university of groningen
faculty of science and engineering

Visualizing Self-Admitted Technical Debt in a Web-Based Application

Bachelor's Project

26 August 2021

Student: M. van Ittersum

Daily supervisor: Y. Li

Primary supervisor: prof. dr. ir. P. Avgeriou

Secondary supervisor: dr. M.A.M. Soliman

Abstract

During software development, problems can come up that would take too long to solve for maintainers at that point of time. Situations like these could take too long to be resolved. Code maintainers can choose to take a quick solution, with a trade-off being that it would take time and effort in the future to properly resolve the problem. Here, we call the time and effort required to rework these solutions Technical Debt (TD). In the cases that the maintainer is aware of creating TD, a comment is left behind on the location of the TD in the source code, or an issue is created in the project's issue tracker. We call this Self-Admitted Technical Debt (SATD).

In this study, the objective was to create a web-based application that visualizes the SATD in a project, combining the SATD identified in source code comments and in issue trackers. We approached this objective by splitting this process into three parts: extracting SATD in source code comments, extracting SATD in issues and then combining the data in a dashboard. We used two classification methods described in previous studies. To evaluate our system, we asked a group of developers to use this system and to give a score on the visualization of SATD in comments and issues apart and combined, focusing on the *accuracy* that the system has in detecting SATD, *awareness* the system creates of the SATD in a project and the *effectiveness* the system has on the developers' actions that are taken in the future. We found that the system performs best in creating more *awareness* and combining the data from both issues and comments.

Contents

1 Introduction

Software development is a process that consists of writing, testing, documenting and maintaining software. Looking at the process from a more global scope, the process can be considered a constant problem analysis and solving cycle. This cycle of finding problems, analysing them and trying to come up with a solution can be very time and energy consuming and greatly depends on the available knowledge of the ones involved. Often during this process in shortage of time, developers look for a more simple, short-term solution, much like a shortcut [1]. As for that moment, the problem seems solved and the developers can move on to a different pressing issue. However, in the long-term, the choices made at that moment can have consequences and require more time and effort to be solved properly. In software development, this is referred to as *Technical Debt* (TD).

We can identify Technical Debt by deriving the definition from the term's name. When we look at debt, whether the debt is financial, technical or even social, we look at a situation in which action is owed from one to another. With TD this is much the same. If we look at the above mentioned explanation, we see a situation in which the developer has to later on put in effort to find a long-term solution to the issue. The term TD however is broader than just choosing shortcuts during the software development cycle, as it affects every stage of the cycle. Can show up without the need of code changes, as it can be caused by external parties, such as system upgrades [7].

Staying aware and keeping track of TD and code quality is a good method to increase the overall quality of the software development cycle and the end product [11]. As many automated tools[1] nowadays more and more keep track of the quality of the code by judging the structure and comparing against common bugs [15], developers often have to explain code to improve readability. Commenting code leads to readers having much less to figure out by themselves, thus increases readability and understanding of the code [2], [3]. As readability and understanding improves, a sense of code quality is formed.

1.1 Self-Admitted Technical Debt

While leaving comments in the code to explain code and improve quality is one application, one can also leave comments to admit that the code in that section is considered TD. Here, we refer to this as *Self-Admitted Technical Debt* (SATD). These comments are left when developers find their current solution not optimal and suggest to come back later [10]. Comments as these give information about what is missing or what needs to be done to resolve this technical debt issue.

Another accepted method of self-admitting that the solution is considered TD, is to leave an entry in the issue tracker of the project [12]. Issue trackers are used to keep track of issues and features that need to be resolved [19]. Development teams use these trackers to coordinate development. Leaving an SATD issue refers back to the location of the code where TD exists. These issues can be planned for resolution by development teams later on.

1.2 Classification

When talking about TD, we can refer to many types of issues that can come up. For example, a feature can be missing; a function can be badly written; a test's coverage should be expanded; or documentation should be added. Researches have identified many types of TD, each with their own indications [8].

[1]Tools such as *SonarQube* and *Code Climate*.

1.3 Objective

This study's objective is to create a system that can show developers insights into their work about SATD. We want to make developers aware of their own indicated TD, forgotten in time or hidden in complexes.

We divide this objective into three research questions. The first and second research question are about implementing data collection and creating a visualization system. Here, we focus on what data to extract and show. The third question is about the effectiveness of our implementation. Here, we test our system against open source projects and ask volunteers about their opinion.

1.4 Study Summary

For achieving our objective, we created a web-based application that is easily deployable using Docker, separating some processes into their own containers and services. This application tracks Git repositories and JIRA projects and indexes the required data into its own database. For the source code, we extract the source code comments. We classify the textual data as either SATD or non-SATD. Finally, we present the user with the statistics of the SATD in their projects.

To find out the effectiveness of this application, we surveyed a group of 6 developers. In the survey, we focused on three points, the *accuracy* that the system has in detecting SATD, *awareness* the system creates of the SATD in a project and the *effectiveness* the system has on the developers' actions that are taken in the future. We also asked about the effectiveness of only visualizing SATD in the source code, SATD in the issue trackers and a combined visualization system. We found that developers agreed that the application increased the *awareness* the most, with a combined system being the most effective.

Project	**Comment**
Eclipse	// TODO this is such a hack it is silly
Chromium	// Unsafe; should error
ArgoUML	// FIXME: This is such a gross hack...
Apache	/* Ugly, but what else? */

Table 1: Example of comments indicating SATD (by Potdar *et al.* [9])

2 Related work

In this study, we extend on work in three main subjects in a practical way. The first two subjects are about the types of SATD and how to collect. The last subject is about visualizing SATD. Therefor, in this section, we divide up the work into those three subjects.

2.1 Working with SATD in code

In an exploratory study by Potdar *et al.* [9], they researched four different open source projects on the appearance and removal of SATD. Here, they used the source code of these open source projects to extract the code comments. They show a simple pipe-lined process to extract and identify these comments. In this study, they also show an example of comments that indicate SATD, as shown in table 1. Furthermore, Potdar and Shihab showed in their study that on a file level, the extracted comments were classified in a range between 2.4% and 31.0% as SATD.

Shown in a study by Maldonado *et al.* [10], five main types of TD out of the list described by Alves *et al.* [8] were found in source code comments. These types are *Design debt*, *Requirement debt*, *Defect debt* and *Documentation debt.* In this study, five open source software projects are analysed on source code comments. The comments that are considered SATD are classified on the five different types of SATD. After classifying the source code comments, the study showed that the distribution of the types of SATD in all five projects were similar. Here, the majority of the analysed comments that were classified as SATD were of the type *Design debt*, having a range of 42% to 85% of the SATD-identified comments. The type *Requirement debt* generally came second, with in only one case slightly overtaking *Design debt* and in the majority of the cases having a significant lower percentage than the largest type. This type had a range of 5% to 45%. The remaining three types had in all projects a significant lower percentage, being below 10%.

An extension on detection of SATD in source code comments was done in a study by Maldonado *et al.* [14] by introducing Natural Language Processing (NLP) into the classification process. First, comments are manually classified as *Design debt*, *Requirement debt* or simply not SATD. For the two types, some examples are shown.

> *"TODO: - This method is too complex, lets break it up"* - [from ArgoUml]
>
> *"//TODO no methods yet for getClassname"* - [from Apache Ant]

Maldonado *et al.* show that the first comment is an example of *Design debt*, as it clearly refers to the current design of the code. The second comment is an example of *Requirement debt*, as the comment refers to missing code. After manually classifying comments, they use the comments as a training dataset for a Stanford Classifier. Applying the classifier on 10 open source projects, they found that certain keywords were very effective in classifying the types of debt. For example, they found that 'hack' and 'workaround' refers to *Design debt*, while keywords as 'todo' and 'needed' refer more to *Requirement debt.*

Our work will extend on these studies by using four of the five types of TD used by Maldonado *et al.* [10] to classify source code comments, skipping *Defect debt* and adding *Test debt* [8]. Here, we implement the NLP classification method described by Maldonado *et al.* [14] in all five TD types.

2.2 Working with SATD in issues

In a study by Bellomo *et al.* [12], TD detection is extended towards issue trackers, presenting a method for classifying issues in types of TD. They conducted the study on four projects, two open source and two unidentified government IT projects. They manually classified issues as TD, using certain conditions of the state of the issue.

Automation of the process is described by Dai *et al.* [13]. Here, they described TD types similar to the types used in detecting SATD in source code comments. Six types were used, adding *UI Debt* and *Architecture debt* (excluding *Documentation debt* and *Test debt*). The automation was done very similar to the process described by Maldonado *et al.* [14], using the Naive Bayes classifier to learn and predict TD.

Work by Li *et al.* [18] takes the amount of TD types up to eight. In their study, they examined 500 issues out of 2 open source Apache projects, Hadoop and Camel. These issues contained 152 SATD items. They showed that an issue can have more than one SATD type linked, with one case being classified with four types.

We follow up on these works by implementing the automated issue classification system and combining it together with the source code extractor.

2.3 Visualization

Some simple methods showing TD are described by Power [6]. In their study, they give a couple of options. The first option is to show the amount of capacity of a team it takes up to tackle TD. Anther method is to track the amount of TD over time, indicating whether code quality is being neglected over the course of a couple of releases. They state that keeping track of the amount of TD in a team can be essential in managing the team's capacity efficiently and helping the stakeholders understand. In our study, we extend on the method of showing TD over time and showing what kind of TD is present in a project at a time.

Bohnet *et al.* [4] proposes a method of visualizing software statistics using 3D maps. The source code files are organized by modular hierarchy, setting the 2D location. The third dimension is used to show certain information about a file, such as lines of code or McCabe complexity. They studied two large, industrially developed systems: JBoss and Blender. In their case study, they show that a visualization method as this can effectively show hotspots with lower quality code. Our visualization system makes use of a form of the software map, replacing the 3D aspect with colors.

3 Case Study Design

In this case study the primary objective is to find an effective way visualize SATD in a web-based system. When analysing the objective, we came up with three research questions.

3.1 Gathering data

Before being able to visualize SATD, we first need to extract and process data. Therefore, we proposed the first research question:

RQ$_\mathbf{1}$: What data is required for visualizing SATD?

Motivation

We state this question to go more in depth towards analysing and finding good visualization methods, by first analysing what data we actually want to use. Having to process big JIRA issue repositories and Git source code repositories, we want to come up with a method that efficiently extracts the right data out of there. We do this to prevent unnecessary storage and CPU usage.

Approach

To answer this question, we first look at what kind of data is available and what can we extract out of this data.

For analysing the source code comments, we looked at the processes described in the studies described in section 2.1. Here, the studies show the main process of extracting source code comments and applying classification. As we are analysing the state of a Git repository, we looked further into storing revisions of files and linking source code comments to commits to find out at what point they appeared and at what point they were removed.

For analysing issues in a JIRA issue tracker, we first retrieve the comments stored on a JIRA server. After that, we use a similar way as shown by the study by Dai *et al.* [13], classifying issues automatically. We also keep track of when issues are added and resolved.

3.2 Visualization methods

Once we know what data we are using, we can look further at what how we would like to visualize the data. We research the possibilities and state the following question:

RQ$_\mathbf{2}$: How can we visualize the collected SATD data?

Motivation

Here we analyse the different kinds of visualization of SATD, so we can look at what can be seen as effective and useful. We know that showing SATD in an insightful way can improve a team's capacity planning and help business side [6]. Thus, finding a good combination of SATD visualization methods helps out towards the main objective.

Approach

We answer this question by an developmental approach, building up the system based on the data we extracted. Our main approach towards visualizing is to show the location of the comments indicating SATD, the amount of different types of SATD and the amount of SATD over time.

3.3 Effectiveness

After we find different methods of visualizing SATD, we want to test the effectiveness of each method.

RQ_3: How effective is the visualization system in assisting developers?

Motivation

This question is to evaluate the answer of RQ_2 using real world observations. We want to evaluate the results to to test whether the visualization system is useful for developers. This is what we want as our main objective.

Approach

A qualitative and quantitative survey was used as the approach, asking the participants to give a score, but also elaborate more with open questions. For this survey, a small group of developers were able to volunteer. This consisted out of a team of four people and two external source. Before answering the survey, we first had an online video conference in which we demonstrated the system. We showed some the system working on open source software data, explaining the available data. Afterwards, the participants imported their own work into the system. We discussed the data after it was processed and then proceeded to the survey.

In our survey, we focused on three types of questions. The first type of questions are about the *accuracy* of the data. We asked the participants to give a score to the accuracy of the system and how much they agreed with the types of SATD that were classified.

The second questions were about the *awareness* created by the system. We asked this to see if the participants showed an increase of awareness of the state of their work, in particular the amount of SATD in their work. The questions also asked the participants to give a score, but now asking to score their own increased/decreased awareness.

The last questions were about the *effectiveness* of the system on the participants. With this, we mean the system triggered a response in the participants to change their priorities into solving problems visualized by the system.

4 Implementation

zIn our Case Study Design, we found three main processes in visualizing SATD: *Collecting data*, *classifying* and *visualization*. Furthermore, we divided up the first process, collecting data, into two processes that combine their data. The first part consists of reading a Git repository and filtering out source code comments. The second part is to connect to a JIRA server and read out the issues from the project.

4.1 Service infrastructure

To implement this system, we first looked at how we want to have our infrastructure. Being a web-based application, we looked at some simple web server infrastructures, as we did not want to make this application more complex that necessary. Here, we took the base as a simple combination of a database and a web server.

As visualization was important, we wanted to split up the back-end and front-end. This also has the benefit that, when designing the service correctly, we would have an improved user experience. To combine these together into one web service, we use the NGINX as a reverse proxy[2]. NGINX lets us to direct user traffic to the right service, based on the request.

After deciding on our main web infrastructure, we also added a service for analysing comments and issues. This service uses a machine learning model to classify these items as SATD. We decided to split this off of the back-end service, as that allows us to use an entirely different language and environment.

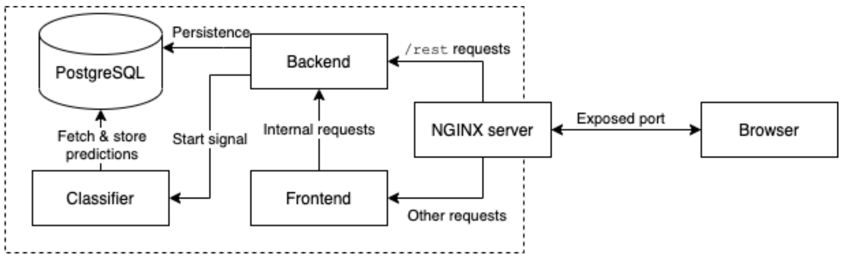


Figure 1: The service infrastructure

The complete infrastructure that we decided on is displayed on figure 1. Everything in the dotted line is encapsulated and not able to talk to the outside world, only the NGINX container is. We also decided on using a PostgreSQL database server[3], taking advantage of their advanced data types and recursive queries.

4.1.1 Back-end service

As the web has become more and more popular over the years, the availability of different kinds of web servers has become enormous, each with their pros and cons. When choosing the programming language and type of web server, we had some requirements. The technology should be mature enough, should be able to handle large data sets and we should be familiar with the technology in some way to speed up development.

[2]As explained by NGINX: *What Is a Reverse Proxy Server?*
[3]For more information, see: `https://www.postgresql.org/`

Here, we decided that the Spring Boot framework[4], using the Java programming language, was a solid candidate, as we were very familiar with the framework and it is considered quite mature. Moreover, the framework takes care of handling database connections and allows for some multi-threaded processing.

Next to that, the module is also responsible for scanning Git repositories for comments and JIRA boards for issues, and storing the collected data into the database (as explained in section 4.3.1 and section 4.3.2).

4.1.2 Front-end service

Our front-end module is responsible for presenting the user with an interface that the user can interact with. It connects back to the back-end module by sending HTTP requests to the back-end module. The module is built with React library and uses the Next.JS framework. We chose this combination for two reasons: familiarity and there are a considered amount of open source libraries available for this configuration that help with visualizing data.

4.1.3 Classifier service

Our analyser/classifier module [21] is built in Python 3 and uses Google's Tensorflow library. This module's responsibility is to read the database for issues and comments that are not classified. After retrieving each comment and issue one by one, it tries to predict the type of SATD against a pre-learned model. If the data is considered SATD, it will gather the keywords that would likely have caused the classification. After that, it would store the prediction back into the database. This module contains a simple Flask-server, allowing the back-end module to easily communicate with this module.

4.1.4 Virtualization

Finally, deploying this application as a whole, we virtualize every module in containers. For this, we use Docker. Docker allows us to separate every service into their own container and makes more efficient usage of the system's resources [16]. Combining this with Docker Compose[5] we can quickly develop and deploy the application.

4.2 Data structures

When working with larger projects, containing megabytes of files and thousands of commits, we were required to come up with efficiently picking, storing and loading data. Here, we used some data structures that we need to elaborate on.

4.2.1 File snapshots

A snapshot is the state of the file structure at the point of time of a commit. Storing all files for each commit is going to be very storage consuming and slow (shown on figure 2). Thus, we divide up snapshots into two types for data storage efficiency. The first is a *key snapshot.* This snapshot contains references to every file present at that point. The second type is a *partial snapshot.* This type contains a reference to a previous snapshot (parental snapshot), using that as a base. Then the snapshot contains references to files added and removed after the parental snapshot. This method as shown on figure 3 is based on video compression algorithms.

[4]For more information, see: `https://spring.io/`

[5]A tool that lets you configure multiple Docker containers together.

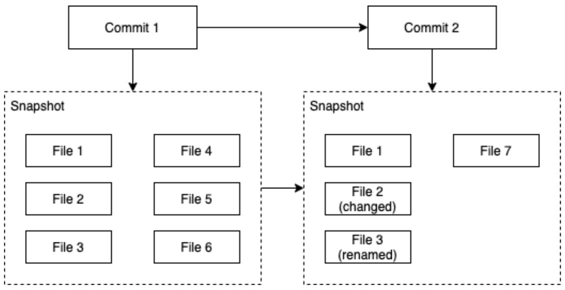


Figure 2: Data structure when storing snapshots entirely

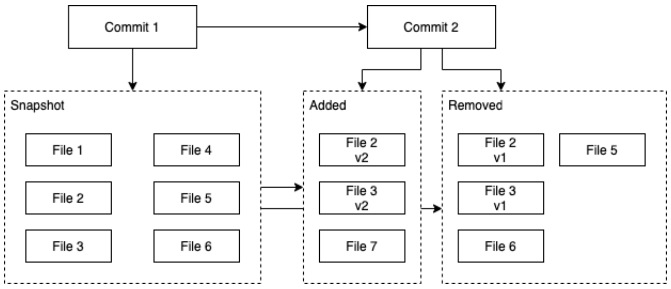


Figure 3: Data structure when storing snapshots using key and partial snapshots

4.3 Processes

As stated before, we have three main processes. In figure 4 a flowchart of these processes is shown, going from Git commit data and issues to visualized SATD statistics.

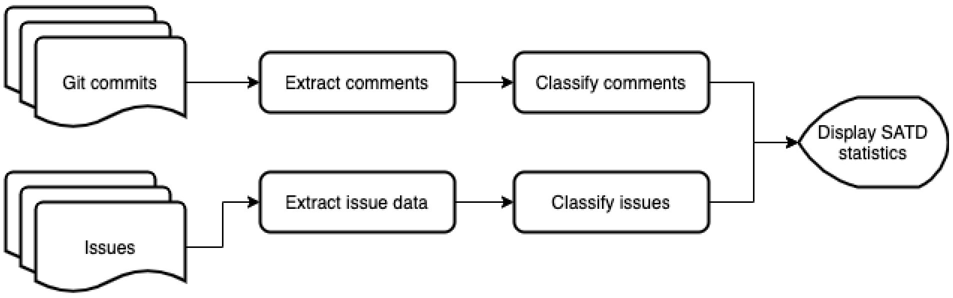


Figure 4: Flowchart of the system

4.3.1 Scanning Git repositories

This process takes advantage of the JGit project for storing the files and their revisions. This project is Java implementation of the Git version control system [20]. For extracting comments out of the source code files, we use ANTLR (ANother Tool for Language Recognition). Using ANTLR, we wrote simple grammars based on types of comments that we could encounter.

Step 1 We store information of the repository provided by the user in the database. When storing that data, we retrieve the ID of the entry that we inserted in the database. Using this ID, we create a directory and use JGit to clone the repository into that directory.

Step 2 We now retrieve the available branches from the locally stored repository. The user selects which branch they would like to track. The local repository checks out the branch and we collect the information of the commits in that branch.

Step 3 After the commit data is stored, the user has the option of selecting the commits for creating a *snapshot* (explained in section 4.2.1). After this step, we end up with a data structure looking as shown in figure 5.

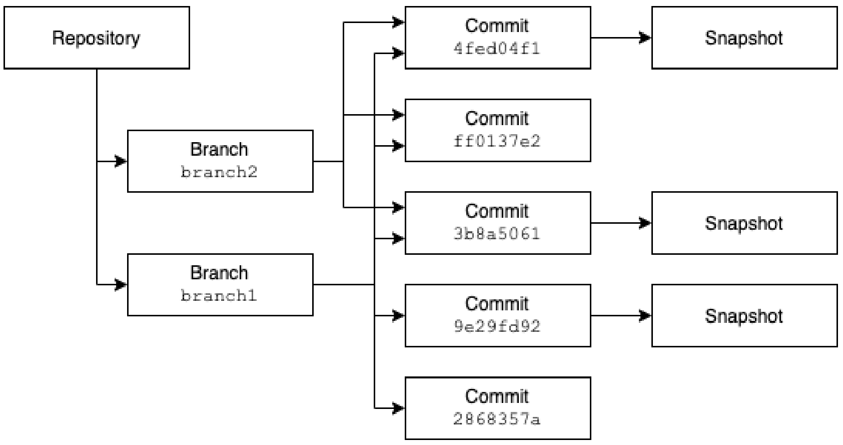


Figure 5: Data structure of a Git repository

Step 4 After all files and their revisions are stored, we extract the comments out of source code files. Here, we look at the file's extension and choose one or more ANTLR grammar to extract the comments.

Step 5 Having the comments extracted, we look for duplicates. Here we use the hash code property of the Java-object. We compare the hash code of the content of the found comments to hash codes stored in the database and other found comments. If we find duplicates, we only store one instance of the comment, but multiple instances linking the file and the comment together.

4.3.2 Scanning JIRA projects

Scanning a JIRA project requires interfacing with the JIRA server's REST API. This API is documented and have open source clients available. For this project, we use the Jira REST Java Client (JRJC).

Step 1 The user is asked to fill in the URL to the JIRA server. If required, the user can provide login information.

Step 2 After connecting successfully to the JIRA server, we provide the user with the available projects. The user can select one here.

Step 3 The user is tasked with selecting the range of issues that they would like to analyse.

Step 4 We collect the information of the issues and its revisions. This information is provided by the JIRA API.

4.3.3 Classifying data

As mentioned in the section 4.1.3, we use a different environment, completely encapsulated from the other processes to classify texts [21]. The process is shown in figure 6. In this process, we constantly ask the database if there are any process-able data entries left and run that data against our models. The resulted information is stored back into the database.

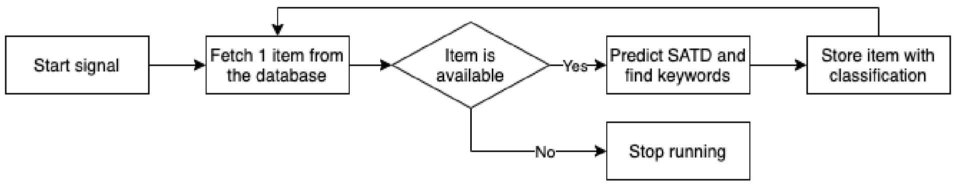

Start signal
Fetch 1 item from the database
Item is available
Yes
Predict SATD and find keywords
Store item with classification
No
Stop running

Figure 6: The process of classifying items

4.3.4 Visualization of SATD comments

We use four methods to visualize SATD in source code comments in a Git repository.

SATD types pie chart We show how much of each SATD type is present at a certain commit.

SATD over time We show a graph with the amount of SATD over time.

SATD heatmap We use a sunburst graph to show using colors where the most SATD is located in a project.

File browser We show the SATD data in a file browser, showing for each file and directory how much SATD is present.

Keywords For each SATD comment, we show what keywords have caused the classification.

4.3.5 Visualization of SATD issues

Types of SATD We show a pie chart visualizing the accumulated types of SATD found in the project.

SATD per issue status We show a pia chart visualizing the total accumulated SATD per status of the issue. Usually, statusses consist of an issue being open, in progress, closed/resolved.

SATD per issue type We show a pie chart visualizing the total accumulated SATD per issue type. Usually, the issue types consist of bugs, improvement, user stories and more.

5 Results

In this section, we briefly go over the results of RQ_1 and RQ_2, which are the results of the implementation. After that, we go over the results of RQ_3, which consists of the results of the survey.

5.1 RQ$_1$: What data is required for visualizing SATD?

For this question, we primarily focused on the classification methods available described in previous studies [14], [21].

For visualizing SATD out of the source code, we track Git repositories. We require one or more branches to be selected, which will be tracked for commits. We index the files of these commits in snapshots (explained in section 4.2.1). Comments are extracted from these snapshots and linked back to each file revision that contains the comments. Finally, we classify the comment as either SATD, giving it one of the types and extracting the keywords indicating SATD, or classifying the comment as non-SATD. This data can be accumulated, showing amounts of SATD at one time and amounts of SATD over time.

For visualizing SATD out of issue trackers, we track JIRA projects. In this process, we extract all existing issues and store the summary and description. These text fields are classified as SATD with the same method as the we use in classifying source code comments. However, we do use a different model for both classification methods.

5.2 RQ$_2$: How can we visualize the collected SATD data?

In our study, we focused on three main visualization methods. These are: showing SATD over time, showing the amount of SATD per type at a time and showing the location of SATD. For the SATD over time, we used a line graph, calculating the SATD for 15 different points in a given time interval. For SATD per type, we used pie charts. We showed amounts per SATD type, comment type, issue type and issue status. Finally, for showing the location of the SATD in the source code, we created a file browser that shows per file and directory the amount of SATD. In addition to that, we show a heat map of SATD using a sunburst graph.

5.3 RQ$_3$: How effective is the visualization system in assisting developers?

For the last research question, we look at the results of the survey given over 6 developers. We evaluate these answers by dividing the questions over the three main focus points of the survey. We also look at how the 2 processes scored individually and combined. This gives us insights into how the processes scored apart from each other and insights whether these processes combined improve the effectiveness. In figure 7 the average scores on each focus point are graphed. The quantitative results of the survey are available in appendix A.

Evaluating the results of the *accuracy* focus point, we see that the visualization of SATD in source code comments has a score of 4.25 out of 5. Discussing the accuracy of the SATD visualization, the participants noticed that only a minor cases of inaccurate classification of the comments, observing the classifier being mislead by certain keywords. In the visualization of JIRA issues, the participants gave an average score of 4 out of 5. Here, the participants noticed that classifying the summary and the description of the issue gave in minor cases different results. They noticed that once again, certain keywords were causing the classifier into being mislead. The combined accuracy scored 4.33 out of 5 points. The participants considered overall accuracy quite accurate good, with the system effectively being able to pick out SATD in their projects.

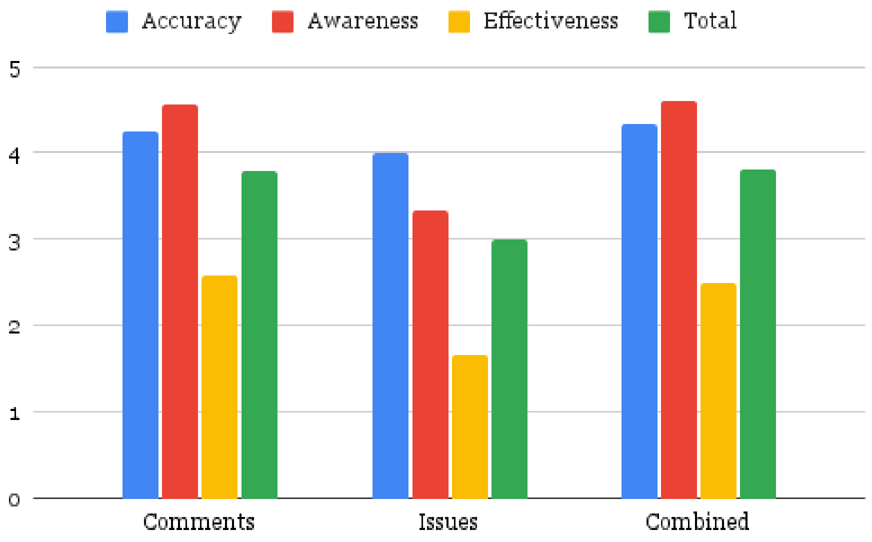


Figure 7: The average scores of the survey results

Moving over towards the *awareness* of SATD created by the system, the participants quickly indicated that they found points of interest in their work that were forgotten about in the source code comments. The score of the visualization system of SATD in comments here scored on average a 4.55 out of 5. The participants noted that visualizing the SATD over time was not as helpful when only using this system for a short time, but said that if they were able to track the state of their work in a longer time span, they hypothesized that the information would be useful in planning the team's capacity. The score of the awareness by visualizing SATD in issues was lower, scoring an average of 3.33. Here the participants argued that issues already have their own classification system (in JIRA you can assign a type to an issue) and issues were much more sorted and organized. The overall awareness scored the highest, with 4.61 out of 5 points.

The final questions focused on the *effectiveness* of the system towards developers. The participants gave an average score of 2.58 out of 5 when looking at the comments and only 1.66 out of 5 for the issues. Overall, this section scored the lowest, with a combined average of 2.5. The participants mostly argued that the SATD visualized was already in their work for a longer time, thus having not a high priority of rethinking the code. They also stated that their issue tracker was sorted by priority, having to focus their team's capacity on higher priority issues first.

If we look at the combined scores of these focus points, we see that both visualization systems combined score higher than that the systems score apart from each other, with an average of 3.81 compared to 3.80 and 3.00 for the comments and issues visualization system respectively.

6 Discussion

In this section we discuss the implementation and the evaluation of the system by going back over the results, giving some implications for researchers and developers and making some recommendations for improvements that could be made.

6.1 Implementation discussion

The implementation of the visualization system focused on three things: visualizing SATD in comments, visualizing SATD in issues and combining that data. We used three kinds of visualization here, showing where SATD was located, what kind of SATD is present and how much SATD changed over time.

The system produces very similar statistics of SATD in a project as previous studies have. When extracting source code comments and classifying these, we get very similar percentages of accumulated items per type of SATD as in the study by Maldonado *et al.* [10]. When working with issue trackers, we did find more SATD in issues than that the study by Li *et al.* [18]. We believe that our method of classifying issues is less accurate, as it only uses the summary and description in the classification process.

Our approach does not tackle measuring the TD in a project, as we only classify an item as a type of SATD. TD can be measured in different kinds of risk that the existence of the TD can have in a project [5]. Using this property of TD, we can prioritize certain items over others, indicating developers to focus more on what is important. When testing the system on open source projects, we found many items that have existed in the project for long time spans, in some instances over 10 years. While still admitting TD, developers would likely focus on more important instances.

Another feature that the system is still lacking, is that the data of an issue tracker and the source code is never combined. The data of both sources could be combined into visualizing the total amount of SATD over time, as well as showing the total amount of items per type of SATD. For researchers, this could create evidence of there being correlation between data found in the source code and the data found in the issue trackers. For developers, this could improve the awareness of SATD overall in projects.

For future studies, the effects of pull requests could also be taken into account. A process could be ran on extracting source code comments in the changes that the pull request would provide, showing an increase or decrease in SATD per pull request. A pull request based workflow links pull requests back to issues that the pull request is intended to resolve [17]. An example here could be that a requirement debt could only be partially resolved in a pull request, implementing a feature not in its entirety. A decrease of debt could be noted, but with still one active requirement debt item.

6.2 Survey results discussion

In our survey over 6 software developers, we found that the system can help developers becoming more aware of the state of their work. The primary findings of the survey results are that the combined data seems more effective in helping developers, giving an overall higher score. Participants did seem to be more aware of SATD in their work when using the system, but did not choose to change their priorities.

The system's accuracy could be improved by allowing users to correct a classification. We can send these corrections back to the classification service, training the model with the given corrections. Future items would be classified better as the system learns more and more.

A suggested way of improving the system's generated awareness is by creating an overview of active SATD items. These items could have a life span property, showing the user that the item has either been recently added, or has lived on for a long time.

7 Threats to Validity

7.1 Threat to Reliability

For the system, we used a model that was pre-trained on a set of source code comments and issues. In some cases, the classifier failed to properly predict the type of SATD. This was caused by the system being mislead by certain keywords. To minimize this threat, we suggest to implement manual correction. This should train this model to produce better results in the system.

7.2 Threat to Internal Validity

The survey was don over a small group of six developers, with four coming out of the same team. The small amount of participants forms a threat as it may not represent the opinion of most developers. The four developers in the same team looked at the same data and gave similar answers.

7.3 Threat to Construct Validity

The survey consisted of questions about accuracy, awareness and effectiveness. These points were used towards proving the effectiveness of the system, as these points seemed most important when dealing with TD. It is unknown if these points accurately prove the effectiveness of the system over a longer time period.

8 Conclusion and Future Work

The main objective of this project was to effectively visualize SATD to developers. Briefly summarizing, we found that the system did perform well for developers, helping the developers in the process of maintaining code.

For SATD in source code comments, participants in the survey gave an average score of 3.80 out of 5 points, with giving a high score to creating awareness. The system provided insights into what kind of SATD is present, where it was located and how much SATD changes over time. We saw that developers, before using the system, did not know or had forgotten certain comments that indicated SATD. By visualizing, this became more obvious and developers were able to keep track of these comments.

For SATD in issues, the average given score was lower, with only 3.00 out of 5 points. Here, we conclude that only tracking issues and their state does not seem that useful. This can be mostly explained by that issue trackers, JIRA and similar trackers, have already built in a manual issue classification system, as well as add-ons for time management.

The combined data provided developers a good method of viewing the state of their projects. We saw improvement over all three focus points, while averaging the two focus points resulted in lower scores. Developers were able to lay connections with certain items and group items together. Thus, we can conclude that the system does provide an effective method in visualizing SATD.

As shown in the Discussion, many improvements could be made to the system. These improvements focus on increasing the amount of data and the effectiveness of the data on developers. We would also like study the results of using a visualized system over a longer time span and a bigger group of participants. The state of projects could be measured over time, keeping track of the amount of SATD, the amount of issues resolved and the size of the different releases.

References

[1] M. Fowler, *TechnicalDebt*, Oct. 2003. [Online]. Available: `https://www.martinfowler.com/bliki/TechnicalDebt.html` (visited on 08/12/2021).

[2] D. Spinellis, *Code quality: the open source perspective*, 1st. Addison-Wesley, Apr. 2006, p. 608, ISBN: 978-0321166074.

[3] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: Bugs or bad comments?*/," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 145–158, Oct. 2007. DOI: `10.1145/1323293.1294276`. [Online]. Available: `https://dl.acm.org/doi/abs/10.1145/1323293.1294276`.

[4] J. Bohnet and J. Döllner, "Monitoring code quality and development activity by software maps," in *Proceedings - International Conference on Software Engineering*, 2011, pp. 9–16, ISBN: 9781450305860. DOI: `10.1145/1985362.1985365`.

[5] C. Seaman and Y. Guo, "Measuring and Monitoring Technical Debt," *Advances in Computers*, vol. 82, pp. 25–46, Jan. 2011, ISSN: 0065-2458. DOI: `10.1016/B978-0-12-385512-1.00002-5`.

[6] K. Power, "Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options," in *2013 4th International Workshop on Managing Technical Debt, MTD 2013 - Proceedings*, 2013, pp. 28–31, ISBN: 9781467364430. DOI: `10.1109/MTD.2013.6608675`.

[7] E. Wolff and S. Johann, *Managing Technical Debt*, May 2013. [Online]. Available: `https://www.infoq.com/articles/managing-technical-debt/`.

[8] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," *Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt, MTD 2014*, pp. 1–7, Dec. 2014. DOI: `10.1109/MTD.2014.9`.

[9] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, pp. 91–100, 2014. DOI: `10.1109/ICSME.2014.31`.

[10] E. D. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical Debt," *2015 IEEE 7th International Workshop on Managing Technical Debt, MTD 2015 - Proceedings*, pp. 9–15, 2015. DOI: `10.1109/MTD.2015.7332619`.

[11] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering," *Dagstuhl Reports*, vol. 6, no. 4, pp. 110–138, 2016, ISSN: 2192-5283. DOI: `10.4230/DAGREP.6.4.110`.

[12] S. Bellomo, R. L. Nord, I. Ozkaya, and M. Popeck, "Got technical debt? Surfacing elusive technical debt in issue trackers," *Proceedings - 13th Working Conference on Mining Software Repositories, MSR 2016*, pp. 327–338, May 2016. DOI: `10.1145/2901739.2901754`.

[13] K. Dai and P. Kruchten, "Detecting technical debt through issue trackers," in *CEUR Workshop Proceedings*, vol. 2017, 2017, pp. 59–65.

[14] S. Maldonado, E. Shihab, and N. Tsantalis, "Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, 2017.

[15] C. Vassallo, A. Bacchelli, F. Palomba, and H. C. Gall, "Continuous code quality: Are we (really) doing that?" *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 790–795, Sep. 2018. DOI: `10.1145/3238147.3240729`.

[16] S. Yegulalp, “Why you should use Docker and containers | InfoWorld,” *InfoWorld*, Oct. 2018. [Online]. Available: `https://www.infoworld.com/article/3310941/why-you-should-use-docker-and-containers.html`.

[17] M. Ortu, M. Marchesi, and R. Tonelli, “Empirical analysis of affect of merged issues on GitHub,” in *Proceedings - 2019 IEEE/ACM 4th International Workshop on Emotion Awareness in Software Engineering, SEmotion 2019*, Institute of Electrical and Electronics Engineers Inc., May 2019, pp. 46–48, ISBN: 9781728122809. DOI: `10.1109/SEmotion.2019.00017`.

[18] Y. Li, M. Soliman, and P. Avgeriou, “Identification and Remediation of Self-Admitted Technical Debt in Issue Trackers,” *Proceedings - 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020*, pp. 495–503, 2020. DOI: `10.1109/SEAA51224.2020.00083`. arXiv: `2007.01568`.

[19] Red Hat, *Getting started with an issue tracker*, 2020. [Online]. Available: `https://www.redhat.com/en/resources/getting-started-with-an-issue-tracker-FAQ` (visited on 08/18/2021).

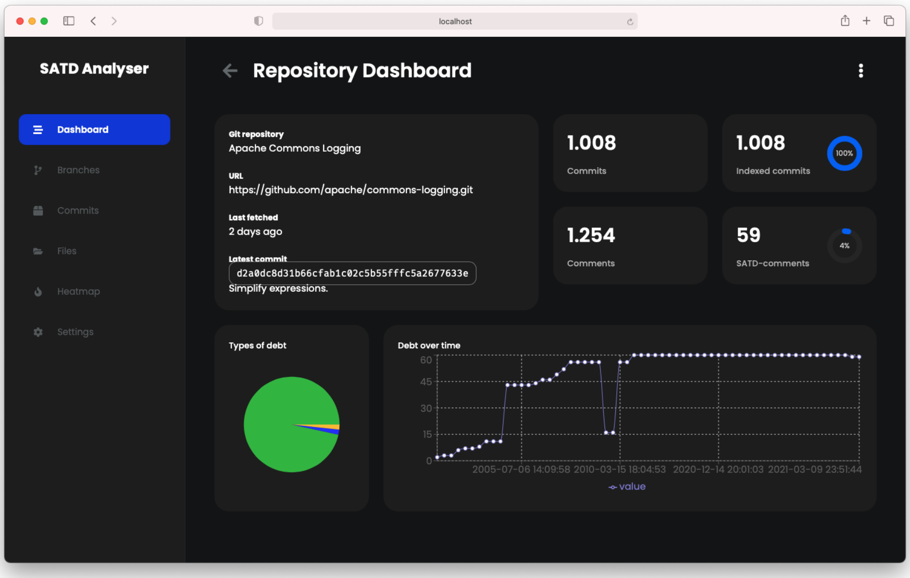
[20] Eclipse Foundation, *JGit | The Eclipse Foundation*, 2021. [Online]. Available: `https://www.eclipse.org/jgit/` (visited on 08/13/2021).

[21] Y. Li, M. Soliman, and P. Avgeriou, “Identifying self-admitted technical debt in issue tracking systems using machine learning,” unpublished, 2021.

A Survey results

Question	**1 2 3 4 5**	**Mean**
Overall		
Considering the overall results, how accurate was the detected SATD?	0 0 1 4 1	4
..., did you agree with the different types of SATD found?	0 0 0 2 4	4.67
..., did you (re)discover new SATD?	0 0 0 2 4	4.67
..., did you gain insights into the current state the project?	0 0 0 1 5	4.83
..., did you gain useful insights into the state of the project over time?	0 0 1 2 3	4.33
..., did you find high priority issues in your project?	1 4 0 1 0	2.17
..., did you or are you going to take (new) actions based on the results?	0 2 3 1 0	2.83
SATD in comments		
Considering the results in the comment analyser, how accurate was the detected SATD?	0 0 1 4 1	4
..., did you agree with the different types of SATD found?	0 0 0 3 3	4.5
..., did you (re)discover new SATD?	0 0 0 2 4	4.67
..., did you gain insights into the current state the project?	0 0 0 1 5	4.83
..., did you gain useful insights into the state of the project over time?	0 0 1 3 2	4.17
..., did you find high priority issues in your project?	2 3 0 1 0	2
..., did you or are you going to take (new) actions based on the results?	0 1 3 2 0	3.17
SATD in issues		
Considering the results in the issue analyser, how accurate was the detected SATD?	0 0 0 1 5	4.83
..., did you agree with the different types of SATD found?	0 1 3 2 0	3.17
..., did you (re)discover new SATD?	0 1 2 3 0	3.33
..., did you gain insights into the current state the project?	0 0 3 2 1	3.67
..., did you gain useful insights into the state of the project over time?	0 0 6 0 0	3
..., did you find high priority issues in your project?	1 3 2 0 0	2.17
..., did you or are you going to take (new) actions based on the results?	5 1 0 0 0	1.17

B Application screenshots



Git repository
Apache Commons Logging

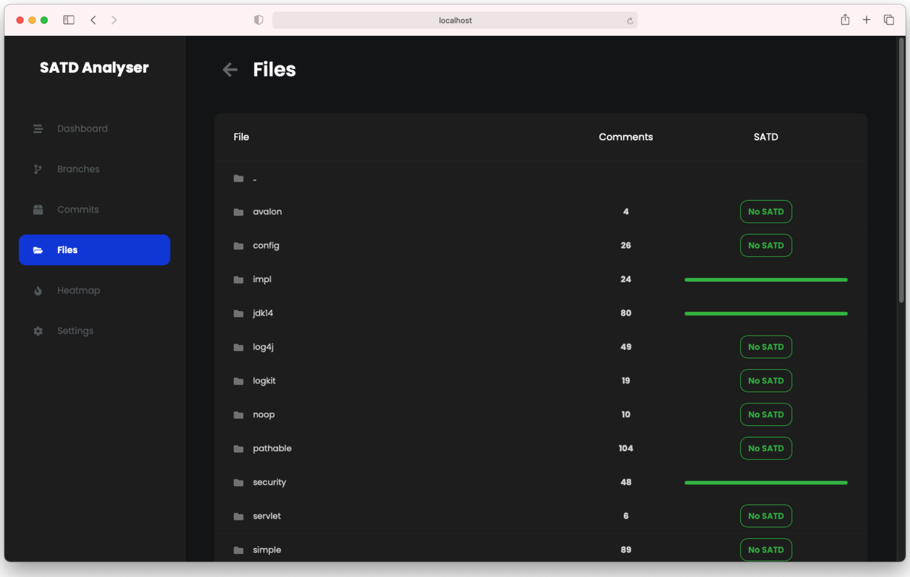
URL
https://github.com/apache/commons-logging.git

Last fetched
2 days ago

Latest commit
`d2a0dc8d31b66cfab1c02c5b55fffc5a2677633e`
Simplify expressions.

1.008	Commits
1.008	Indexed commits (100%)
1.254	Comments
59	SATD-comments (4%)

Figure 8: The dashboard for a Git repository



File	Comments	SATD
..		
avalon	4	No SATD
config	26	No SATD
impl	24	
jdk14	80	
log4j	49	No SATD
logkit	19	No SATD
noop	10	No SATD
pathable	104	No SATD
security	48	
servlet	6	No SATD
simple	89	No SATD

Figure 9: The file browser

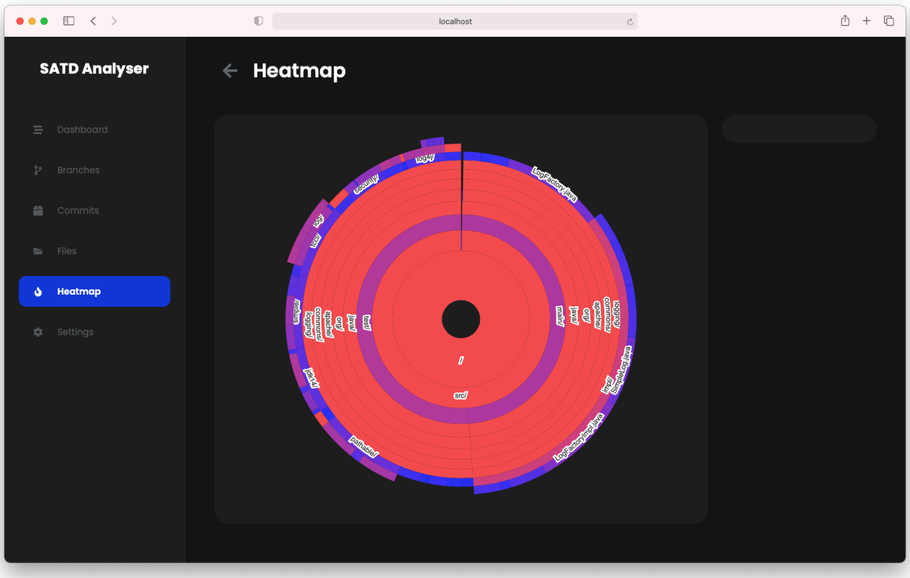
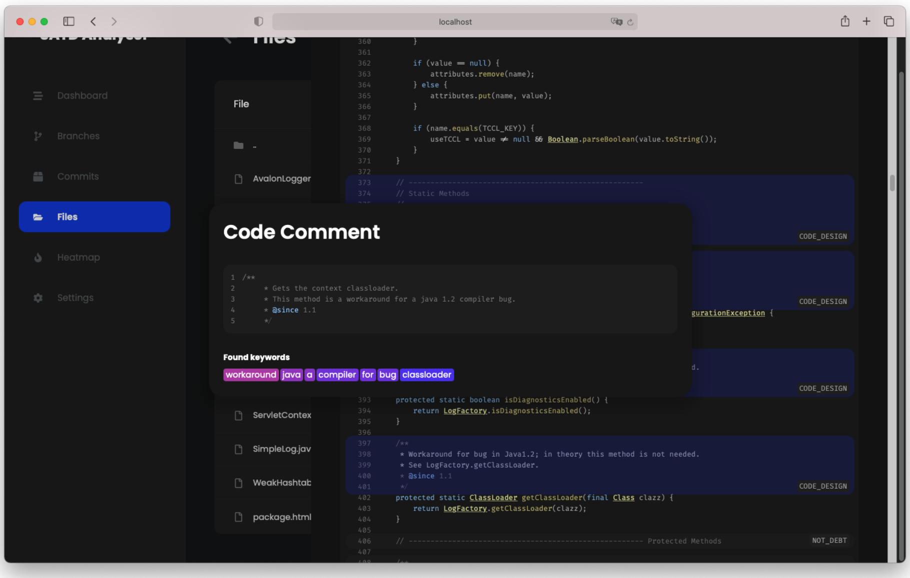


Figure 10: The heatmap of SATD



```
        }

        if (value == null) {
            attributes.remove(name);
        } else {
            attributes.put(name, value);
        }

        if (name.equals(TCCL_KEY)) {
            useTCCL = value != null && Boolean.parseBoolean(value.toString());
        }
    }

    // ------------------------------------------------------
    // Static Methods
```

CODE_DESIGN

CODE_DESIGN

Code Comment

```
/**
     * Gets the context classloader.
     * This method is a workaround for a java 1.2 compiler bug.
     * @since 1.1
     */
```

Found keywords

workaround java a compiler for bug classloader

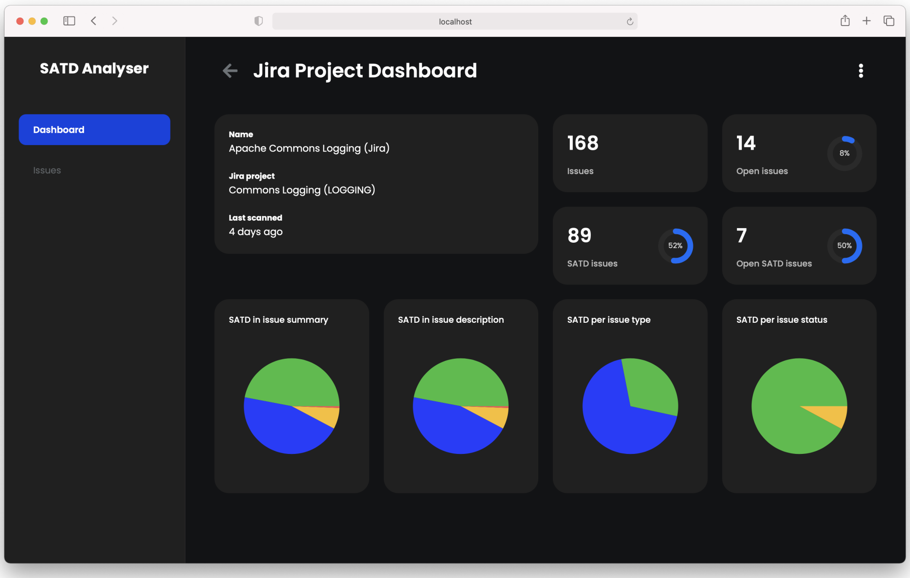
CODE_DESIGN

```
    protected static boolean isDiagnosticsEnabled() {
        return LogFactory.isDiagnosticsEnabled();
    }

    /**
     * Workaround for bug in Java1.2; in theory this method is not needed.
     * See LogFactory.getClassLoader.
     * @since 1.1
     */
    protected static ClassLoader getClassLoader(final Class clazz) {
        return LogFactory.getClassLoader(clazz);
    }

    // ------------------------------------------------------ Protected Methods
```

CODE_DESIGN

NOT_DEBT

Figure 11: A comment being identified as SATD



Name
Apache Commons Logging (Jira)
Jira project
Commons Logging (LOGGING)
Last scanned
4 days ago

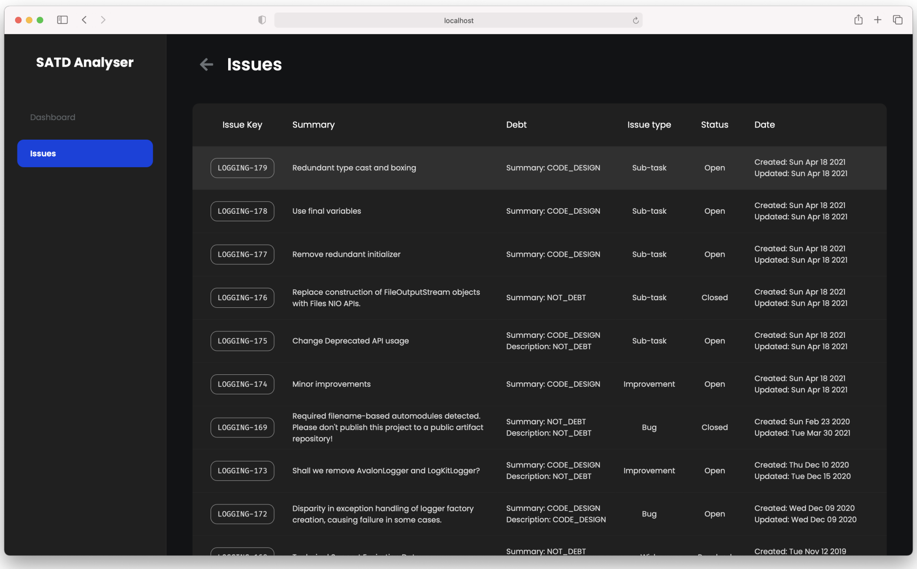
168 Issues

14 Open issues 8%

89 SATD issues 52%

7 Open SATD issues 50%

Figure 12: The dashboard for a JIRA project



Issues

Issue Key	Summary	Debt	Issue type	Status	Date
LOGGING-179	Redundant type cast and boxing	Summary: CODE_DESIGN	Sub-task	Open	Created: Sun Apr 18 2021 Updated: Sun Apr 18 2021
LOGGING-178	Use final variables	Summary: CODE_DESIGN	Sub-task	Open	Created: Sun Apr 18 2021 Updated: Sun Apr 18 2021
LOGGING-177	Remove redundant initializer	Summary: CODE_DESIGN	Sub-task	Open	Created: Sun Apr 18 2021 Updated: Sun Apr 18 2021
LOGGING-176	Replace construction of FileOutputStream objects with Files NIO APIs.	Summary: NOT_DEBT	Sub-task	Closed	Created: Sun Apr 18 2021 Updated: Sun Apr 18 2021
LOGGING-175	Change Deprecated API usage	Summary: CODE_DESIGN Description: NOT_DEBT	Sub-task	Open	Created: Sun Apr 18 2021 Updated: Sun Apr 18 2021
LOGGING-174	Minor improvements	Summary: CODE_DESIGN	Improvement	Open	Created: Sun Apr 18 2021 Updated: Sun Apr 18 2021
LOGGING-169	Required filename-based automodules detected. Please don't publish this project to a public artifact repository!	Summary: NOT_DEBT Description: NOT_DEBT	Bug	Closed	Created: Sun Feb 23 2020 Updated: Tue Mar 30 2021
LOGGING-173	Shall we remove AvalonLogger and LogKitLogger?	Summary: CODE_DESIGN Description: NOT_DEBT	Improvement	Open	Created: Thu Dec 10 2020 Updated: Tue Dec 15 2020
LOGGING-172	Disparity in exception handling of logger factory creation, causing failure in some cases.	Summary: CODE_DESIGN Description: CODE_DESIGN	Bug	Open	Created: Wed Dec 09 2020 Updated: Wed Dec 09 2020
[illegible]	[illegible]	Summary: NOT_DEBT	[illegible]	[illegible]	Created: Tue Nov 12 2019

Figure 13: The list of issues for a JIRA project