The Technical Debt Dataset

Valentina Lenarduzzi  
Tampere University, Finland
valentina.lenarduzzi@tuni.fi

Nyyti Saarimäki  
Tampere University, Finland
nyyti.saarimaki@tuni.fi

Davide Taibi  
Tampere University, Finland
davide.taibi@tuni.fi

1 INTRODUCTION

Companies commonly invest effort to improve the quality of their software by removing technical issues believed to impact software quality. Technical issues include any kind of information that can be derived from the source code or from the software process, such as usage of specific patterns, compliance with coding or documentation conventions, or architectural issues. If such issues are not fixed, they generate Technical Debt.

Technical Debt (TD) is a metaphor from the economic domain that refers to different software maintenance activities that are postponed in favor of the development of new features in order to get short-term payoff [5]. Just as in the case of financial debt, the additional cost will be paid later. The growth of TD commonly slows down the development process [5], [25].

Several Technical Debt measurement tools are available on the market (e.g., Better Code Hub1, Coverity Scan2, and SonarQube3). SonarQube is one of the tools most frequently adopted in research and industry [23] and it is adopted by more than 120K users worldwide3 to improve the maintainance of their systems [22].

Researchers have investigated several aspects of TD, trying to understand whether some specific issues can be considered harmful in certain contexts. For example, different research groups have investigated whether the presence of code smells increases change- or fault-proneness [29], [7], [14], [31], while other studies have investigated whether other code smells can impact maintenance effort [26], [38]. However, each research work is based on custom-built datasets, which are often not shared publicly or, in some cases, are not available anymore. As a result, the outcomes are not directly comparable, as they stem from different projects and contexts.

Moreover, in recent years, researchers have mainly focused their attention on code smells [10] and architectural smells [12]. However, the harmfulness of other technical issues detected by the tools commonly used in industry has not been investigated extensively. This is mainly due to the effort required to collect and analyze data with these tools.

To the best of our knowledge, only a few datasets provided TD data on open-source projects [32], [40] and a dataset containing fault information from Jira [30]. However, no dataset collects and shares information from different tools.

For this purpose, we have created the Technical Debt Dataset [36], which is a curated dataset containing measurement data from five tools executed on all commits of 33 projects from the Apache Software Foundation during the time frames reported in Table 1. The aim of this dataset is to enable researchers to work on a common set of data and thus compare their results.

The dataset was built by extracting the projects’ data and analyzing it using several tools. To get the data, the projects’ GitHub


REFERENCE


@INPROCEEDINGS{Lenarduzzi2019,  
author={Lenarduzzi, Valentina and and Saarim"{a}ki, Nyyti and Taibi, Davide},  
title={The Technical Debt Dataset},  
booktitle={The Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE’19)},  
year={2019},  
month={Sept},  
doi={10.1145/3345629.3345630},  
}
repositories were cloned, commit information was collected from the
git log using PyDriller [39], refactorings were classified using
Refactoring Miner [41], and fault information was obtained by ex-
tracting issues from the Jira issue tracker. After that, code quality
was inspected using two tools: Technical Debt items were analyzed
with SonarQube, and code smells [10] and anti-patterns [1] with
Ptidej [13]. In addition, the fault-inducing and -fixing commits
were identified by applying our implementation [33] of the SZZ

This dataset has been recently used by the authors for different
works [4], [19], [20], [37], [34] and could be used by researchers to
investigate various research questions regarding Technical Debt.

The data is easily accessible as we provide two alternative
ways to access the data, a set of CSV files and an SQLite database
file5, in order to facilitate queries on the data.

The main contributions of this paper are:
- The Technical Debt Dataset. A curated set of projects where
we analyzed all the commits from the time frames reported
in Table 1 with four tools
- Two data formats. CSV files and an SQLite database, to enable
researchers to efficiently query the data via SQL
- The data retrieval pipeline that produced this dataset.

The remainder of this paper is structured as follows. Section 2
describes the project selection strategy. Section 3 reports the tools
used to produce the dataset, while Section 4 describes the char-
acteristics extracted from the data. Section 5 reports on the data
production pipeline. Section 6 describes the data schema. Section 7
reports on the significance of this dataset. In Section 8, we report
how we plan to update the dataset, and Section 9 contains licence
information. Section 10 presents the threats to validity, and in Section
11, we draw conclusions.

2 PROJECT SELECTION

The selected projects had to fulfill all of the following criteria:
- Developed in Java
- Older than three years
- More than 500 commits
- More than 100 classes
- Usage of Jira issue tracking systems with at least 100 issues

Moreover, as recommended by Nagappan et al. [28], we also
tried to maximize diversity and representativeness by considering
a comparable number of projects with respect to project age, size,
and domain.

Based on these criteria, we selected 33 Java projects from the
Apache Software Foundation (ASF) repository6. This repository
includes some of the most widely used software solutions. The
available projects can be considered industrial and mature, due
to the strict review and inclusion process required by the ASF.
Moreover, the included projects have to keep on reviewing their
code and follow a strict quality process6.

In Table 1, we report the list of the 33 projects we considered
together with the number of analyzed commits, and the time frame,
number of the analyzed commits, and number of reported items
for each tool. The analysis time frame differs between the tools.
However, all projects have been analyzed using all of the tools for
the time frame of SonarQube / Ptidej.

3 TOOLS USED TO COLLECT THE DATA

In order to produce the dataset, we used four tools for analyzing
the project data. These tools will be described in the following
subsections.

3.1 PyDriller

PyDriller [39] is a Python framework meant for mining Git reposito-
ries. It provides easy extraction of information from a Git repository.
For example, the tool supports extraction of the commit message,
the number of developers, modifications, diffs, and the source code
of a commit. Moreover, PyDriller calculates structural metrics of
every file changed in a commit relying on Lizard7, a tool that can
analyze source code of different programming languages, both at
class and method level.

3.2 Ptidej

Ptidej (Pattern Trace Identification, Detection and Enhancement
in Java) is one of the most popular tools for detecting code smells
in research [23] and has been adopted by more than 200 research
works8. Ptidej can detect a total of 18 code smells [10] and anti-
patterns [1].

The tool is intended for use by research and development teams
for proposing and validating ideas, methods, and tools. The goal
is to improve the quality of systems implemented following the
object-oriented paradigm.

This goal is achieved by introducing the concepts design, and
architectural patterns. However, these are often not used (due to
time pressure or programmers’ unawareness) neither in the design
phase nor in the coding phase. Ptidej 5 can be freely downloaded
from the official online repository9. Since October 2014, the source
code of the Ptidej Tool Suite has been open and released under the
GNU Public License v2.

The Ptidej Tool Suite offers several tools for the assessment and
improvement of code quality. These include such tools as PADL
(Pattern and Abstract-level Description Language), which is a meta-
model for describing systems at different levels of abstraction; POM
(Primitives, Operations, Metrics), which a framework for comput-
ing several metrics on PADL models; and SAD, which stands for
Software Architectural Defects.

SAD specifies and identifies occurrences of code smells in sys-
tems modeled using the PADL tool. SAD requires the POM module
to identify code smells within the code; it employs a well-structured
procedure by means of rule cards. Each time detection of a specific
smell is requested, SAD imports its rule card and, employing the
POM module, calculates the software metrics.

3.3 Refactoring Miner

Refactoring Miner [41] is an open-source tool that classifies the
different refactorings in the history of Java projects. Refactoring

---

5SQLite Database: https://www.sqlite.org

6http://apache.org

7https://incubator.apache.org/policy/process.html


9http://www.ptidej.net/publications/

10Ptidej GitHub Repository: https://github.com/ptidejteam/v5.2
Miner takes as an input the list of commits, and returns a list of refactoring operations applied between consecutive commits.

Refactoring miner’s release 1.0.0 can detect 15 types of refactorings from different types of code elements (see Table 2).

Table 2: Refactorings detected by Refactoring Miner [41].

<table>
<thead>
<tr>
<th>Code Element</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>Change Package (move, rename, split)</td>
</tr>
<tr>
<td>Type</td>
<td>Move Class, Rename Class, Extract Superclass/Interface Extract Method, Inline Method Pull Up Method, Push Down Method Rename Method, Move Method Extract and Move Method</td>
</tr>
<tr>
<td>Method</td>
<td>Pull Up Field, Push Down Field Move Field</td>
</tr>
</tbody>
</table>

3.4 SonarQube

SonarQube is one of the most common open-source static code analysis tools for static quality analysis. It can be executed on premise or with the free cloud-based service on sonarcloud.io. SonarQube calculates several metrics such as number of lines of code and code complexity, and verifies the code’s compliance against a specific set of “coding rules”. In case the analyzed source code violates a coding rule or if a metric is outside a predefined threshold (also named “quality gate”), SonarQube generates an “issue”. The time needed to remove these issues (remediation effort) is used to calculate the remediation cost and the Technical Debt.

Each rule is classified as being related to Reliability, Maintainability, or Security of the code. Reliability rules, also named “bugs”, create TD issues that “represent something wrong in the code” and that will soon be reflected in a bug. Maintainability rules or “code smells” are considered as “maintainability-related issues” in the code that decrease code readability and modifiability. It is important to note that the term “code smells” adopted in SonarQube does not refer to the commonly known code smells defined by Fowler et al. [10], but to a different set of rules. SonarQube claims that zero false-positive issues are expected from the Reliability and Maintainability rules, while Security issues may contain some false-positives.

---

Footnote: SonarQube Rules: https://docs.sonarqube.org/display/SONAR/Rules

Last Access: June 2019

Author Version - Download the dataset files from GitHub: https://github.com/clowee/The-Technical-Debt-Dataset

---

The Technical Debt Dataset
PROMISE’19, September 18, 2019, Recife, Brazil
SonarQube also classifies these rules into five severity levels:

- **BLOCKER**: "Bug with a high probability to impact the behavior of the application in production", such as memory leaks and unclosed JDBC connections.
- **CRITICAL**: "Either a bug with a low probability to impact the behavior of the application in production or an issue which represents a security flaw". Examples are empty catch blocks and SQL injection.
- **MAJOR**: "Quality flaw which can highly impact the developer productivity", such as uncovered piece of code, duplicated blocks, or unused parameters.
- **MINOR**: "Quality flaw which can slightly impact the developer productivity". Examples: lines should not be too long and "switch" statements should have at least three cases.
- **INFO**: "Neither a bug nor a quality flaw, just a finding."

SonarQube also recommends immediately reviewing blocker and critical issues.

SonarQube has separate sets of rules for the most common development languages such as Java, Python, C++, and JavaScript. SonarQube version 7.5 includes more than 500 rules for Java. The complete list of rules is available online but can also be found in the file "sonar_rules.csv" of the Technical Debt Dataset.

### 3.5 SZZ Algorithm

OpenSZZ [33] is our free implementation of the SZZ algorithm. The SZZ [15] algorithm tries to identify the fault-inducing commits from a project’s version history. The algorithm was developed in 2005 and has since been adopted in more than 200 empirical studies [11], [6].

The algorithm is based on Git’s blame/annotate feature and assumes that the fault-inducing commit of a fault is known. Usually this is done by combining data from an issue tracker and from Git’s log command.

The algorithm consists of three steps. An example is provided in Figure 1.

The first step is to identify the fault-fixing commits, i.e., commits that are known to have fixed a bug. For example, in step 1 of the figure, we can see that fault AMBARI-17618 was fixed from file Resource.java in commit #e8bfdb.

The second step is to identify the changes that fixed the fault. The algorithm inspects the files that were changed in the fault-fixing commit. From these files, the algorithm identifies the changes related to updating a data structure, i.e., bug-fixing activities. In the example, step 2 shows the differences between commit #e8bfdb and its predecessor (#300a7e) in the Resource.java file. In this case, in order to fix the bug, the data structure at line 188 was changed.

Therefore, SZZ identifies the changes that introduced bug AMBARI-17618 through the history of the source configuration management system (GitHub).

The last step determines in which commit the code that caused the fault was introduced. This is done by using Git’s annotate/blame tool. In the example, step 3 shows commit #a2d7c9 being flagged as a potential bug-introducing change by SZZ.

Each commit is tagged with the information about the retrieved fault from the SZZ algorithm. Commits can be of one of three fault types: inducing (I), fixing (F), or not related to faults (N).

It is important to notice that we applied the SZZ algorithm as is, without applying any filtering. It could be possible to apply different filters, considering different contextual information provided in the dataset.

### 4 COLLECTED MEASURES

The Technical Debt Dataset includes information on the following characteristics:

- **Commit Information**: Information about each commit obtainable through Git’s log command. The information includes, for example, the commit hash, date, and message.
- **Refactorings**: List of refactorings applied in each commit, obtained from Refactoring Miner.
- **Code Quality**: List of detected issues related to the code quality of a commit. These consist of style violations and other low-level technical issues monitored by SonarQube, and the detected anti-patterns [1] and code smells [10] such as Large Class and Long Method.
- **Jira issues**: List of all detected issues from the project’s issue tracker.
- **Fault-Inducing and Fault-Fixing Commits**: For each fixed fault in a project, the commit where the fault was created (fault-inducing) and the commit where the fault was fixed (fault-fixing) are determined.

In the following subsections, we will introduce the different characteristics in more depth and describe how we extracted them from the data.

### 4.1 Commit and Changes Information

We extracted the commit information from the Git repositories using PyDriller [39]. This tool was used instead of connecting directly to the GitHub API for maintainability reasons. Connecting to the GitHub API or parsing the Git Logs would require developing a custom script, whereas PyDriller currently allows extracting the same information without major effort.

Thanks to PyDriller, we saved all the information available in the GitLog, including information on each file modified in each commit. Moreover, we collected information on the refactoring applied in each commit by executing Refactoring Miner.

### 4.2 Code Quality

As in the case of software quality, we collected three sets of information:

- **Software Metrics**: 30 different software metrics, including:
  - size-related metrics, such as number of classes, number of packages, and number of lines of code
  - complexity metrics, such as Cyclomatic Complexity [27] and cognitive complexity [3]
  - test coverage, including test coverage as well as the number of lines not covered by tests
  - duplications, including number of duplicated lines and duplicated files.
• Technical Debt information detected by SonarQube, including:
  – Issues detected in the projects, classified as "bugs", "code smells", and "security vulnerabilities" (see Section 3.4)
  – Technical Debt remediation time, including the time for refactoring "bugs", SonarQube "code smells", and "security vulnerabilities" issues.

The complete list of metrics collected in the dataset together with their description is reported in Table 3, while the collected anti-patterns and code smells are described in Table 4. For reasons of space, the list of SonarQube issues and rules is not reported here, but it is available in the dataset.

4.3 Jira issues
For each project, we extracted all issues from the Jira issue tracker. The information includes, for example, the creation and resolution date of the issue, issue key, and the type and priority of the issue. Descriptions for the extracted fields can be found from Jira’s documentations.

4.4 Fault-Inducing and Fault-Fixing Commits
For this step, we first extracted the faults from Jira issues. This was possible as the ASF policy states that all faults reported in the issue tracker have to be tagged as "Bug".

Faults are commonly discovered after the code has already been committed. Therefore, in order to ensure that we would find the vast majority of faults related to commits, we analyzed the projects from their first commit until the end of 2015, considering all the faults raised until the end of March 2018. We believe that the vast majority of the faults introduced in the commits from the time frames reported in Table 1 should have been discovered after more than two years.

We tried to identify the fault-inducing and -fixing commits for all the faults retrieved from Jira. This was done by applying the SZZ algorithm [15] to the selected projects. We used our own, publicly available implementation of the algorithm [33] for the analysis.

As stated in subsection 3.5, the algorithm assumes that the fault-fixing commit is known for every fault. The selected ASF projects fulfill this assumption, as the ASF policy requires that in all fault-fixing commits the developer must report the fault-id from the issue tracker in the commit message. Therefore, we relied on this mechanism to map the commits from which developers had removed faults.

5 DATASET PRODUCTION
We performed the following tasks for each project independently. We first cloned the repository. Then we iterated each commit using PyDriller [39].

For each commit, we performed the following steps:
• Retrieval of the commit information from the GitLog using PyDriller
• Classification of the refactorings using Refactoring Miner
• Analysis of the code with SonarQube using the default quality model (Sonar way)
• Analysis of the code with Ptidej

This set of tasks was the most time-consuming part, since each commit took an average of four minutes to be analyzed with SonarQube and 50 seconds with Ptidej. As the open-source license of SonarQube does not allow analyzing multiple projects in parallel, we analyzed the projects sequentially. However, sequentiality was also required because SonarQube calculates evolutionary metrics based on the changes performed in each commit. SonarQube and Ptidej were executed in parallel.

These three steps took a total of 220 days on a Ubuntu server with 128GB RAM and 50 cores. However, because of the limitation of SonarQube’s open-source license, the SonarQube analysis used only one core and an average of 12GB RAM, while Ptidej used one core and an additional 6GB RAM on average. It is important to note that not all the commits were analyzed correctly, as SonarQube or Ptidej raised exceptions when a commit did not compile properly. However, we stored the information about all the commits in the

While the script analyzing the commits was running, we extracted the fault information from Jira by means of the Jira APIs. The dataset includes all the issues with the status “closed”. This task took less than two days of execution time on a virtual machine with 4 cores and 8GB RAM.

At the end of the execution of both scripts, we executed our SZZ implementation [33] to identify the fault-inducing commits. This task took ten days of execution time on the same machine used for the commit analysis (SonarQube and all the other services were stopped in order to enable SZZ to use as many resources as possible).

After the extraction of all the data, we performed a data-sanity check to evaluate possible inconsistencies, including duplicated and unexpected values.

Finally, we imported the data into an SQLite database to facilitate queries on the data.

6 THE DATA SCHEMA

The Entity Relationship schema of the dataset is presented in Figure 2. Regarding the fields in the tables, the fields needed to distinguish a row are underlined and “...” means that the table has more fields than presented in the figure. Looking at the fields of the tables, it is evident that the database has not been normalized. However, this is intentional and data is often replicated in different tables to speed up queries by avoiding heavy use of joins between tables.

The tables are briefly described below and the data of each a table can be found in the respective identically named CSV file in the Technical Debt Dataset.

- Table PROJECTS contains the links to the GitHub repository and the Jira issue tracker of each project.
- Table SONAR_MEASURES contains the different measures SonarQube analyses from the commits. They contain, for example, number of code lines in the commit, the code complexity, and number of functions. Note, that the table contains information about the code in the commit rather than the commit itself.
- Table COMMITS reports the commit information retrieved from the git log, including the commit hash, the commit message, the author and committer name, date and timezone, the list of branches that contain the commit, in_main_branch (true if the commit is in the main branch), merge (true if the commit is a merge commit), parents (list of the commit parents).
- Table COMMITS_CHANGES contains the changes performed in each commit, including the old path of the file (can be _None_ if the file is deleted, the new path of the file (can be _None_ if the file is deleted), the type of the change reported in the git Log (Added, Deleted, Modified, or Renamed), the diff of the file as Git presents it, the number of lines added, the number of lines removed, the number of lines of code of the file (nloc), the Cyclomatic Complexity of the file (calculated with Lizard), the number of tokens of the file, the list of methods of the file (might be empty if the programming language is not supported or if the file is not a source code file).
- Table JIRA_ISSUES contains Jira issues for the analyzed projects. Fields include the key (the issue-id used by Jira to identify each issue), the creation and resolution dates, the priority assigned by the developers, and many others.
- Table FAULT_INDUCING_COMMITS reports the results from the execution of the SZZ algorithm. Each row in the table is a fault retrieved from Jira. The fields include information on the fault-inducing commits (FaultInducingCommitHash and FaultInducingTimestamp) and on the fault-fixing commits (FaultFixingCommitHash and FaultFixingTimestamp). If the fault-fixing commit could not be determined, the related fields are left empty.
- Table REFACTORYING_MINER reports the list of refactoring activities applied in the repositories. The table contains the project, commit hash, the type of refactoring applied, and the details of the refactoring activity.
- Table SONAR_ISSUES lists all of the SonarQube issues as well as the anti-patterns and code smells detected by Ptidej. The value of field squid of the issues detected by SonarQube starts with either the prefix squid: or common-java:; while the code smells and anti-patterns detected by Ptidej are identified using the prefix code_smells:. Each row represents one issue, reporting which rule, code smell, or anti-pattern was violated, where it was found, when it was created, and the fixing date if it has been fixed. In addition to these features, several other attributes about the issue are reported as well.
- Table SONAR_RULES lists the rules monitored by SonarQube. For each rule, the feature description provides a short description. In addition, the table contains, for example, the type and severity assigned by SonarQube for the rule.

6.1 How to use the dataset

The dataset is accessible online [36] and it is provided both as a set of CSV files and as an SQLite database. In both cases, the same data is provided; it is structured as shown in Figure 2.

Each CSV file contains the data of one table. The SQLite database is provided as a .db file and can be opened with a SQLite browser such as SQLiteStudio[^15]. After opening, the data can be queried using SQL by selecting “Tools” and “Open SQL Editor”. Data can also be exported as a CSV file by selecting “Tools” and “Export” and then exporting a whole table or an SQL query.

Listing 1 provides an example on how to use data from different tables. The example combines data from tables SONAR_MEASURES and SONAR_ISSUES in order to get the number of SonarQube issues in a project.

7 SIGNIFICANCE

Source code analysis has been attracting more attention from research and industry, mainly because of the availability of tools like SonarQube. More than 100K companies are currently using SonarQube and are therefore considering the information, metrics, and issues reported by it.

[^15]: https://www.sqlite.org/index.html
[^15]: SQLiteStudio: https://sqlitestudio.pl
Figure 2: Entity Relationship Diagram of the dataset.

**Listing 1:** An SQL query for getting the number of "Bug" and "Vulnerability" SonarQube issues in a project.

```sql
SELECT SONAR_ISSUES.projectID, count(SONAR_ISSUES.projectID) AS numberOfBugIssues, numberOfVulnerabilityIssues
FROM SONAR_ISSUES
JOIN (SELECT SONAR_ISSUES.projectID, count(SONAR_ISSUES.projectID) AS numberOfVulnerabilityIssues
      FROM SONAR_ISSUES
      WHERE TYPE = 'VULNERABILITY'
      GROUP BY SONAR_ISSUES.projectID) AS V ON SONAR_ISSUES.projectID = V.projectID
WHERE TYPE = 'BUG'
GROUP BY SONAR_ISSUES.projectID
```

**Listing 2:** An SQL query for getting the commit message and author of all the commits fixing a SonarQube issue.

```sql
SELECT GIT_COMMITS.commitMessage, GIT_COMMITS.author
FROM SZZ_FAULT_INDUCING_COMMITS
JOIN GIT_COMMITS
ON SZZ_FAULT_INDUCING_COMMITS.faultFixingCommitHash = GIT_COMMITS.commitHash
```

Author Version - Download the dataset files from GitHub: https://github.com/clowee/The-Technical-Debt-Dataset
As for the selected projects, a number of studies have been carried out about projects in the Apache Software Foundation ecosystem [9], [8], [2], [31], but they mainly focus on different languages or on different types of issues.

The areas that can benefit from our Technical Debt Dataset are statistical machine learning, natural language processing on source code, but also empirical studies on Technical Debt.

Possible future work could also be related to the investigation of the harmfulness of the different SonarQube issues, including their fault- and change-proneness. This area has already been widely investigated considering code smells [14], [17], [38], [31] but researchers have never considered TD issues detected by SonarQube from the point of view of fault- and change-proneness.

Our dataset will allow researchers to compare the effect of code smells with those of SonarQube violations, as the data include the analysis of code smells with Ptidej and the SonarQube issues on the same projects.

Different research questions could be investigated with the Technical Debt Dataset. Besides the possibility to replicate the existing works on code smells, it would be possible to investigate many other research questions, such as:

- How are the projects maintained?
- How are faults and Technical Debt related?
- How do different Technical Debt issues co-evolve?
- Do these patterns differ between projects?
- Which refactoring activity introduces more technical debt?
- Which Technical Debt issue is more prone to refactoring?

8 UPDATE

Open-source projects are continuously evolving and thus our dataset will also need to be updated regularly. We provide a GitHub repository to allow other researchers to contribute by analyzing other projects [36].

We will provide new releases of the dataset whenever a new project has been analyzed completely.

9 LICENSE

The Technical Debt Dataset has been developed only for research purposes. It includes the historical analysis of each public repository, including commit messages, timestamps, author names, and email addresses. Information from GitHub is stored in accordance with GitHub Terms of Service (GHTS), which explicitly allow extracting and redistributing public information for research purposes.

The Technical Debt Dataset is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International license.

10 THREATS TO VALIDITY

Researchers should always consider potential threats to the validity of their research.

Regarding the data collection process and the traditional trade-off between being up-to-date and curation [6], we preferred emphasizing the curation of the dataset rather than providing a limited up-to-date amount of data. Moreover, we relied on PyDriller to iterate over the different commits and to retrieve the commit information. We are aware that both Ptidej and SonarQube might analyze the code incorrectly under some conditions. Moreover, regarding SonarQube, we adopted the out-of-the-box “Sonar way” model. Although SonarQube recommends customizing the set of rules used [17], practitioners are reluctant to customize it and commonly rely on the standard rule set [42]. Querying the SonarQube public instance APIs reveals that more than 98% of the public projects (14,732 projects out of 14,957) use the “Sonar way” rule set. Moreover, it is important to notice that the SonarQube remediation time can be overestimated [35]. Therefore, future studies considering the remediation time should consider this threat.

Another possible tool-related threat concerns our implementation of SZZ. The implementation of the SZZ algorithm has been manually validated in [33]. However, we are aware that in some cases, the SZZ algorithm might not have identified fault-inducing commits correctly because of the limitations of the line-based diff provided by Git, and also because in some cases bugs can be fixed by modifying code in another location than the lines that induced them. Moreover, we rely on the developers that must report the ID of the fault to the commits that fixed it to identify the fault-fixing commit, but we are aware that, even if the ASF is enforcing this approach, some faults could be not correctly labeled.

Another important threat is related to the generalization of the dataset. We selected the list of projects based on different criteria (see Section 2). However, even though the projects are widely used in industry, they cannot possibly represent the whole open-source ecosystem. Moreover, since the dataset does not include industry projects, we cannot make any speculation on closed-source projects.

11 CONCLUSION

In this paper, we presented the Technical Debt Dataset [36]. It is the largest source code dataset analyzing Java projects with different tools, including tools widely used in industry and research.

We described the pipeline we adopted to collect the data. The Technical Debt Dataset is made available both as an SQLite database file and as CSV files in order to facilitate queries on the data. The metadata is carefully provided as part of the download.

The creation of the dataset took more than 200 days due to the license limitation of SonarQube, whose open-source license does not allow running analyses in parallel.

The results will allow researchers to perform various studies on a common dataset, comparing their results and concentrating more on their research goals instead of on data collection. The current research on code smells could be extended in the future to SonarQube issues, enabling researchers to investigate different quality aspects, including technical debt, maintenance effort, fault-proneness of different metrics, code smells, and several other aspects.

We are planning to extend this dataset in the future applying different measurement tools to the same projects. Moreover, we are planning to investigate further datasets such as [50] that could be

---

[17] SonarQube Quality Profiles: https://docs.sonarqube.org/display/SONAR/Quality+Profiles
[18] Last Access: June 2019
[19] https://docs.sonarqube.org/display/DEV/API+Basics
used to complement the data of our dataset. Moreover, we are plan-
ing to extend our previous work with this version of the dataset, applying NLP on requirements and Jira issues [18], and analyzing the relationship between technical debt and coupling [4]. Moreover we are planning to analyze differences between technical debt is-sues in open source projects and in SMES [21] and to use the results as baseline for continuous quality monitoring approaches [16],[24].

REFERENCES


[21] Nyyti Saarimäki, Valentina Lenarduzzi, and Davide Taibi. 2019. On the diffuse-


neering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).


https://docs.sonarqube.org/latest/user-guide/metric-definitions/
Table 3: The software metrics collected in the dataset. Complete descriptions can be found in the SonarQube documentation

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of classes</td>
<td>Number of classes (including nested classes, interfaces, enums and annotations)</td>
</tr>
<tr>
<td>Number of files</td>
<td>Number of files.</td>
</tr>
<tr>
<td>Lines</td>
<td>Number of physical lines (number of carriage returns).</td>
</tr>
<tr>
<td>Ncloc</td>
<td>Also known as Effective Lines of Code (eLOC). Number of physical lines that contain at least one character which is neither a whitespace nor a tabulation nor part of a comment.</td>
</tr>
<tr>
<td>Ncloc language distribution</td>
<td>Non Commenting Lines of Code Distributed By Language</td>
</tr>
<tr>
<td>Number of classes and interfaces</td>
<td>Number of Java classes and Java interfaces.</td>
</tr>
<tr>
<td>Missing package info</td>
<td>Missing package-info.java file (used to generate package-level documentation)</td>
</tr>
<tr>
<td>Package</td>
<td>Number of packages</td>
</tr>
<tr>
<td>Statements</td>
<td>Number of statements</td>
</tr>
<tr>
<td>Number of directories</td>
<td>Number of directories in the project, also including directories not containing code (e.g., images, other files...).</td>
</tr>
<tr>
<td>Number of functions</td>
<td>Number of functions. Depending on the language, a function is either a function or a method or a paragraph.</td>
</tr>
<tr>
<td>Number of comment lines</td>
<td>&quot;Number of lines containing either comment or commented-out code. Non-significant comment lines (empty comment lines, comment lines containing only special characters, etc.) do not increase the number of comment lines.&quot;</td>
</tr>
<tr>
<td>Number of comment lines density</td>
<td>Density of comment lines = Comment lines / (Lines of code + Comment lines) * 100</td>
</tr>
</tbody>
</table>

Complexity

| Complexity                               | It is the Cyclomatic Complexity calculated based on the number of paths through the code. Whenever the control flow of a function splits, the complexity counter gets incremented by one. Each function has a minimum complexity of 1. This calculation varies slightly by language because keywords and functionalities do. |
| Class complexity                         | Complexity average by class                                                  |
| Function complexity                      | Complexity average by method                                                 |
| Function complexity distribution         | Distribution of method complexity                                           |
| File complexity distribution             | Distribution of complexity per class                                         |
| Cognitive complexity                     | How hard it is to understand the code’s control flow.                       |
| Package dependency cycles                | Number of package dependency cycles                                         |

Test coverage

| Coverage                                 | It is a mix of Line coverage and Condition coverage. Its goal is to provide an even more accurate answer to the following question: How much of the source code has been covered by the unit tests? |
| Lines to cover                           | Number of lines of code which could be covered by unit tests (for example, blank lines or full comments lines are not considered as lines to cover). |
| Line coverage                            | On a given line of code, Line coverage simply answers the following question: Has this line of code been executed during the execution of the unit tests? |
| Uncovered lines                          | Number of lines of code which are not covered by unit tests.                 |

Duplication

| Duplicated lines                         | Number of lines involved in duplications                                    |
| Duplicated blocks                        | Number of duplicated blocks of lines.                                       |
| Duplicated files                         | Number of files involved in duplications.                                   |
| Duplicated lines density                 | (duplicated lines / lines) * 100                                            |

Table 4: Code smells and anti-patterns detected in our projects. Descriptions are adapted from [10] and [1].

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplicated code</td>
<td>The same code reused in different locations.</td>
</tr>
<tr>
<td>Blob</td>
<td>A big class (usually a singleton) that has dependencies with data contained in other data classes. It could monopolize several system operations.</td>
</tr>
<tr>
<td>Class data should be private</td>
<td>Class publicly exposing variables.</td>
</tr>
<tr>
<td>Cyclomatic complexity</td>
<td>Also referred to as McCabe’s Cyclomatic Complexity. Refers to methods that should be simplified since they have too many independent execution paths.</td>
</tr>
<tr>
<td>Downcasting</td>
<td>A cast to a derived class.</td>
</tr>
<tr>
<td>Excessive use of literals</td>
<td>Too many literal variables embedded into the code instead of being declared as constants.</td>
</tr>
<tr>
<td>Feature envy</td>
<td>An object gets at the fields of another object to perform some sort of computation or make a decision, rather than asking the object to do the computation itself.</td>
</tr>
<tr>
<td>Functional decomposition</td>
<td>A class with too many functionalities, which needs to be broken down into smaller and simpler classes.</td>
</tr>
<tr>
<td>God Class</td>
<td>A huge class implementing different responsibilities.</td>
</tr>
<tr>
<td>Inappropriate intimacy</td>
<td>A method that has too much intimate knowledge of the inner workings, inner data... of another class or method.</td>
</tr>
<tr>
<td>Large class</td>
<td>A class that is too big.</td>
</tr>
<tr>
<td>Lazy class/Freeloader</td>
<td>A class with very limited functionalities.</td>
</tr>
<tr>
<td>Orphan variable or constant class</td>
<td>A class containing variables used in other classes.</td>
</tr>
<tr>
<td>Refused request</td>
<td>A children class that never uses the inherited methods.</td>
</tr>
<tr>
<td>Spaghetti code</td>
<td>A class with a very complex control flow.</td>
</tr>
<tr>
<td>Speculative generality</td>
<td>An abstract class with a very limited number of children who are not using its methods.</td>
</tr>
<tr>
<td>Swiss army knife</td>
<td>A class that is providing many services for different purposes, such as a Swiss army knife.</td>
</tr>
<tr>
<td>Tradition breaker</td>
<td>A class that, despite inheriting from another class, does not fully extend the parent class and has no subclasses.</td>
</tr>
<tr>
<td>Excessively long identifiers</td>
<td>Variable names that are too long and do not respect the naming conventions.</td>
</tr>
<tr>
<td>Excessively short identifiers</td>
<td>Variable names that are too short and do not allow understanding the purpose of the variable.</td>
</tr>
<tr>
<td>Excessive return of data</td>
<td>A method that returns more than what is needed from the calling methods.</td>
</tr>
<tr>
<td>Long method</td>
<td>A method that is too long.</td>
</tr>
<tr>
<td>Too many parameters</td>
<td>Methods with too many parameters, which usually become hard to read and to test.</td>
</tr>
</tbody>
</table>