify when source code elements disappear, as many "disappearances" can actually be instances where code elements move throughout a project. This happens to be the case for the current primary dataset used to understand SATD [32]. This thesis will seek to provide a means of automatically detecting these removals correctly, as well as tracking removals to their associated additions. This tool will provide large and rich datasets of SATD instances from throughout a project's history in the form of a large-scale empirical history for each project studied. This intends to allow the study of SATD to progress much faster, and with more confident claims than were previously attainable.

3.2 Aligning SATD Removals & Refactorings

It would be a sensible hypothesis to determine that technical debt which includes a self-documented explanation as to its existence would be more likely to receive a refactoring than other areas of the project. This can be difficult to determine, as there does not exist a perfect method of quantifying developer intent to perform a legitimate removal of SATD instances. However, with better tracking of instances that was previously available, a more reliable conclusion can be drawn about the impact of refactorings on SATD removals.

This understanding can be achieved using two datasets: A dataset of SATD instance removals, and a dataset of refactoring operations. By aligning the code elements that are refactored with the areas in which SATD instances were removed, an understanding of the frequency of these co-locations can be drawn. Then, any abnormalities between this distribution refactoring distributions from other areas of the software projects will determine if SATD instances entail any additional needs for refactoring in practice.

There is also an opportunity to determine if design classified SATD may have impacts on the refactor rates of SATD instances. As refactoring primarily addresses design-quality concerns, it would make sense that design-classified SATD instances would be more widely refactored than other SATD instances. This study will generate a classification model based on a prior dataset of manually classified SATD instances to use for classifying the SATD removal dataset.

Chapter 4

Related Work

SATD is a subsection of technical debt that was only brought to term recently, in 2014 [26]. Since then, several important discoveries have been made. However, most, if not all findings have stemmed from a very limited dataset of questionable completeness [32]. This section will discuss the works that have been done in relation to data curation for SATD instances, as well as the studied impacts of refactorings on technical debt (focusing mostly on SATD).

4.1 SATD Data Curation

The investigation of Self-Admitted Technical Debt began to gain traction in 2014 with the study of Potdar and Shihab [26]. Initial approaches to classifying source comments as SATD involved intensive manual efforts. Potdar and Shihab manually classified 101,762 Java code comments and generated a string matching heuristic based off of 62 commonly occurring comment patterns. This heuristic inspired Maldonado to apply this classification to 10 projects in 2015[9], and then Bavota and Russo to expand this classification to 159 projects in 2016[4].

Maldonado would later expand the classification approach to use Natural Language Processing in 2017[10]. Despite the increased performance of classification models, little work has been done to develop an empirical understanding of SATD in Java projects, as only one empirical study has been conducted (by Maldonado[19]) which addressed 5 large open source projects. The quality of this dataset has been

brought into question by Zampetti, who manually filtered and improved the dataset in an effort to improve its quality [32]. While this filtered dataset is regarded to be high quality, the filtering process removed a significant number of entries. Additionally, there does not exist a means to expand its size or to curate data from other projects.

In 2018, Huang et al. developed a new SATD classification model which improves the F-1 score of classification over Maldonado et al. by 27.95% [10, 14]. This model is built using a composite classifier which selects SATD classifications based on a majority voting selector off of the top 10% most effective features extracted from the manual classifications of Maldonado in 2015 [9]. This implies that it will produce project-independent classifications, which satisfies a significant requirement for a classifier used in a large-scale mining tool.

There is now an opportunity to take advantage of this improved detection tool to enhance research efforts with a highly accurate, large scale empirical history of SATD instances in Java projects previously unavailable. This can be accomplished alongside of fixing some of the data quality issues noted with Maldonado's empirical study[19]. This part of this thesis will aim to package these improvements and model in a tool will allow further efforts to expand past these 7 previously available software projects in terms of size and quality. In addition to publication of this tool, an empirical history of SATD instances in 790 open source software projects will be made available as produced by the tool, SATDBailiff.

4.2 Aligning Refactorings and SATD Removals

There is currently only a single study which has observed this exact cross-section of topics. Iammarino et al. first observed the impacts of refactorings on SATD removals using data from the RMINER tool juxtaposed with the dataset from Maldonado's 2017 empirical study [19] [15]. This was done with an objective of determining the extent of co-occurrences in a simple sense, as this was the first study to pay attention

to such an idea. Iammarino also observed the impacts of different types of refactorings on SATD removals. To perform their study, Iammarino er al. aligned the commit hashes of the two datasets to determine whether a commit that contained a refactoring would also contain a removal of an SATD instance, and visa versa. The study concluded that between 11.45% (Apache Camel) and 59.85% (Apache Tomcat) of SATD instance removals co-occurred with a refactoring action during the same commit. This is a large range, especially considering the study only observed 4 projects. Iammarino concluded that only within the Apache Camel project was there a statistically significant difference between the refactorings centered around SATD removals and the refactoring taking place within the remainder of the project. In regards to the impacts different refactoring operations have on SATD instance removals, Iammarino et al. found that there was a large variance between projects, however the most impactful operations were the moving of source code operations and attributes and extracting source code operations.

This preliminary study leaves much room for improvement. Specifically, the grain in which SATD instance removals are observed can be significantly improved, as to get a better distinction between inter-commit impacts of refactorings versus the real location of SATD instance removals. The study also admits to statistical insignificance due to data sample size, which can be remedied by using a larger dataset provided by SATDBailiff. As with any study, increasing the fidelity and size of the observed sample is likely to produce more conclusive findings.

Chapter 5

Methodology

This thesis is separated into two distinct parts, the second of which relies on the first. The first objective is do create a mining tool to offer a more complete picture of all SATD instances available in a project. The second objective is to build upon the data provided by this tool to acquire a more accurate and confident conclusion about the impact of SATD on refactoring actions and visa versa. Also within the second object, we will address whether design-classified SATD instances have any significant impact on refactoring operations.

5.1 Mining Tool

The mining tool is built with the intention of obtaining the highest accuracy possible for detecting SATD instances. This accuracy is defined by the tool's ability to not only track when instances are added to and removed from a project, but also in the tool's ability to detect when an instance is modified. In order to make sense of these modifications, the tool must also track instances from the time they are added until the time they are removed. The tool will join any related operations on a single "SATD Instance Id" to show this relation. Figure 5.1 shows the relationship between entities in the proposed solution, SATDBailiff.

The Eclipse JGit library[1] (3) is used to collect Java source files from local or remote Git repositories. This library is also used to collect any commit metadata

[1] https://github.com/eclipse/jgit

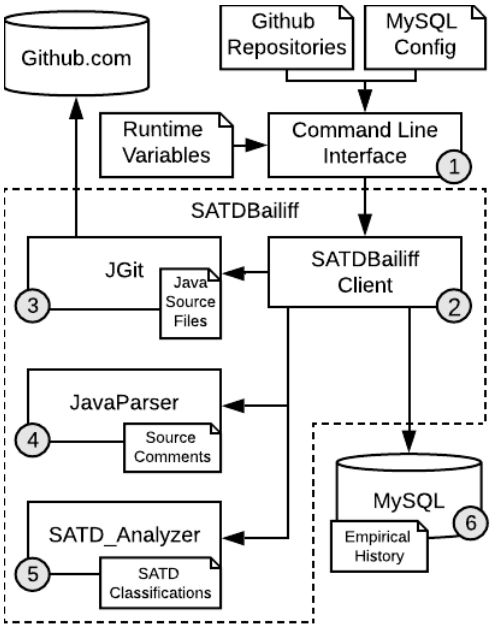

Figure 5.1: SATDBailiff Flow

available, which is output alongside any associated SATD operations found during mining. JGit is also used to generate edit scripts between different versions of a project's source code. These edit scripts detail which lines contain removals and additions, and can be used by logic elsewhere to determine to what extent source code comments are modified between commits. SATDBailiff is configured to use both the Myers and Histogram differencing algorithm to generate these edit scripts.

The JavaParser library[2] (4) is used to extract source code comments from the Java source files obtained by JGit (3). The library is also used to extract comment metadata such as the containing method and class, and line numbers. This metadata is output alongside any associated SATD operations found during mining.

The SATD_Analyzer tool presented by Huang et al. [14] (5) is used for the binary classification of source comments as SATD. This state-of-the-art tool achieved an average F-score of 0.737 during the classification of comments from 5 major open source projects. The classification model works in a high-performance environment,

[2]https://github.com/javaparser/javaparser

and will have a minimal impact on the runtime of SATDBailiff. Within SATDBailiff, this classification interface was designed with a level of abstraction, and any future higher-performance models can be used with SATDBailiff as well given minimal modifications.

The logic that bridges all of these tools together is located within the SATDBailiff client (2). The client begins by generating parent-child pairs for every single parent commit found under a given head of the git repository. For the sake of simplicity, commits with multiple parents (i.e. merge commits) are ignored. Then, for each of those pairs, all source code differences (edit scripts) are calculated for each Java file. Before execution, the algorithm to be used for generating these edit scripts can be configured between the Myers algorithm and the histogram algorithm. All SATD instances are recorded from each file impacted by a source code modification in the parent commit, as well as the child commit. This is achieved by first recording all comments in only the files impacted by changes between the commits. The set of comments are cleaned by removing all commented-out source code, JavaDoc comments, and license comments, following the precedence of similar studies [19, 9].

A mapping approach is taken to identify which SATD instances may have been impacted by these changes. An SATD instance will map between two commits if both commits contain the same comment, under the same method signature (or lack thereof), and the same containing class name (or lack thereof). SATD instances that share all of those identification properties (ex. two identical SATD comments in the same method), a number is assigned based on the order they occur. This step is taken to avoid issues presented by multiple identical SATD comments located within the same method. It is important to understand that SATDBailiff does not track the line-number-location of SATD instances during commits in which the SATD instance is not modified, as this would severely impact the runtime duration of the tool.

All SATD instances that were not mapped between the two commits are then classified as removed or changed. This classification is determined by the edit scripts

generated earlier, and the logic is further described in the subsection below, Section 5.1.2. The result of this process is a complete empirical history of all operations to SATD instances between a given point in a project's lifetime and its origination.

5.1.1 Output

The implicit implementation of SATDBailiff outputs to a SQL database (6), but the tool supports a modular implementation allowing for an extension of other output formats. A simplified data-point sample from the Apache Tomcat project is included in Table 5.1. The data includes some important features:

- SATD Id and SATD Instance Id. Each entry has two identifying integers. The SATD Id is a unique identifier for a single operation to an SATD Instance. An SATD Instance ID is an overarching identifier used to group many SATD operations to a single contiguous instance. In Table 5.1, the "instance_id" field represents the shared instance between the three different SATD operations. Each entry in this sample would have a different and unique SATD Id.

- Resolution. Each SATD operation has a single resolution that impacts the SATD between two commits. These operations include: SATD_ADDED, SATD_REMOVED, SATD_CHANGED, SATD_MOVED_FILE, FILE_REMOVED, FILE_PATH_CHANGED, and CLASS_OR_METHOD_CHANGED. The definitions of these operations are described in detail in Section 5.1.2.

- Comment Metadata. When each SATD operation is recorded, SATDBailiff also records the comment's metadata at the time of the operation. This includes data such as the comment type (Line, Block, or JavaDoc as recorded by JavaParser), start and end line, containing class and method, the file name, and the comment itself.

instance_id	resolution	commit	comment
789	SATD_ADDED	09b640e	TODO: 404
789	FILE_PATH_CHANGED	decfe2a	TODO: 404
789	FILE_REMOVED	a457153	None

Table 5.1: Simplified Sample Data from the Apache Tomcat project

- Commit Metadata. When each SATD operation is recorded, SATDBailiff also records the metadata of both the child and parent commit. This includes author name and timestamp, committer name and timestamp, and SHA1 commit hash.

5.1.2 Operations

Previously, Maldonado [19] established an empirical history that recorded changes to SATD instance incorrectly as removals and additions. In addition to mistaking file renames, this would detect instances like the example in Figure 5.4 and Figure 5.5 as having both resolved the original SATD instance and added the new version to the project. SATDBailiff resolves this issue by handling SATD comment and file name changes as operations in-between additions and removals of SATD. In order to observe a more fine-grained change in source code changes, the tool observes edit scripts for changes to specific lines of code made between each commit.

The next subsections describe the process of identifying each of the operations that SATDBailiff handles. The sections use the variables:

- C_a, C_b, the parent commit and the more recent commit, respectively.
- S_a, S_b, a specific SATD instance in the parent commit and the more recent commit, respectively. It should be assumed that S_a and S_b are intentionally related to each other if not identical.
- SA_a, SA_b, an arbitrary other SATD instance in the same file unrelated to S_a or S_b in commits C_a and C_b respectively.

- $E_1, E_2, ...E_n$, the edit scripts generated when differencing C_a and C_b that impact the lines of SATD Comment S_1. Multi-line SATD comments may have multiple edit scripts that impact it where n is used to differentiate these line-based edit scripts.

SATD_ADDED & SATD_REMOVED

The previous and naive algorithm determines SATD_ADDED instances would exist in any C_a where S_a is not present and the associated C_b where S_b is present. This satisfies a basic case in Figure 5.2.

```
      body = exchange.getOut().getBody();
+  // TODO: what if exchange.isFailed()?
      if (body != null) {
```

Figure 5.2: A basic case SATD_ADDED instance

However, SATDBaillif needs to account for changes in SATD comments. In Figure 5.4, these changes would be identified by separate SATD_REMOVED and SATD_ADDED instances using the naive logic. Instead, it should be determined that if a single edit script E_n exists such that E_n impacts S_b without impacting SA_a, then S_b was added by C_b.

The previous and naive algorithm also determines SATD_REMOVED instances would exist in any C_b where S_b is not present and the associate C_a where S_a is present. This satisfies the basic case in Figure 5.3.

However, in the case of Figure 5.4, it is seen that a more robust algorithm must be used to detect SATD removals. SATDBailiff handles this case such that if a single edit script E_n exists such that E_n impacts S_a without impacting SA_b, then S_a was removed by C_b.

It can also be the case that the previous logic used by SATDBailiff to classify

```
  protected void connectIfNecessary() {
-   // can we avoid copy-pasting?
      if (!client.isConnected()) {
```

Figure 5.3: A basic case SATD_REMOVED instance

additions and removals to be false, and for the tool to still classify an operation as an SATD_ADDED or SATD_REMOVED. This case is better identified by the logic in the following SATD_CHANGED section. There also exists a case where the SATD instance is removed in the currently observed file, but is added in another file. This case is better identified by the logic in the follow SATD_MOVED_FILE section.

```
      logger.log("Init successful");
-   // Moved this config to the bottom
+   // Moved this config
+   // to the bottom
      super.init();
```

Figure 5.4: A case of a would-be false SATD_ADDED and SATD_REMOVED instances

SATD_CHANGED

Changes in an SATD comment can be difficult to determine because changes can possibly remove the SATD comment entirely, replacing it with a new non-SATD or irrelevant comment. Only SATD comments which preserve the original intent of the SATD comment should be recorded as an SATD_CHANGED operation. To identify whether a change in a comment is possibly an SATD_CHANGED operation, the edit scripts of the file are first observed. If a single edit script E_n exists such that E_n impacts both S_a and SA_b, then it can be determined that a change may have

occurred. Figure 5.5 and Figure 5.6 detail two cases where this is the case.

```
   try {
-   // Maybe this already exitst
+   // Maybe this already exists
       success = client.changeDir(dirName);
```

Figure 5.5: A valid SATD_CHANGED instance

```
   c = endpoint.createChannel(session);
- // TODO: what if creation fails?
+ // Bug 1402
   c.connect();
```

Figure 5.6: A possibly valid but false SATD_CHANGED instance

To determine whether a change to an SATD instance preserves the original intent of the instance, a normalized Levenshtein distance [31] is used to determine the extent of the modifications. If this normalized distance is less than an arbitrarily chosen threshold of 0.5, then we can determine the SATD instances are related after an update.

This method works exceedingly well for many of the common cases of changes that SATD comments face. These changes include additions of newlines (see Figure 5.4), spelling corrections(Figure 5.5), and URL updates. Other common changes in which this method may be less predictable include the addition of adjacent related and unrelated comments, which the system will group with the SATD comment when extracting comments from the source files.

SATDBailiff also checks the updated comments to determine if they still can be classified as SATD instances. Figure 5.7 shows an example of an SATD instance

which is recorded in C_a as "lets test the receive worked\nTODO" due to how the tool groups adjacent comments. In C_b, the removal of the \nTODO substring of the instance results in the instance no longer being classified as SATD, and thus SATDBailiff reports this instance as SATD_REMOVED. If this additional check to determine if the changed instance in C_b was not made, this SATD instance would be incorrectly reported as having only been changed.

```
  // lets test the receive worked
- // TODO
- // assertMessageRec("???@localhost");
+ assertMessageRec("copy@localhost");
  c.connect();
```

Figure 5.7: SATD_REMOVED instance removing only part of a comment

SATD_MOVED_FILE

Detecting the movement of SATD instances within a single commit can be difficult because movement will not always be a one-to-one operation. It is commonly a case where an operation in a parent file is duplicated in multiple children files, thus proliferating the SATD instance alongside the pushing down of the operation. In this case, we need to establish a parent-child relationship. We also cannot rely on edit scripts to make this detection, as edit scripts do not bind between files. This means it is exceedingly difficult to accurately predict developer intent in making this classification.

SATDBailiff accomplishes this detection using only mapping. Whenever a SATD_REMOVED instance is identified, SATDBailiff compares the instance against every potential SATD_ADDED instance in the new version of the project. If any instances have matching comment bodies, then we can determine that the SATD instance was moved. However, in the case where multiple identical-bodies SATD instances are moved as

multiple methods containing a instances of each are moved between files, this mapping logic will not suffice. In this case, SATDBailiff identifies if there first exists at least one SATD_ADDED instance in the new version of the project with both an identical comment body and method signature. If there exists one (or more) of these, then the parent-child mapping is made only to those methods. Otherwise, the parent-child mapping is made to new SATD_ADDED instances.

This is by no means a perfect method of detection, however it achieves a goal of limiting the number of SATD_REMOVED instances that should actually be classified as SATD_MOVED_FILE instances. This is of utmost importance for this study as many refactoring actions involve the movement of source code objects between files, and we do not want these movements detected as removals as they would significantly impact the number of removals that falsely co-occur with refactoring actions.

CLASS_OR_METHOD_CHANGED

The final edit script source code change detected by SATDBailiff is modification to an SATD instance's containing class or method. These cases are detected if any E_n impacts the class or method containing S_a such that the method signature or the class name are changed.

FILE_REMOVED & FILE_PATH_CHANGED

File removals are detected implicitly by Git, where a similarity between added and removed files determines whether a file is removed or renamed when it is no longer present in the repository when committed. This detection method was available as part of the JGit library, and was utilized for identifying FILE_REMOVED and FILE_PATH_CHANGED instances.

Project	# SATD Instances	# SATD Removals
camel	2,098	1,090
gerrit	5,41	226
tomcat	2,205	994
hadoop	2,271	766
log4j	195	64
total	7310	3140

Table 5.2: Datapoints from SATDBailiff v1.1

5.2 SATD & Refactoring Co-occurrences

SATD instances were collected using SATDBailiff (version 1.1), an Open Source tool for obtaining empirical data of SATD instances from Java projects. The data provided by this tool includes the tracking of SATD instances from creation to removal from a project, and accurately detects when SATD Instances are removed rather than modified slightly or moved between files. Table 5.2 shows the number of datapoints made available from each project. For the sake of consistency with priorSATD-centric studies, this data was only collected from between each project's initial commit, and the latest commit recorded for each project contained in the dataset popularized by Zampetti's manually filtered dataset [32]. However, this dataset contains approximately twice the number of instances and removals as the filtered dataset, as SATDBailiff offers a more complete set of data points than was previously available.

The RefactoringMiner tool (version 2.0) [28] was used to collect refactoring instances from the same 5 projects in Figure?? between the same temporal bounds. The tool's output was modified slightly to include a different formatting of method signatures needed to align with the dataset of SATD instances. These changes include reconstruction of method signatures before and after refactorings, and was necessary due to how the method signatures are obtained by the RefactoringMiner tool. The script used to obtain this modified dataset can be found on GitHub[3] as a fork of the

[3]https://github.com/bbchristians/RefactoringMiner

Project	# Refactoring Instances
camel	44,262
gerrit	17,752
tomcat	23,702
hadoop	43,955
log4j	2,903
total	132,574

Table 5.3: Data points from RefactoringMiner

master branch of RefactoringMiner 2.0. The number of data points collected using the modified version of RefactoringMiner can be seen in Table 5.3.

Alignment of the SATD and the Refactoring dataset is performed in 3 different methods, where each method constitutes a more coupled level of relation to by classified as a co-occurrence. It is impossible to determine a developers entire intention behind removing an SATD instance, and also in performing a refactoring action, so we look to other metrics to determine how related two of these actions may be. These three levels can be called ”Primary”, ”Secondary”, and ”Tertiary”, following the conditions:

- Primary Co-occurrence: This is the highest level of confidence that a refactoring action and SATD instance removal are related. In this case, the commit, method, class, and file path of both the SATD which is removed, as well as the refactoring action must be identical. In a primary co-occurrence, it can be determined that the refactoring action taken is likely closely related to the removal of the SATD instance.

- Secondary Co-occurrence: In this case, the commit, class and file path of both the SATD instance which is removed, as well as the refactoring action, must be identical. In a secondary co-occurrence, it can be determined that there is possibly some relation between the refactoring action taken, and the removal of the SATD instance. Some examples of secondary co-occurrences in the studied

```
-         // TODO: on demand use configuration
-         SslHandler sslHandler = configureServerSSLOnDemand();
+         SslHandler sslHandler = configureServerSSLOnDemand(configuration);
```

Figure 5.8: Example of a Secondary Co-occurrence retrieved from https://github.com/apache/camel/commit/c90f924412bbcba14678d3eed8b7c32a57e95c05?diff=unified#diff-e50303cf1d6c780530c4d09428eb591dL87

projects include refacgoring to SATD instances without methods, large-scale file re-writes that include many refactorings where none of which refactorings impact the SATD instance's method[4], modification of another method's signature (Figure 5.8), modifications to dependent methods (Figure 5.9), etc.

- Tertiary Co-occurrence: In this case, only the commit of both the SATD instance removal and the refactoring action must be identical. In a tertiary co-occurrence, it can be determined that there is possibly some relation between the refactoring action taken and the removal of the SATD instance. A tertiary co-occurrence is identical to the classification made by Iammarino's SATD and refactoring co-occurrence study [15]. There exist very few examples of an precise connection between a tertiary removal's refactoring action and SATD removal co-occurrence, and while there were no instances that were easily identifiable through a brief manual verification, it is possible that some exist. For example, the co-occurrence detailed in Figure 5.8 could very easily contain a refactoring to a method in a separate file, as opposed to within the same file as it was actually. However, it has been shown that refactorings often occur in batches [23], which could also entail indirect relations between a presence of technical debt and large-scale refactoring operations.

There is also a need to make design classifications for each of our SATD instances, as we want to observe the impacts of refactorings on SATD instance removals in

[4]https://github.com/apache/camel/commit/006e9bba7e2d2f80daaae1173484a647a9ac8843?diff=split#diff-340b66e822537363a921c78e1c796bec

```
102  94        protected void appendHeadersFromCamel(MimeMessage mimeMessage, Exchange exchange,
103       -                                          org.apache.camel.Message camelMessage) throws MessagingException {
104       -          Set<Map.Entry<String, Object>> entries = camelMessage.getHeaders().entrySet();
105       -          for (Map.Entry<String, Object> entry : entries) {
          95 +                                          org.apache.camel.Message camelMessage)
          96 +          throws MessagingException {
          97 +
          98 +          for (Map.Entry<String, Object> entry : camelMessage.getHeaders().entrySet()) {
106  99                 String headerName = entry.getKey();
107 100                 Object headerValue = entry.getValue();
108 101                 if (headerValue != null) {
       @@ -110,28 +103,26 @@
110 103
111 104                       // Mail messages can repeat the same header...
112 105                       if (ObjectConverter.isCollection(headerValue)) {
113 106                           Iterator iter = ObjectConverter.iterator(headerValue);
114 107                           while (iter.hasNext()) {
115 108                               Object value = iter.next();
116 109                               mimeMessage.addHeader(headerName, asString(exchange, value));
117 110                           }
118 111                       } else {
119 112                           mimeMessage.setHeader(headerName, asString(exchange, headerValue));
120 113                       }
121 114                   }
122 115               }
123 116           }
124 117       }
125 118
126 119       /**
127 120        * Appends the Mail attachments from the Camel {@link MailMessage}
128 121        */
129 122       protected void appendAttachmentsFromCamel(MimeMessage mimeMessage, Exchange exchange,
130 123                                                 org.apache.camel.Message camelMessage)
131 124           throws MessagingException {
132 125
133       -          // TODO: Use spring mail support to add the attachment
134       -
```

Figure 5.9: Example of a Secondary Co-occurrence retrieved from https://github.com/apache/camel/commit/9a9e8041610ec05e9a35987ed6b748c4de341901#diff-4c7fdf27ca62f5ea6dcc801f4662608eL102

general, as well as on design-classified SATD instances. To achieve this, we utilize a dataset provided by Maldonado & Shihab, which contains manual classifications of SATD type (design, defect, documentation, requirement, and test) in association with their source code comment [9]. This dataset is utilized to train a binary classifier which we use to classify our SATD instances as either 'design' or 'non-design'.

A model was created using a Random Forest algorithm and achieved a F-score of 0.72, a precision of 0.71 and recall of 0.74. This model was used to classify all 7310 mined SATD instances. Because it is possible for an instance to contain multiple variations of an SATD comment in situations where the SATD comment is changed without qualifying as a removal, the determination was made that if any of the comments associated with an SATD instance were classified as a design instance, then the entire instance was classified as a design instance.

Chapter 6

Analysis & Discussion

In this chapter, we present answers to our research questions by analyzing both the performance of the SATDBailiff tool, and the findings from aligning the refactoring dataset with our newly acquired SATD instances. We also observe the impact of our design classifications on this process.

6.1 RQ1: What level of accuracy can be attained by our SATD instance mining tool, SATDBailiff?

To verify the accuracy of SATDBailiff, a manual analysis was performed on a stratified random sample of 200 entries mined from 5 large open source Java projects, each as their own strata. The number of samples taken from each project is determined by the total number of SATD instances mined from the projects. Each of the 200 instances selected for this random stratified sample includes all operations (additions, changes, and removals) performed on a single instance of SATD. Each instance in the sample will represent an entirely unique instance of SATD, of which a simplified example of a single SATD instance can be seen in Table 5.1. The results of this analysis can be seen in Table 6.1.

The 5 projects used for this validation are the same projects used by Maldonado to establish a prior dataset of SATD[9, 32]. SATDBailiff was configured to only mine SATD instances between the original commit to each project, and the most recent commit reported by the Maldonado study.

Project	# Entries	# Correct	Accuracy
gerrit	14	14	1.000
camel	59	55	0.932
hadoop	61	56	0.918
log4j	5	4	0.800
tomcat	61	48	0.787
Total	200	177	0.885

Table 6.1: SATDBailiff Manual Analysis Results

When given this set of SATD instances, we located the exact location of each of the SATD operations using the Github website's difference browser to determine whether the SATD instances mined by SATDBailiff were correctly identified, preserved, and reported. A "correct" entry was identified as an entry in which every operation made to the SATD instance could be located using the Github interface. Any unnecessary additional, missing, or inaccurate operations found using the Github client would result in the entire entry being incorrect. For entries that were not removed from the project, their existence in the terminal commit supplied to SATDBailiff was confirmed. For transparency of this analysis, a Github link to the exact source modification was recorded in each of the projects where available. These results are available on the project's website[1]. During validation, it was assumed that all binary classifications of source code comments as SATD were correct. It should be noted that this classification was performed on SATDBailiff v1.0, which did not include the SATD_MOVED_FILE logic.

The results of the manual analysis (Table 6.1) find SATDBailiff to have an accuracy of 0.885. While a higher level of accuracy could have been achieved, it should be noted that many of the incorrect instances were partially correct. For example, instances frequently were found to be incorrect because they became dissociated with one another, where a connection between an SATD instance's addition to the project and its deletion from the project was not made by the tool. In cases where only

[1]https://bbchristians.github.io/SATDBailiff-site/data

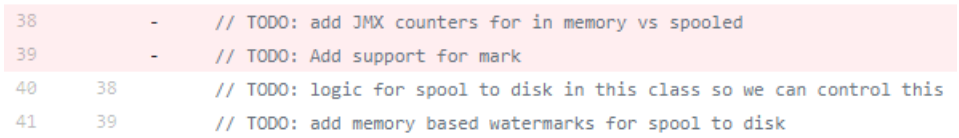

Figure 6.1: A failed case from the Camel project: https://github.com/apache/camel/commit/94133b952907b85faad2a6f3bab79a04be8ebc2a#diff-c80de0a6c0d3e157f20ce8d34163c335L38

the additions or removals are observed from the dataset, the accuracy of the data provided by the tool is much more reliable.

Difficulties in solving many of the tool's issues came from the imperfect nature of working with Edit Scripts produced by Git differencing tools. Edit scripts are used to show an algorithm's best guess of changes in files inside of a Git repository, and do not always reflect the true intentions of the developer who made them [12]. An example of an edit script can be seen in Figure 5.5 depicted as the red and green highlight used to represent a source code change. For this analysis, the Myers algorithm was used during the performed manual validation, which is shown to maintain a manually validated accuracy of less than 0.9 [12]. While, in many cases, an invalid edit script will not directly invalidate SATDBailiff's ability to identify operations to SATD instances, this inaccuracy still serves as a significant limitation in the upper bound of accuracy achievable by this tool.

Other issues arose from the inaccuracy of the all-encompasing use of the normalized Levenshtein distance algorithm used to determine whether an SATD instance is changed or removed. Figure 6.1 shows a failure provided by use of this logic. In any example where multiple adjacent SATD instances occur, we are likely to run into issues with this model, as shown in this example. There is no perfect work-around for this issue, however it is not a common occurrence. In a positive sense, some complex changes are solved by this algorithm, as shown in Figure 6.2. There are many changes to permute the first comment to the second, and the algorithm solves this very nicely.

To assure that these errors are not able to silently pollute the dataset, the tool

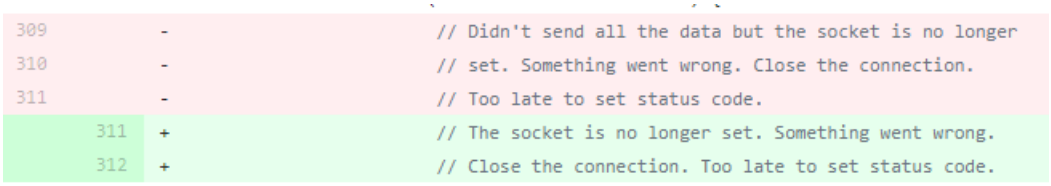

Figure 6.2: A successful case from the Tomcat project: https://github.com/apache/tomcat/commit/82b8c6b9f75f2d82d5c42786483f9ea85d7672ef#diff-3a2bc5e00ec0eccae6b7706734d70bdcL309

reports any known errors that are encountered during the mining process. This workaround was taken as an optimistic precaution for an issue that may not have a perfect alternative solution. For example, SATD that is added during a merge commit which was not present in either of the merge branches is not detected with an SATD_ADDED entry. If that SATD is modified or removed later, then the entry would be added to the project before the SATD_ADDED entry was found. Because the search occurs chronologically starting with the oldest commit in the project, the system can detect this as an issue and will output an error to the terminal during runtime.

6.2 RQ2: Are the refactorings that co-occur with SATD removals different than refactorings that occur elsewhere in the project?

In answering this question, we can hope to show there is different intentions behind normal refactorings than the intentions behind refactoring SATD instances. In making this distinction, we can hopefully determine whether it would be reasonable to assume that other differences exist between how developers view source code with a presence of SATD. To answer this question, we observe the data set containing the number of refactoring actions present in the 5 studied project, compared to the number of those instances which co-occur with an SATD removal, and to what extent those co-occurrences exist. Table 6.2 shows these relations, sorted by the percent of

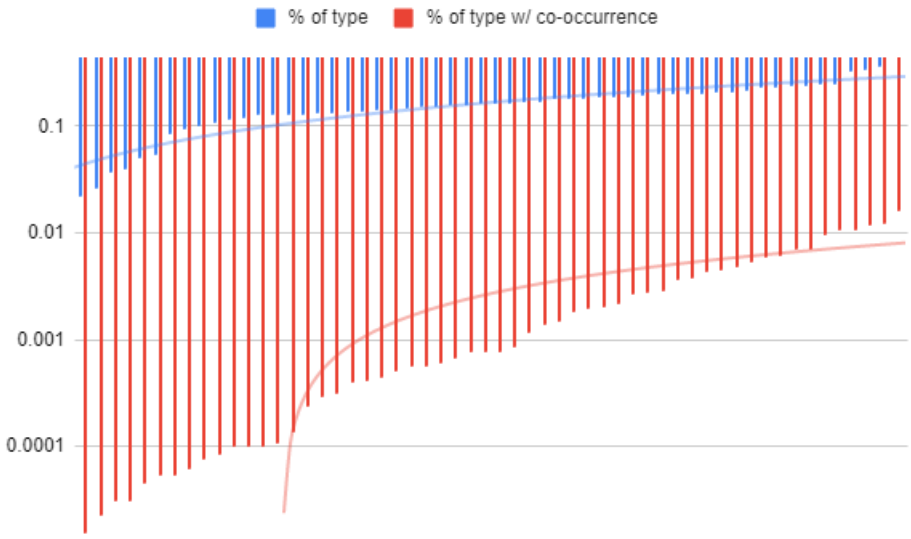

Figure 6.3: Mann-Whitney U Test Distributions

which the total of each action are made up of co-occurrant refactorings.

This table shows that there is obviously an uneven distribution of refactorings that have any co-occurrence with an SATD removal, as an entirely arbitrary constraint would produce a consistent value within the ‘percent co-occurred‘ column. However, this evidence in itself does not prove without a doubt that refactorings around SATD instance removals share a different intent.

To show this in a statistically significant manner, we can perform a Mann Whitney U test. The two tails used for the test are the percent of each operation that co-occur with a SATD instance removal in some degree, and the percentage of each operation that make up the sum of all operations that also share some degree of co-occurrences with an SATD instance removal. The result of the test determine that the normality of the two distributions differ to a significant degree with values of 8.06e-3 and 1.08e-8. The trendlines between the datasets as shown in Figure 6.3 show a negative correlation between these distributions as well.

Refactoring Action	# occ	primary	secondary	tertiary	percent co-occurred
REMOVE_PARAMETER_ANNOTATION	165	0	0	73	44.24%
MOVE_AND_INLINE_OPERATION	553	3	35	158	35.44%
REMOVE_CLASS_ANNOTATION[1]	852	0	25	256	32.98%
REPLACE_ATTRIBUTE[1]	22	0	4	3	31.82%
MOVE_RENAME_CLASS[1]	353	0	11	77	24.93%
MOVE_AND_RENAME_OPERATION	411	5	37	59	24.57%
RENAME_PACKAGE[1]	42	0	1	9	23.81%
MERGE_ATTRIBUTE[1]	55	0	5	8	23.64%
MOVE_ATTRIBUTE[1]	2569	0	261	327	22.89%
MOVE_OPERATION	3137	20	348	339	22.54%
SPLIT_ATTRIBUTE[1]	19	0	0	4	21.05%
RENAME_CLASS[1]	1252	0	30	231	20.85%
REMOVE_PARAMETER	3017	18	123	487	20.82%
EXTRACT_CLASS[1]	500	0	33	68	20.20%
RENAME_METHOD	6156	39	216	987	20.18%
SPLIT_VARIABLE	10	0	2	0	20.00%
RENAME_VARIABLE	4079	45	190	569	19.71%
INLINE_OPERATION	940	8	54	122	19.57%
RENAME_ATTRIBUTE[1]	2709	0	104	400	18.60%
RENAME_PARAMETER	5011	16	221	685	18.40%
EXTRACT_SUBCLASS[1]	71	0	3	10	18.31%
CHANGE_RETURN_TYPE	7585	13	157	1213	18.23%
CHANGE_VARIABLE_TYPE	11951	49	226	1854	17.81%
REORDER_PARAMETER	62	0	0	11	17.74%
PULL_UP_OPERATION	3337	6	119	440	16.93%
EXTRACT_SUPERCLASS[1]	397	0	12	54	16.62%
INLINE_VARIABLE	618	4	24	73	16.34%
CHANGE_PARAMETER_TYPE	9899	8	235	1369	16.28%
MOVE_CLASS[1]	2340	0	15	362	16.11%
PUSH_DOWN_OPERATION	717	2	22	90	15.90%
MERGE_PARAMETER[2]	90	0	3	11	15.56%
ADD_PARAMETER	9210	3	268	1132	15.23%
REPLACE_VARIABLE_WITH_ATTRIBUTE	349	2	17	34	15.19%
CHANGE_ATTRIBUTE_TYPE[1]	6451	0	160	772	14.45%
EXTRACT_AND_MOVE_OPERATION	1862	14	48	204	14.29%
MOVE_RENAME_ATTRIBUTE[1]	28	0	0	4	14.29%
SPLIT_PARAMETER[2]	22	0	3	0	13.64%
REMOVE_METHOD_ANNOTATION	3517	46	44	389	13.62%
PULL_UP_ATTRIBUTE[1]	1822	0	44	199	13.34%
PARAMETERIZE_VARIABLE	571	3	28	43	12.96%
REMOVE_ATTRIBUTE_ANNOTATION[1]	420	0	4	50	12.86%
EXTRACT_ATTRIBUTE[1]	299	0	14	24	12.71%
EXTRACT_OPERATION	6240	3	273	508	12.56%
MERGE_VARIABLE	48	1	1	4	12.50%
EXTRACT_INTERFACE[1]	151	0	1	17	11.92%
EXTRACT_VARIABLE	3095	33	75	245	11.41%
ADD_CLASS_ANNOTATION[1]	3401	0	18	346	10.70%
PUSH_DOWN_ATTRIBUTE[1]	307	0	6	25	10.10%
ADD_METHOD_ANNOTATION	16573	30	157	1372	9.41%
MODIFY_PARAMETER_ANNOTATION	84	0	0	7	8.33%
ADD_ATTRIBUTE_ANNOTATION[1]	2853	0	10	142	5.33%
MODIFY_METHOD_ANNOTATION	1543	0	5	73	5.06%
MOVE_SOURCE_FOLDER[1]	207	0	4	4	3.86%
ADD_PARAMETER_ANNOTATION[1]	354	0	0	13	3.67%
MODIFY_CLASS_ANNOTATION[1]	1559	0	7	34	2.63%
MODIFY_ATTRIBUTE_ANNOTATION[1]	2691	0	24	35	2.19%

Table 6.2:

[1] Operation does not include method signature – cannot be primary

[2] Operation method signature could not be preserved – cannot be primary

Design				
Sample	Total Instances	Primary	Secondary	Tertiary
All	1,924	5.83%	22.34%	34.13%
Removed Instances	723	12.47%	47.78%	72.99%
Removed Instances With Methods	652	13.82%	-	-

Table 6.3: Design SATD Instance Occurrences

Non-Design				
Sample	Total Instances	Primary	Secondary	Tertiary
All	5,386	3.97%	18.08%	31.10%
Removed Instances	2,415	6.78%	30.87%	53.09%
Removed Instances With Methods	2,189	9.78%	-	-

Table 6.4: Non-Design SATD Instance Occurrences

6.3 RQ3: Are design-classified SATD instances more likely to entail a refactoring co-located with their removal than non-design SATD instances?

To answer this question, we must observe the differences in how design-classified SATD (DC-SATD) and Non-design-classified SATD (nDC-SATD) are removed. Table 6.3 and Table 6.4 show the number of DC-SATD and nDC-SATD instances (respectively), alongside their co-occurrences with refactoring operations. The significance of these co-occurrences varies from a primary co-occurrence to a tertiary co-occurrence, referring to the highest potential of relation to the lowest, respectively,

The most important data points from these tables are the number of primary co-occurrences from all removed instances with methods. The DC-SATD instances under this criteria primarily co-occur 13.82% of the time with a refactoring action, while nDC-SATD instances only primarily co-occur 9.78% of the time with a refactoring action. This finding shows that DC-SATD is 42.2% more likely to share a primary co-occurrence refactoring than nDC-SATD. This number also holds for the number of secondary and tertiary co-occurrences maintaining a 54.8% and 37.5% difference

between DC-SATD and nDC-SATD respectively.

This distinction clearly shows that Design-classified SATD instances are much more likely to see a refactoring than non-Design-Classified SATD instances. In following this conclusion, it can be determined that contributing SATD with attempts to detail flaws in a system's design is a more effective way to produce a future refactoring than by contributing more general SATD.

Chapter 7

Threats to Validity

In this section, we identify potential threats to the validity of our approach and our experiments.

7.1 SATDBailiff

Threats to the validity of this tool include the limited manual evaluation and general lack of testing.

Since the manual verification of samples is a human intensive task and it is subject to personal bias, it can be addressed as the most important threat. Manual validation was performed by a single author, so for the sake of transparency, all decisions that were made were documented as transparently as possible. Direct links to the locations in the source code through the GitHub commit browser were included so that any future validation efforts can merely confirm whether those classifications were made correctly without duplicating a majority of the classification effort.

Only 200 samples of SATD were recorded and addressed to determine the accuracy of SATDBailiff, which would ideally be much higher. There was limited time available for other formal and repeatable forms of programmatic testing, as a majority of the validation effort was allocated to the manual validation and validation done during development.

There exists more threats that relate to the SATD instances that are extracted only from open source Java projects. Our results may not generalize to commercially

developed projects, or to other projects using different programming languages. The classification model of SATD is implemented using a composite voting system in an attempt to remain as project-independent as possible [14]. However, it is still possible that SATDBailiff will have widely different performance metrics on other projects.

7.2 Co-occurrence Study

This study was performed with the intention of remaining within the same system scope as previous popular studies stemming from the primary previously available dataset [32]. This was done with the intention of directly confirming or denying prior conclusions which may have been drawn with potentially skewed data from the same projects. While this is a benefit to offer the field of SATD, it does have the negative impact of limiting the size and age of our datasets. Ideally, these studies would have been performed on additional system scopes to gain more complete and thorough conclusions, however time constraints prevented this from happening. The run-time of the two tools used to generate the datasets is exceedingly long, taking upwards of a few days each to obtain data from another sample of 800 projects.

Some refactoring actions could not be accurately modified to work in our study as detailed by [2] in Table 6.2. Because of this, the number of primary co-occurrences may actually be higher for each project, which could also impact the number of DC- and nDC-SATD instances we recorded with co-occurrences.

The data provided by SATDBailiff is not currently accepted by a recognized software engineering authority. The manual validation of the dataset has provided reasonable confidence as to the conclusions in which it can draw, however there is a potential for the data to be skewed or incorrect.

Chapter 8

Conclusion & Future Work

In this thesis we concluded that SATD solicits a different kind of developer attention in regards to encouraging refactoring. We also found that design-classified SATD instances are more likely to be refactored than non-design classified SATD. This concludes that SATD is potentially effective at begging further need to refactor, especially in regards to quality-related refactoring changes. Using this conclusions, we can deduce that developers may be interested in including more self-admissions to areas of their code which may be lacking quality design. Developers may be concerned with these findings as they have the potential to encourage a smarter and more efficient management of their project's technical debts.

However, if it weren't obvious, the conclusions drawn by this paper are not concrete. There exists a large remainder of work to determine the extend in which the effectiveness of SATD, especially as it related to design-classified SATD, can be used to improve the progressing quality of software projects. In addition, there are many improvements that can be made to this study's methodology to achieve more confident results. Perhaps a more extensive manual analysis of the specifics ways in which primary, secondary, and tertiary co-occurrences occur will bring forth a new understanding from which further research may resolve. Such a qualitative conclusions would surely provide applicable findings if not just motivation. It is a significant surrender as part of this study to not include these findings, however time is always an unforgiving opponent of completion.

In general, many potential paths were not taken which may have been made

newly available with a differently-formatted dataset. Future studies can address more directly the difference between projects which use and resolve SATD and those that don't. Future studies can also track the difference between primary co-occurrences and source code quality metrics to determine the consistency of appropriate design SATD.

While new opportunities are made available by SATDBailiff, the quality of its output is also something that can be improved greatly. Pitfalls relating to the accuracy of edit scripts can surely be addressed using potentially unreleased methodologies. Tracking in a general sense, is always prone to error, as an all-encompassing solution is far from the reaches of the meager time provided to this study.

Bibliography

[1] Mohammad Alshayeb. Empirical investigation of refactoring effect on software quality. Information and Software Technology, 51(9):1319 – 1326, 2009.

[2] Nicolli Alves, Leilane Ribeiro, Vivyane Caires, Thiago Mendes, and Rodrigo Spínola. Towards an ontology of terms on technical debt. Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt, MTD 2014, pages 1–7, 12 2014.

[3] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering, 28(1):4–17, 2002.

[4] Gabriele Bavota and Barbara Russo. A large-scale empirical study on self-admitted technical debt. In Proceedings of the 13th International Conference on Mining Software Repositories, pages 315–326, 2016.

[5] Barry W. Boehm. Software Engineering Economics. Prentice Hall PTR, USA, 1st edition, 1981.

[6] Fernando Brito e Abreu, Miguel Goulão, and Rita (inesc/ist. Toward the design quality evaluation of object-oriented software. 12 2002.

[7] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta. On the impact of refactoring operations on code quality metrics. In 2014 IEEE International Conference on Software Maintenance and Evolution, pages 456–460, 2014.

[8] Ward Cunningham. The wycash portfolio management system. SIGPLAN OOPS Mess., 4(2):29–30, December 1992.

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

[10] E. d. S. Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. IEEE Transactions on Software Engineering, pages 1044–1062, 2017.

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, pages 313–324, 2014.

[12] V. Frick, T. Grassauer, F. Beck, and M. Pinzger. Generating accurate and compact edit scripts using tree differencing. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 264–274, Los Alamitos, CA, USA, sep 2018. IEEE Computer Society.

[13] Péter Hegedűs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. Information and Software Technology, 95:313 – 327, 2018.

[14] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering, 23, 05 2017.

[15] M. Iammarino, F. Zampetti, L. Aversano, and M. Di Penta. Self-admitted technical debt removal and refactoring actions: Co-occurrence or more? In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 186–190, 2019.

[16] Clemente Izurieta, Antonio Vetro, Nico Zazworka, Yuanfang Cai, Carolyn Seaman, and Forrest Shull. Organizing the technical debt landscape. pages 23–26, 06 2012.

[17] R. Khatchadourian and H. Masuhara. Automated refactoring of legacy java software to default methods. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 82–93, 2017.

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

[19] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik. An empirical study on the removal of self-admitted technical debt. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 238–248, 2017.

[20] J. Matsumoto, Y. Higo, and S. Kusumoto. Beyond gumtree: A hybrid approach to generate edit scripts. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pages 550–554, 2019.

[21] I. H. Moghadam and M. Ó. Cinnéide. Automated refactoring using design differencing. In 2012 16th European Conference on Software Maintenance and Reengineering, pages 43–52, 2012.

[22] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. pages 252–266, 01 2007.

[23] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. IEEE Transactions on Software Engineering, 38(1):5–18, 2012.

[24] Eugene W. Myers. An o(nd) difference algorithm and its variations. Algorithmica, 1:251–266, 1986.

[25] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. How different are different diff algorithms in git? use -histogram for code changes. CoRR, abs/1902.02467, 2019.

[26] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In 2014 IEEE International Conference on Software Maintenance and Evolution, pages 91–100, 2014.

[27] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007), pages 10–10, 2007.

[28] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. IEEE Transactions on Software Engineering, 2020.

[29] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. Examining the impact of self-admitted technical debt on software quality. pages 179–188, 03 2016.

[30] Peter Weißgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06, page 112–118, New York, NY, USA, 2006. Association for Computing Machinery.

[31] L. Yujian and L. Bo. A normalized levenshtein distance metric. IEEE Transactions on Pattern Analysis and Machine Intelligence, 29(6):1091–1095, 2007.

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