SATDBailiff - Mining and Tracking Self-Admitted Technical Debt

Ben Christians, Mohamed Wiem Mkaouer

Rochester Institute of Technology, NY, USA

Abstract

Self-Admitted Technical Debt (SATD) is a metaphorical concept to describe the self-documented addition of technical debt to a software project in the form of source-code comments. SATD can linger in projects and degrade source-code quality, but can also be more visible than unintentionally added or undocumented technical debt. Understanding the implications of adding SATD to a software project is important because developers can benefit from a better understanding of the quality trade-offs they are making. However, little has been done to establish a high quality and large scale empirical history for SATD in software projects. SATDBailiff is a first attempt at a tool for mining and tracking SATD instances from Git repositories. The tool mines empirical histories of SATD comments by looking at all versions of source code, and identifying SATD using a previously published Natural Language Processing tool designed specifically for SATD detection.

Keywords: Self-Admitted Technical Debt, Mining Software Repositories

1. Introduction

Technical debt (TD) is a metaphor that describes taking shortcuts in software development that will require additional time to fix (or pay back) in the future [1]. Technical debt commonly occurs when developers conclude development of a section of code before it is complete and optimized [2]. This is done with an understanding that this creates a *debt* which will need additional time and effort to be managed later. Developers tend to understand *some* quality-related implications of adding this debt to their projects, which can be seen unmistakably when looking specifically at *Self-Admitted Technical Debt* (SATD) [3].

Self-Admitted Technical Debt is a candid form of technical debt in which the contributor of the debt self-documents the location of the debt. 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*. 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.

Modern Integrated Development Environments (IDEs) have begun recognizing the utility of SATD. 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.

Since SATD is a written testament of an existing negative manifestation in the source code, several studies have observed the removal of SATD instances to better understand how developers manage and

Email address: {bbc7909,mwmvse}@rit.edu (Ben Christians, Mohamed Wiem Mkaouer)

resolve TD. Detecting the removal of TD is of interest to both researchers and practitioners, as, in addition to indicating the disappearance of a problem, it indicates the changes containing the fix to that TD. Such TD fixes are important to locate, since they can be valuable when taking corrective actions against similar TDs. Several recent studies have been focused on accurately identifying the removal of SATD. For instance, Bavota and Russo [4] have shown that up to 57% of SATD is addressed, and that instance are typically addressed by the same developer who initially contributed to the SATD. However, the identification of SATD removal is complex when taking into account common occurrences when code changes overshadow SATD removals [5], such as the renaming of code elements containing the instance or the accidental deletion of containing source files, among others. While not all accidental *disappearances* of SATD comments imply the correction of any associated technical debt, there is a need for a tool that can reliably track the appearance and removal of SATD instances. This effort would provide valuable support for existing studies by clearly capturing these removals without the need to manually validating them.

To address the above-mentioned challenge, we propose SATDBailiff, a tool that mines, identifies, and tracks SATD instances, while providing a clear description of their *lifespan* in the project. For each identified instance of SATD, SATDBailiff detects the commit in which it was introduced, enumerates all changes that it underwent throughout the later commits up until the commit in which each instance was removed, if applicable.

SATDBailiff was validated using a dataset of previously detected and manually validated SATD instances [6]. SATDBailiff is challenged in identifying and tracking those instances throughout the evolution of five long-lived open-source projects from different application domains, namely *gerrit*, *camel*, *hadoop*, *log4j*, and *tomcat*. Results show that SATDBailiff is efficient by averaging an accuracy score of *0.885* when tracking SATD instances from their appearance in the project until their disappearance.

Tool, documentation, and demo video. SATDBailiff is publicly available as an open source tool[1], with a demo video[2]. A few datasets it has created are also available on the project website[3].

The rest of the paper is organized as follows. Section 2 gives an overview of the necessary information related to SATD and summarizes the related work. Section 3 describes our approach, SATDBailiff. Section 4 details the results of our experiments to evaluate SATDBailiff. Section 5 describes how SATDBailiffcan be used in practice. Section 6 discusses the known threats to validity, while Section 7 draws our conclusions and future investigations.

2. Background & Motivation

The investigation of Self-Admitted Techincal Debt began to gain traction in 2014 with the study of Potdar and Shihab [3]. 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[7], 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[6]. 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[8]) 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 [9]. While this filtered dataset is regarded to be high quality, the filtering process removed a significant number of entries, there does not exist a means to expand its size.

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]. There is now an opportunity to take advantage of this

[1]https://www.github.com/bbchristians/SATDBailiff
[2]https://bbchristians.github.io/SATDBailiff-site/usage
[3]https://bbchristians.github.io/SATDBailiff-site/

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[8]. This study 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 796 open source software projects will be made available as produced by SATDBailiff.

3. Overview of SATDBailiff

SATDBailiff is a Java tool designed to mine the empirical history of SATD instances from Java project Git repositories on a large scale. This is done with the goal of tracking the additions, removals, and changes to SATD instances that occur during the process of software development. SATDBailiff's output can be used to better understand the prominence of SATD in software projects at different points over the course of those projects' lifetimes, while also offering new ways to interpret and visualize those SATD instances. While SATDBailiff accomplishes its objective using a state-of-the-art classification model and a scalable output format, the tool was also designed with modularity in mind, to allow for the implementation of new classification models and output formats. To collect its data, the tool leverages several existing tools as shown in Figure 1.

The Eclipse JGit library[4] (3) is used to collect Java source files from Github. This library is also used to collect any commit metadata 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 represent all changes to a file between one version of a file to the next. Examples of edit scripts can be seen in Figure 3, Figure 4, Figure 5, and Figure 6. SATDBailiff is configured to use both the Myers and Histogram differencing algorithm to generate edit scripts.

JavaParser[5] (4) is used to extract source code comments from the Java source files obtained by JGit (3). It also extracts comment metadata, 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. [10] (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. This classification interface was designed with modularity in mind, and any future higher-performance models can be used with SATDBailiff as well.

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. All SATD instances are recorded from *each file* impacted by a source code modification in the parent commit, as well as the child commit. 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. 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 3.1. 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.

[4]https://github.com/eclipse/jgit
[5]https://github.com/javaparser/javaparser

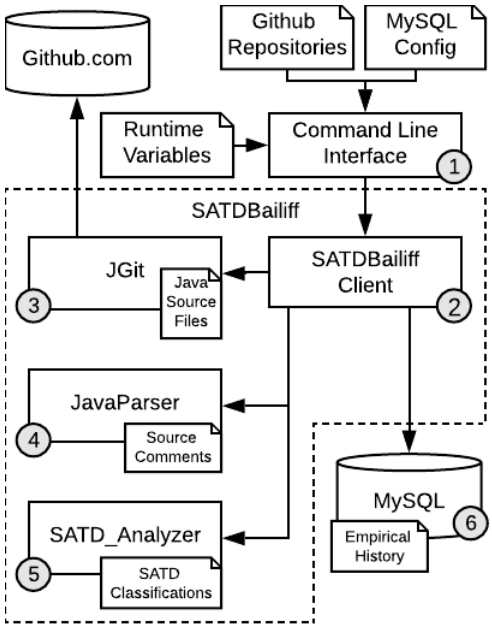

Figure 1: SATDBailiffflow

Figure 2: Simplified Sample Data from the Apache Tomcat project

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

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 Figure 2. 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 Figure 2, 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*, *FILE_REMOVED*, *FILE_PATH_CHANGED*, and *CLASS_OR_METHOD_CHANGED*. The definitions of these operations are described in detail in Section 3.1.

- **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.

- **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.

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

Figure 3: A basic case **SATD_ADDED** instance

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

Figure 4: A basic case **SATD_REMOVED** instance

3.1. Operations on SATD

Previously, Maldonado [8] 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 and Figure 6 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. Edit scripts detail the addition and removal of specific lines of code within a file. An example edit script can be seen in Figure 5 detailed by the red and green highlighted source text.

The next subsections describe the process of identifying each of the operations that SATDBailiffhandles. 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.

*3.1.1. **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 3.

However, SATDBaillif needs to account for *changes* in SATD comments. In Figure 5, 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 4.

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

Figure 5: A case of a would-be false **SATD_ADDED** and **SATD_REMOVED** instances

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

Figure 6: A valid **SATD_CHANGED** instance

However, in the case of Figure 5, 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 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.

3.1.2. ***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 6 and Figure 7 detail two cases where this is the case.

To determine whether a change to an SATD instance preserves the original intent of the instance, a normalized Levenshtein distance [11] 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), spelling corrections(Figure 6), and URL updates.

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

Figure 7: A possibly valid but false **SATD_CHANGED** instance

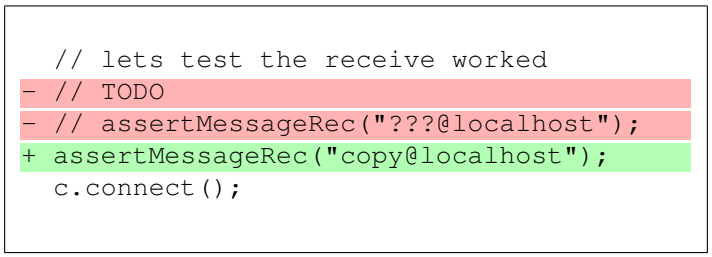

Figure 8: **SATD_REMOVED** instance removing only part of a comment

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 8 shows an example of an SATD instance which is recorded in C_a as `"lets test the receive worked\` 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.

3.1.3. ***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.

SATDBailiff does not currently identify when SATD is moved throughout a file by multiple separate edit scripts.

3.1.4. ***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.

4. SATDBailiff Validation

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. An simplified example of a single SATD instance can be seen in Figure 2. The results of this analysis can be seen in Figure 9.

The 5 projects used for this validation are the same projects used by Maldonado to establish a prior dataset of SATD[8]. 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.

To perform the analysis, one of the authors was given the set of SATD instances and asked to locate the exact location of each of the SATD operations using the Github website. 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 on Github would result in the entire entry being incorrect. For entries that were not removed from the project, their existence in the terminal

Figure 9: SATDBailiff Manual Analysis Results

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**

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[6]. During validation, it was assumed that all binary classifications of source code comments as SATD were correct.

The results of the manual analysis (Figure 9) find SATDBailiff to have an accuracy of 0.885. It is infeasible to achieve a perfect measure of accuracy due to the inconsistent nature of open source development practices. 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 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 6 depicted as the red and green highlight used to represent a source code change. JGit uses the Myers[13] and the Histogram diff algorithm to produce edit scripts. SATDBailiff provides the ability to change between these two algorithms, and the Myers algorithm was used during performed manual validation. Both of these algorithms 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.

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

5. SATDBailiff Usage

This section describes the usage of SATDBailiff and its features.

5.1. Installation

The most up-to-date precompiled binaries can be found on the project's GitHub[7] or at the tool's website[8]. The project can be run using a java version 1.8 and is otherwise OS independent.

[6]https://bbchristians.github.io/SATDBailiff-site/data
[7]https://github.com/bbchristians/SATDBailiff
[8]https://bbchristians.github.io/SATDBailiff-site/

```
$ java -jar SATDBailiff.jar -d mySQL.properties -r repos.csv
Completed analyzing 78 diffs in 26,103ms (334.65ms/diff, 0 errors) - analogweb/core
3wks/thundr -- Mining SATD  (13.3%, 46/346, 0 errors) - e048736
```

Figure 10: Sample Runtime Snapshot

5.2. Usage

SATDBailiff can be used either through its CLI or API, and can be easily modified to support different output types and classification models.

5.2.1. CLI

The SATDBailiff CLI (Seen in Figure 1 as (1)) has two main inputs: A CSV file containing a list of Github repositories and a MySQL configuration file. The CLI has many other optional inputs to optimally configure the tool.

The CSV file containing repository information details which repositories will be mined by the tool, and where the mining will terminate. A terminal commit value can be added next to the repository URI to add an terminal point in time to which the tool can mine. This is done primarily to assure reproducibility between datasets mined at different times. If absent, the terminal commit value will default to the most recently available commit in the repository. The output format of the data also allows for a manual filtering of SATD operations by date. However, it should be noted that un-merged branches of the repository at certain timestamps are likely to cause difference between a pre- and post-execution commit filtering. Git credentials for private repositories can be added as a separate program argument.

By default, SATDBailiff outputs to a MySQL database. The tool intends for a MySQL database to be set up to receive the system's output. Configuration fields for this database must be supplied to the program to connect to the database. A description for the required fields, as well as the required schema for the database can be found in the Github repository.

Other runtime variables available within SATDBailiff include:

- File differencing algorithms available through JGit - currently only Histogram and Myers
- The Normalized Levenshtein distance threshold (between 0 and 1) described in Section **??**.
- A toggle for error display
- A help menu display

When run, SATDBailiff will display using the interface in Figure 10. This output includes a detailed description of the runtime duration of the tool, the number of commits differences, and a description of each entry that caused any sort of detectable error in the system, if errors display is toggled. If error display is disabled, then only the number of errors encountered will be displayed.

5.2.2. API

The tool is also available in the form of a Java API. All functionality that is achievable through the CLI is available as part of the API. The usability of SATDBailiff as an API is not well supported. Information about it can be found on the tool's website.

5.3. Interpreting Output

The released implementation of SATDBailiff outputs to a SQL database. The schema for the output is detailed in Figure 11.

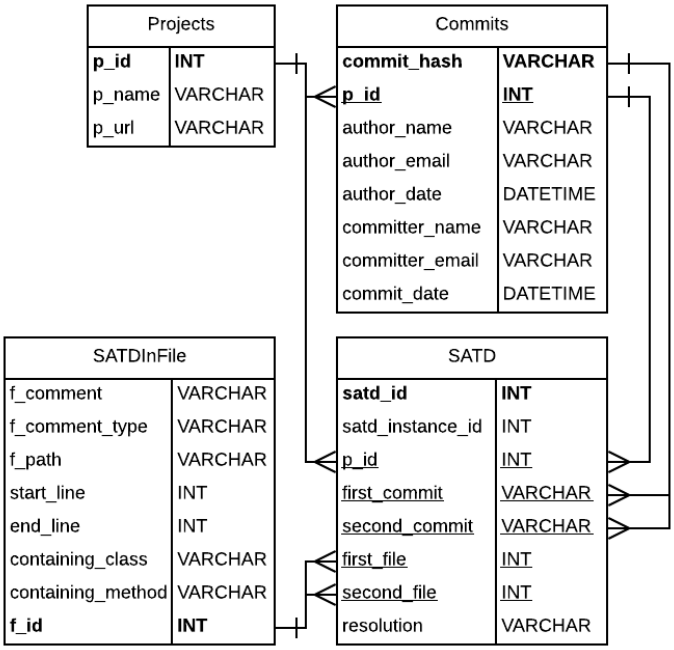

Figure 11: SATDBailiff output schema

6. Threats to Validity

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

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

An important potential threat relates to our manual classification. Since the manual verification of samples is a human intensive task and it is subject to personal bias, we mitigate this first by selecting SATD instances from an existing dataset. Then we performed the tracking of their existence one author. Only 200 samples of SATD were recorded and addressed to determine the accuracy of SATDBailiff. Ideally, this number would 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.

Another threat relates 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. Another threat concerns the generalization of SATD patterns used in this study. Since a method is considered holder of TD when a comment contains SATD, this may not generalize to other projects if they do not allow inline documentation (comments).

7. Conclusion and Future Work

In this paper, a preview of SATDBailiff was given and its benefits were discussed. The tool aims to offer an unmatched ability to extract SATD instances and the development operations upon them from Git repositories. This paper discussed the benefits that a high quality empirical history would have for the further study of SATD.

An acceptable level of accuracy was achieved with SATDBailiff, however there is still opportunity for improvement. In their nature, source code differencing tools may not always record modifications that reflect the true nature of a developer's intentions[14]. It is because of these inaccuracies that solving each

and every edge case is infeasible for the purpose of data generation. Some attempts to fix these edge-cases were made, however their large numbers and unpredictability made it a relatively futile task. To compensate, a list of known edge cases are included on the tool's website to detail known areas where inaccuracies may appear. As Edit Scripts are more reliably instantiated from these differences, more accurate detection of SATD-impacting operations will be possible.

Edit scripts were generated for this tool using the Histogram and Myers algorithms made available through JGit. Using a differencing tool like GumTree [14] or a hybrid approach [15] may produce more accurate results. However, GumTree currently does not offer the tracking of comment changes and does not plan on implementing that functionality[9]. Modification of the edit script generation methodology may also have positive impacts on the project's runtime which may currently be excessive for larger project.

Some other issues encountered include the handling of non-English language source comments. None of the projects used to validate SATDBailiff contained any known instances of these comments, however the larger dataset of 691 projects released alongside the tool does contain these instances. Ideally, a tool would be able to detect a non-English comment before classifying it as SATD, but no attempt was made to solve this issue as it only represents a small subset of the addressed projects.

References

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

[2] W. Cunningham, The wycash portfolio management system, ACM SIGPLAN OOPS Messenger 4 (2) (1992) 29–30.

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

[4] G. Bavota, B. Russo, A large-scale empirical study on self-admitted technical debt, in: Proceedings of the 13th International Conference on Mining Software Repositories, 2016, pp. 315–326.

[5] S. Wehaibi, E. Shihab, L. Guerrouj, Examining the impact of self-admitted technical debt on software quality, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, IEEE, 2016, pp. 179–188.

[6] E. d. S. Maldonado, E. Shihab, N. Tsantalis, Using natural language processing to automatically detect self-admitted technical debt, IEEE Transactions on Software Engineering 43 (11) (2017) 1044–1062.

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

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

[9] F. Zampetti, A. Serebrenik, 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), 2018, pp. 526–536.

[10] Q. Huang, E. Shihab, X. Xia, D. Lo, S. Li, Identifying self-admitted technical debt in open source projects using text mining, Empirical Software Engineering 23 (05 2017). doi:10.1007/s10664-017-9522-4.

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

[12] V. Frick, T. Grassauer, F. Beck, M. Pinzger, Generating accurate and compact edit scripts using tree differencing, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 264–274.

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

[14] J. Falleri, F. Morandat, X. Blanc, M. Martinez, M. 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, 2014, pp. 313–324.
URL http://doi.acm.org/10.1145/2642937.2642982

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

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