Technical Debt Prioritization: A Developer's Perspective

Abstract

Background: The prioritization of technical debt is an essential task in managing software projects because, with current analysis tools, it is possible to find thousands of technical debt items in the software that would take months or even years to be fully paid. **Aims:** In this study, we aim to understand which criteria software developers use to prioritize technical debt in real software projects. **Methods:** We performed a survey to collect data from open-source software projects in order to reach a large and diverse set of experiences. We analyzed the data using Straussian Grounded Theory techniques: open coding, axial coding, and selective coding. **Results:** We grouped the criteria into 15 categories and divided them into 2 super-categories related to paying off the technical debt and 3 related to not paying it. **Conclusions:** When participants decided to pay off technical debt, they wanted to do it soon. However, when they decided not to pay it, it is often because the debt occurred intentionally due to a project decision. Also, participants using similar criteria for their decisions tended to choose similar priority levels for those decisions. Finally, we observed that each software project needs to tailor the rules used to identify technical debt to their project context.

CCS Concepts: • **Software and its engineering** → **Software evolution**; **Maintaining software**.

Keywords: technical debt, technical debt prioritization, survey, grounded theory

ACM Reference Format:
. 2022. Technical Debt Prioritization: A Developer's Perspective. In *Proceedings of TechDebt '22: International Conference on Technical Debt 2022 (TechDebt '22)*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Technical debt is a metaphor introduced by Cunningham [2], who highlighted the benefits of taking on debt to speed up software development against the cost of the interest to pay that debt off later. Acquiring technical debt could bring short-term benefits [5], such as time and effort reduction in the development of tasks. However, it could result in extra costs in the long term, for instance, in more expensive new feature development and the maintenance of the legacy code.

Software development teams need to balance the benefits of technical debt against the cost to pay it off and its possible interest [12]. The uncertainties related to technical debt make the decision-making even more complex. Therefore, effective technical debt management could help teams by providing relevant information to aid decision-making.

The technical debt management process consists of identifying technical debt items, measuring their costs, and making decisions about which technical debt items must be paid off and in which order [6]. The identification consists of finding technical debt items and storing them in a list with data such as identifier, date and time, type of problem, and location. It is difficult, but sometimes possible, to estimate the principal, interest, and probability of interest for each technical debt item. With these items estimated, one could use some prioritization method to decide whether an item must be paid or not. It can also help decide which item should be paid first, which will incur less interest, and/or should be allowed to persist to achieve goals that increase the project's value.

In this study, we aim to understand which criteria software developers use in practice to prioritize technical debt items in real software projects. This fills a gap in the literature regarding studies of technical debt prioritization criteria in real software projects.

We used a survey to collect information from developers about open-source projects hosted on Github. The survey questions were based on technical debt items on projects to which the respondent had contributed. After showing each technical debt item, the survey asks the respondent to indicate how soon the item should be paid off and why.

We analyzed the answers using Straussian Grounded Theory (Straussian GT) techniques, namely open coding, axial coding, and selective coding, to identify the criteria developers used to prioritize technical debt. We grouped the criteria into 15 categories and 2 super-categories related to paying off the technical debt item: CODE_IMPROVEMENT and COST_BENEFIT; and 3 super-categories related to not paying off the item PROJECT_SPECIFIC_DECISION, PROBLEM_WITH_RULE, and UNUSED_CODE.

We found that when developers chose to pay off a technical debt item, they decided to do so soon. When they chose not to pay it was a project-specific decision. Also, when developers

TechDebt '22, May 22–23, 2022, Pittsburgh, United States
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

used the same criterion, the payment priority chosen was in the same neighborhood. Finally, we noted that each project needs a specific set of criteria to prioritize technical debt.

This work addresses the following research questions:

1. How do developers decide whether a technical debt item should or should not be paid off?
2. How do developers decide when a technical debt item should be paid off?

This paper is organized as follows: Section 2 discusses the previous related work. Section 3 describes the research methods used in this study. Section 4 shows the results and coding process. Section 5 discusses the findings. Section 5.2 discusses the threats to the validity of this study. Section 6 presents the conclusions and final remarks.

2 Related Work

Considerable research has been conducted in this area, including several studies related to the concepts and criteria used to prioritize technical debt.

Some work on technical debt prioritization offers a general view of the relevant criteria. Daneva et al. [3] used in-depth interviews and grounded theory techniques for data analysis to show that all prioritization criteria are related to the concept of business value. Martini and Bosch [8] found, through a combination of interviews, observations, and a survey, that competitive advantage, specific customer value, market attractiveness, and lead time are the most relevant prioritization criteria mentioned by the respondents. They also pointed out that the top three architectural technical debt effects relevant to prioritization are quality issues, doubling the effort, and contagious debt. The same authors [9] also studied how the cost of refactoring grows over time correlated with the propagation of architectural technical debt, and the impact of that growth. Their results imply that the rate at which interest affects development speed should be used as a prioritization criterion. Fernández-Sánchez et al. [4] found through a systematic mapping that expert opinion provides information that cannot be obtained from available tools, such as information about contracts, expected changes and the adoption of new technologies. They also found that time-to-market and team availability are criteria used to decide when a technical debt item should be paid off.

Other studies provide detailed criteria lists based on literature reviews. Riegel and Doerr [11] and Ribeiro et al. [10] each proposed a set of criteria based on their studies of the literature. These authors organized their criteria differently but used such categories as benefits, costs, risks, context, and effort. Becker et al. [1] organized results from a systematic literature review of papers about TD measures into a theoretical decision-making process.

However, researchers rarely seem to investigate how decisions are made in practice. Leppanen et al. [7] interviewed professionals and developed a framework for refactoring (one way to pay off technical debt). They found that static and dynamic code analysis can contribute to perceiving the need to pay off technical debt. Developers are the expert stakeholders who can determine whether a particular code structure would be better off after refactoring. However, the framework does not define which criteria led to the decision to pay the technical debt.

Our work seeks to understand the prioritization of technical debt in real software projects by directly collecting the opinions of developers. Our study extends Becker's [1] theoretical process by studying how prioritization decisions are made in practice. Our study also builds on Leppanen et al.'s framework [7] by identifying decision criteria.

3 Research Method

Our main goal is to understand which criteria software developers use to decide whether and when a technical debt item should be paid off in real software projects. For that purpose, We decided to use a survey to reach many projects and multiple types of technical debt rather than applying interviews that would bring more details but for a smaller number of projects.

The survey participants answered two questions for each technical debt item presented. The first is a multiple-choice question on the payment priority of the item. The second is an open text question to explain the reason for the choice made in the first question. These two questions provided data to understand how developers decide whether and when a technical debt should or not be paid off.

This survey is part of a larger effort to gather data to populate machine learning models to predict payment prioritization and to be able to evaluate these methods not only quantitatively, but also qualitatively.

We developed a *GitXplorer* tool to scrape public Github code repositories to find a large number of open-source projects and their developers. From a repository, *GitXplorer* finds developers and organizations, and from them, find new repositories and so on.

We analyzed Java software projects hosted on public Github repositories which we were able to analyze with SonarQube through the Sonarlizer tool, resulting in technical debt items related to each project. An example of such technical debt items is *"Remove this "close" call; closing the resource is handled automatically by the try-with-resources from the HttpProxy repository"*. With the technical debt items identified, we proceeded to the next step, which was asking developers to evaluate those items via our survey.

In order to administer our survey, we developed an InteraSurveyTD tool that shows respondents only those technical debt items related to the projects they have worked on. InteraSurveyTD shows a technical debt item and some information related to it, such as file, line location, and description. Then the respondent is asked "When should the item be paid off?" (multiple-choice) and "Why?" (open-text field).

We used Straussian GT [14, 15] techniques (open, axial, and selective coding) to analyze answers to the "Why" question. This analysis allowed us to understand which criteria the developers use to prioritize technical debt items.

3.1 Data Collection

The data collection process is composed of four steps: finding repositories and developers, matching developers with the files they have contributed to, identifying technical debt items, and applying the survey to respondents. Figure 1 shows the data collection process.

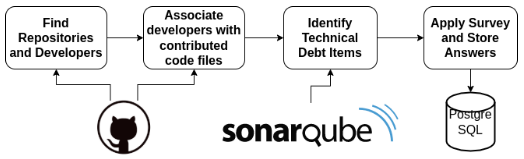

Find Repositories and Developers
Associate developers with contributed code files
Identify Technical Debt Items
Apply Survey and Store Answers
sonarqube
Postgre SQL

Figure 1. Data collection process.

We developed the GitXplorer tool to find public software repositories and developers at GitHub. After collecting git data, we developed the Sonarlizer tool to use SonarQube to identify technical debt items and code metrics and to associate developers with code files where they have been contributed. Finally, we developed the InteraSurveyTD tool to apply a survey in which developers answered questions about technical debt prioritization on items from projects they contributed to. The data were stored in a database to be analyzed by using Straussian GT [14, 15].

3.1.1 Find Repositories and Developers. We developed the GitXplorer tool to scrape public Github software project repositories and find real software projects and related developers. There is currently no public dataset available containing Github repositories and developers' contact information, so we used GitXplorer to develop this dataset. We used the GitHub REST API to access the data.

A repository is an entity that stores a project codebase, and it is related to many developers (users). It can also be owned by an organization (defined as a group of developers). Examples of repositories include Apache Maven, Google Kubernetes, and Microsoft Visual Studio Code. Developers can contribute to many repositories. They can also belong to many organizations. Examples of organizations include Apache Foundation, Microsoft, and Kubernetes Authors.

Each Github API call returns only one repository, organization, or developer. For this reason, we had to use the relationships to find all the data. For example, when we access a repository, we can access the list of developers who contribute to it and its owner organization, if there is one.

The data extraction started by adding the Apache Maven repository to the repository queue. The tool consumes the developer, organization, and repository queues iteratively to get the complete entity information and their related entities that have not yet been processed to feed the queues.

3.1.2 Technical Debt Item Identification. We developed the Sonarlizer tool to automatically perform the SonarQube analysis for different kinds of build environments, such as Apache Maven, Apache Ant, and Gradle. The SonarQube analysis was used to identify technical debt items.

As input, Sonarlizer receives a project name and a GitHub repository URL. Sonarlizer clones the repository from GitHub to our server. Then it performs a SonarQube analysis to find technical debt items and collect code metrics, such as number of lines, number of files, average complexity per file, and cognitive complexity. SonarQube has a list of rules, and when one of these rules is violated, a technical debt item is created. Some examples of rules are *Member Name, Unused Imports, Nested If Depth and Method Length.*

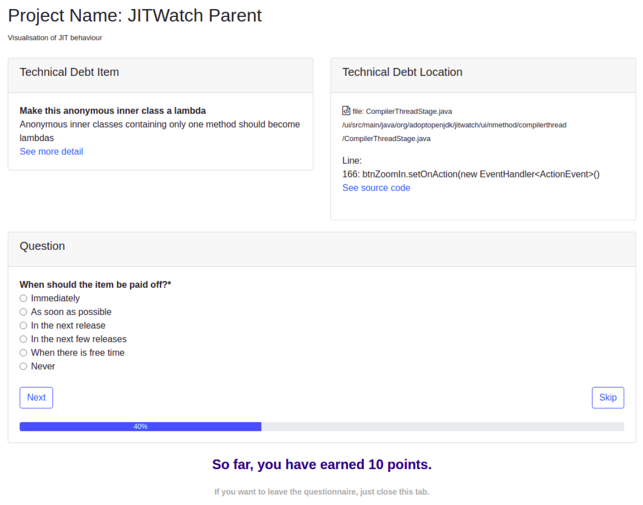
Finally, Sonarlizer relates the technical debt items to the developers who have contributed to the code in which the item is located by analyzing the git history to find which developers worked in each file. It then matches those developers to the technical debt items related to the file. This process allows the survey tool to show each respondent only the technical debt items related to files that the respondent has contributed to.

3.1.3 Survey. We developed the customer survey tool InteraSurveyTD to show developers technical debt items from projects they have contributed to and ask them whether and when the item should be paid off. We sent invitation emails [1] to the developers with a project brief and a link to the questionnaire with an identifier to load the technical debt items specific to that developer. Once the developer(respondent) follows the link, they are shown a set of instructions for completing the survey and informed consent for participation [2]. Before starting the survey, they had to confirm having read and agreed with consent terms and to be at least 18 years old. Then, the InteraSurveyTD tool chooses a technical debt item from the respondent's project.

To preserve anonymity, not all the items shown to a respondent are from files they have contributed to. InteraSurveyTD randomly chooses an item specific to the developer respondent only 70% of the time. This avoids the situation where a particular file has just one contributor, so the developer's responses could be identified. Also, we do not store information about any relationship between the answer and the participant. All selected projects have three or more participants, which allows for anonymity and prevents tracking of participant answers.

[1]http://raptor210.startdedicated.com/pack/email-template-anonymized.pdf - Later, we will store in a scientific repository in order not to compromise doubly anonymous reviewing
[2]http://raptor210.startdedicated.com/pack/research-web-consent-anonymized.pdf

Figure 2 shows a survey screen presenting a technical debt item and the questions, which capture the developer's view of when the technical debt item needs to be paid off.


Project Name: JITWatch Parent

Visualisation of JIT behaviour

Technical Debt Item

Make this anonymous inner class a lambda
Anonymous inner classes containing only one method should become lambdas
See more detail

Technical Debt Location

file: CompilerThreadStage.java
/ui/src/main/java/org/adoptopenjdk/jitwatch/ui/nmethod/compilerthread
/CompilerThreadStage.java

Line:
166: btnZoomIn.setOnAction(new EventHandler<ActionEvent>()
See source code

Question

When should the item be paid off?*
- Immediately
- As soon as possible
- In the next release
- In the next few releases
- When there is free time
- Never

So far, you have earned 10 points.

If you want to leave the questionnaire, just close this tab.

Figure 2. Questionnaire question example.

Question 1 has six possible answers on a descending scale according to how urgent the item is (these explanations are provided in the instructions to respondents):

- **Immediately**: pay the item off before developing anything else;
- **As soon as possible**: pay item off in the current release;
- **In the next release**: plan item payment for next release;
- **In the next few releases**: it doesn't postpone payment indefinitely, but it doesn't have to happen in the next release;
- **When there is free time**: no planning is required, but eventually the item should be paid;
- **Never**: the item is not important for the project or it is not in fact a technical debt item, or for some other reason, should not be paid.

The answer to question 2 is an open text field where the respondents can explain their answer to question 1. After storing the answers to both questions, InteraSurveyTD repeats the process with another TD item.

Respondents can leave the survey at any time, and come back using the email invitation link. To motivate respondents, there was also a gamification scheme whereby developers could earn points by answering more questions and by referring other developers to the study. Points earned allowed respondents to receive prizes at the end of the study.

3.1.4 Pilot Study. We conducted a pilot study to evaluate and improve the data collection flow. The pilot study included 15 students who developed open-source software in Java in an Extreme Programming course from the University of São Paulo. We sent the invites through email and Facebook Messenger with the link to the survey.

After one week, we sent an evaluation survey to understand how easy it was to respond to the original survey, if the website and invite email provided enough and clear information about the research and technical debt, and concerning the gamification. For each of these areas, we asked for improvement suggestions. The evaluation survey was answered by 5 students.

We used the collected suggestions to improve parts of the text, to add more information and improve the interface, and to change the survey flow - for example by adding a button to skip a question.

3.2 Data Analysis

We analyzed the survey data qualitatively applying Straussian GT [14, 15]. We decided on Straussian GT instead of classic Grounded Theory because our research questions were defined upfront and derived from the literature [13]. They also are broad and open-ended.

The data analysis process started as soon as data was available. We analyzed applied open, axial, and selective coding. Every time we added a new code during analysis of an answer, we would write a memo describing it. Otherwise, we would try to improve the existing memo. The application of these techniques was iterative for each new response.

From the beginning of data collection, we constantly compared data, memos, codes, subcategories, categories, and super-categories to ensure that the data were correctly interpreted and were in the categories that best fit them. The period between sending the first invitation by e-mail and closing the survey was approximately six weeks.

We applied open coding to the survey responses. This process involves segmenting the answers into excerpts with a singular meaning and expressing that meaning with a code. For each answer to question 1, we applied a unique code based on the multiple-choice option. For example, for the answer "As soon as possible" we used the code ASAP. For answers to question 2, we applied between one and three codes for each answer. Like in "This code is correct. Lint rules cannot be applied blindly.", where we applied two codes: RULE_SHOULD_NOT_BE_APPLIED and LINT_RULES. This was an iterative process where we added, changed, and removed codes with each answer analyzed.

During axial coding, the open codes were reassembled in new ways to form categories. The goal was to create a higher abstraction level. Thus, codes were grouped to form subcategories, and in turn, they were organized into categories. In addition, we also tried to find relationships between the categories to form super-categories. This process was highly iterative, with codes and categories forming and re-forming as more data was incorporated into the evolving understanding. We wrote a memo to explain each category and provide examples of the answers that motivated its creation.

We applied selective coding to refine and integrate categories, revealing the main categories indicating the developers' criteria to prioritize technical debt and their relationships. We were able to identify the technical debt prioritization criteria used by the developers from the categories and their relationships.

We stopped collecting data when we identified that the answers did not result in new codes.

4 Results

In this section, we describe our results and findings. Although applying the Gaussian GT techniques has been made iteratively, and all the steps were taken simultaneously, in order to improve reading, we divided them into open, axial, and selective coding.

We sent invite emails to 2471 developers distributed in 855 projects. 1869 developers opened the invitation email, and 341 accessed the survey; however, only 39 developers from 21 projects [3] answered it.

The developers chose 11 times to pay off the technical debt item immediately, 30 times as soon as possible, 7 in the next release, and 66 never; thus 42% chose to pay off the technical debt, and 58% chose not to pay it off.

4.1 Open Coding

We organized the collected data in a spreadsheet [4]. Each row contained one participant's answer about one technical debt item. Besides the developers' answers, each row has columns to describe the technical debt item based on SonarQube data. All open codes were created in vivo, meaning they were derived directly from the collected data.

4.2 Axial Coding

We organized and assembled the open codes in order to form two levels of categories: the first level contains categories that group open codes, and the second level contains super-categories that group the first-level categories. For each open code, we tried to add it to an existing category that encompassed its meaning. If that was not possible, we created a new category. We performed this step iteratively, constantly revisiting and evaluating the set of categories. We performed the same iterative process to group the categories into super-categories. Below we list and briefly discuss the categories and super-categories.

Super-Category PROJECT_SPECIFIC_DECISION includes the following categories:

DESIGN_DECISION: More experienced developers often use design patterns to create the software architecture and write the code. However, some of these design patterns break quality rules that generally apply to only a code snippet and not the architecture of the software as a whole. Therefore, for some software architectures, it makes sense to break some quality rules so that a design standard is maintained. Thus it is easier to read, maintain, scale and improve the performance of the software.

Examples: When developers needed to choose between paying a technical debt item or keeping their design intact, they always chose to preserve the design and *Never* pay off the debt. One developer refers to another Lint rule: "This code is correct. Lint rules cannot be applied blindly", and for another item, he referred to Javadocs to explain his decision: "This file is correct per the Javadoc documentation".

MEANINGFUL_NAMES: Some developers prefer to use their naming conventions for software projects or modules. For example, Java convention defines variable names using the following regular expression: '^[a-z][a-zA-Z0-9]*$', that is, the first character must be a lowercase letter followed by zero or more alphanumeric characters. Using other conventions triggers a TD item.

Examples: One developer explained that the names came from reflection: "I suspect this code has to interface with generated code that gets those names via reflection." Another developer preferred using their naming convention: "I prefer the way I name variables/parameters to the Java convention". Another one uses specific names for that project: "The name is specific to the project". Another developer explained: "There are reasons for the names. I likely will never fix them."

KEEP_READABILITY: Sometimes giving up standards to make the code more readable and easier to understand is the best choice. Some problem solutions are complex, and trying to reduce or fit them into code patterns can make the solution hard to read and understand, so it is best to leave the pattern aside so that the code is easier to maintain.

Examples: Two developers chose *Never* to pay off the technical debt items to keep the code easy to read: "GitException is used to carry failure information and declaring it makes it clear that it can be thrown.", and "No. Code is more readable the way it is."

BREAK_SOMETHING_ELSE: Sometimes paying off a technical debt item could break something else; that is, changing a snippet could break functionality or compatibility. For this reason, sometimes paying off a technical debt item is not worth it because its principal could be very high, costing several days of development.

Examples: One respondent chose to pay off an item *As soon as possible* because changing the code can break the existing code: "This TODO should be treated with more care, because it can break existing code." For two technical debt items, another developer chose *Never* to pay off the

[3]http://raptor210.startdedicated.com/pack/projects.csv
[4]Our data is available at http://raptor210.startdedicated.com/pack/answers.csv - Later, we will store in a scientific repository in order not to compromise doubly anonymous reviewing

technical debt items because: “Removing this code would break functionality and compatibility”. And for the other: “Deprecation in Jenkins does not mean "remove the code". If we remove the code, it will break compatibility. One of the compelling values of Jenkins is that it retains compatibility so that plugins compiled many years ago continue to operate with current releases.”

Super-Category PROBLEM_WITH_RULE includes the following categories:

RULE_SHOULD_NOT_BE_APPLIED: Some projects use their standards, so not all quality rules should apply.

Examples: Every time a respondent explained that a rule should not be applied, they decided *Never* pay off the technical debt item. They explained that rule application did not precisely identify a technical debt item, e.g. “This is a stupid rule. Sometimes verifying that the given code executes without throwing is all you need.” They also explained that some rules are wrong based on program language documentation, as in the previous example about Javadoc documentation.

ARBITRARY_RULE: For some developers, some rules should not be applied because they believe that the kind of technical debt found by the rule is not technical debt. Thus, the rule should be ignored - for instance, rules that try to anticipate possible runtime errors.

Examples: All developers that indicated an arbitrary rule chose *Never* to pay off the technical debt item, such as in: “This evaluation seems very wrong — how can a static method call throw an NPE?”, and “The rule is wrong, it actually throws NoSuchElementException, but the linter is unable to detect it. It probably has an incomplete type system and flow analysis.”

FALSE_POSITIVE: In some contexts, a technical debt item is considered a false positive. Unlike the rules that should not be applied to a project or the rules that wrongly identify technical debt items, this code refers to cases in which, in another code snippet, the item might be considered technical debt. However, it does not make sense as technical debt for that snippet of code.

Examples: Every time a respondent indicated a false positive, they chose *Never* to pay the technical debt. Many used the term "false positive", but others said more, e.g. “Once again, default case may or may not make sense: in cases where it is omitted it is literally useless.”

Super-Category UNUSED_CODE: includes the following categories:

UNUSED_CODE: Code to test a concept or idea, but later, is replaced by some better code or, after writing it, the developer realizes it is not a good concept/idea. Some code is considered useless because it was written as an example to show or teach.

Examples: All developers chose *Never* to pay off technical debt where the code was unused. In some cases, the item was inside a test class, or private, or just never used.

Super-Category CODE_IMPROVEMENT includes the following categories:

PERFORMANCE: Code written by a developer is not always the one with the best performance. To improve performance the developer could spend hours or even days to rewrite the code to reduce complexity. They can also simply use some programming language features, such as built-in or call methods that perform processing in parallel, resulting in significant improvement in the performance of the software.

Example: One respondent chose to pay off a technical debt item *As soon as possible* because they saw it specifically to be a performance issue.

REMOVE_BUG: When developers write code, they cannot cannot always see all possible situations. So sometimes there is a need to rewrite the code in order to remove bugs that are generated by the wrong use of logic or because the code is not covering every possible case.

Example: One respondent decided to pay off the technical debt item *As soon as possible* because: “A better exception here would be a good idea. This might even be a bug.”.

TEST_FAIL: Sometimes code can break the tests. This happens because the code was written incorrectly and therefore returns unexpected behavior, in which case the code must be rewritten to behave as the test expects. It may also have been written with a different structure than expected by the tests and therefore the test also fails. In this case the code structure can be changed to adapt to the test or the test can be changed to adapt to the code structure.

Example: One respondent chose to pay off the technical debt *Immediately* because it broke a test: “It should be paid immediately because the test is broken. The expected value is "Number of created files" and the files.size() is an integer.”

IMPROVE_READABILITY: A code needs to be rewritten to be easier to read. Generally, rewriting can be done by renaming variables, classes, methods/functions. It can also be done by decreasing complexity, such as removing "if", "for" and "while" statements. It can also be done through refactoring the code or using methods that encapsulate part of it. These techniques make the code easier for other developers to understand.

Examples: A respondent decided to correct one of the identical sub-expressions on both sides of operator "||" C because he thought it was “Confusing”. Another developer decided to use isEmpty() As soon as possible to check whether the collection is empty or not to make it “clearer”.

INCOMPLETE_CODE: The code is incomplete when some implementation is missing for the method/function to work as expected. Most of the time, developers annotate these missing parts with the following comment: "TODO: explanation", where TODO means the code needs to be written, and sometimes there is an explanation about what needs to be written and/or how.

Examples: Sometimes this code leads to *Immediate* payment and sometimes the payment should be done *As soon as* *possible*. A developer chose to pay two technical debt items *Immediately* because he considered that TODO annotation should be paid off urgently: "From my point of view, TODOs should be urgent, otherwise people are going to leave it there and procrastinate as much as they can."

Super-Category COST_BENEFIT includes the following categories:

LOW_PRINCIPAL: The code is easy to modify, that is, in a few minutes a developer is able to fix the problem or complete the logic. Therefore, the technical debt item has a low principal cost.

Examples: Respondents tended to pay off these items either *Immediately* or *As soon as possible*. They used terms such as "trivial", "easy and safe", and "easy change". In one case, the respondent thought that the change could even be automated.

HIGH_PRINCIPAL: The code is hard to modify, that is, a developer could take several hours or even days to (re)write the code to fix the problem or complete the logic. Therefore, the technical debt item has a high principal cost.

Examples: In one case, a developer thought that the technical debt item should be paid off *As soon as possible*, but "This TODO should be treated with more care, because it can break the existing code". Another developer chose *Never* to pay the debt because the effect was minimal and the complexity to remove the item was huge: "This TL works as part of injected code during test runs. The overhead is minimal and the complexity of calling remove is huge."

LOW_INTEREST: The code is almost never called by other methods/functions, that is, it has a low probability of causing extra effort if the item is not paid off. Therefore, the technical debt has low interest.

Examples: One respondent chose to pay off an item *In the next release* because he did not face that issue at that moment: "Since it might define other method interfaces, I would prioritize this issue to the next release. In this code, we do not face this issue at the moment because the method is private and the public methods do not re-throw the exception." The same developer also chose to pay off *In the next release* another technical debt item because it had a minor impact: "Good catch. Minor impact/nit." Another developer chose *Never* pay the technical debt item because it had low impact and high effort to pay off: "This TL works as part of injected code during test runs. The overhead is minimal and the complexity of calling remove is huge.". Another developer also chose not to pay off an item because it is inside a private method.

4.3 Selective Coding

We performed selective coding to refine and integrate categories. The main goal was to understand the main criteria that developers use to decide whether a technical debt item should be paid off or not and when to make the payment. Abstraction was used to incorporate all aspects related to the collected data and coding [15].

The codes and categories fell naturally into those representing influences on the decision to pay off technical debt and those influencing the decision *not* to pay off technical debt. The super-categories PROJECT_SPECIFIC_DECISION, PROBLEM_WITH_RULE and UNUSED_CODE represent the codes describing decisions not to pay off the technical debt item. Figure 3 shows the categories and super-categories related to decisions *not* to pay off the technical debt item.

In some situations, a PROJECT_SPECIFIC_DECISION is made to keep the current solution (i.e. not pay off the debt) instead of rewriting the code to follow the rules. These are are project-specific decisions because a similar situation in a different project might lead to a different decision about paying off the debt. Sometimes these decisions are related to architectural design (DESIGN_DECISION); for example, when a generic interface is designed to propagate any checked exception or singletons are used as constants. Another type of project-specific decision is related to naming conventions (MEANINGFUL_NAMES), i.e. when a project prefers to use a different way to name variables, methods, functions, and classes than conventions commonly used. Another type of project-specific decision is related to a concern about readability (KEEP_READABILITY); the decision is not to change the code and follow the rules because that would make the code harder to read. Finally, developers also decided not to pay the technical debt item when it would break compatibility or functionality (BREAK_SOMETHING_ELSE); fixing a code snippet would imply changing a lot of other code snippets that depend on the first one. The only exception is when it breaks something else, but it is important to pay off a debt (TODO_SHOULD_BE_TREATED_CAREFULLY) as soon as possible; that is when a missing snippet needs to be written to provide the expected behavior.

The PROBLEM_WITH_RULE category describes cases where a respondent cites a problem in a rule used to detect technical debt items in SonarQube. Respondents felt some rules should not be applied (RULE_SHOULD_NOT_BE_APPLIED) because they identified irrelevant technical debt. Others felt that some rules were arbitrary (ARBITRARY_RULE), not logical, or incorrectly evaluated a technical debt item. Some cases were cited as false positives (FALSE_POSITIVE); that is, the respondent considered the found item not to be a technical debt item, at least in the specific situation.

Developers sometimes wished not to remove or change code that exists just as an example, even though the code has been superseded by other code or is no longer used. Sometimes, developers want to keep the UNUSED_CODE for future comparison. Some projects are created as proof-of-concept or just to teach software development.

The super-categories CODE_IMPROVEMENT and COST_BENEFIT summarize the influences for developers deciding to pay off the technical debt item, as shown in Figure 4.

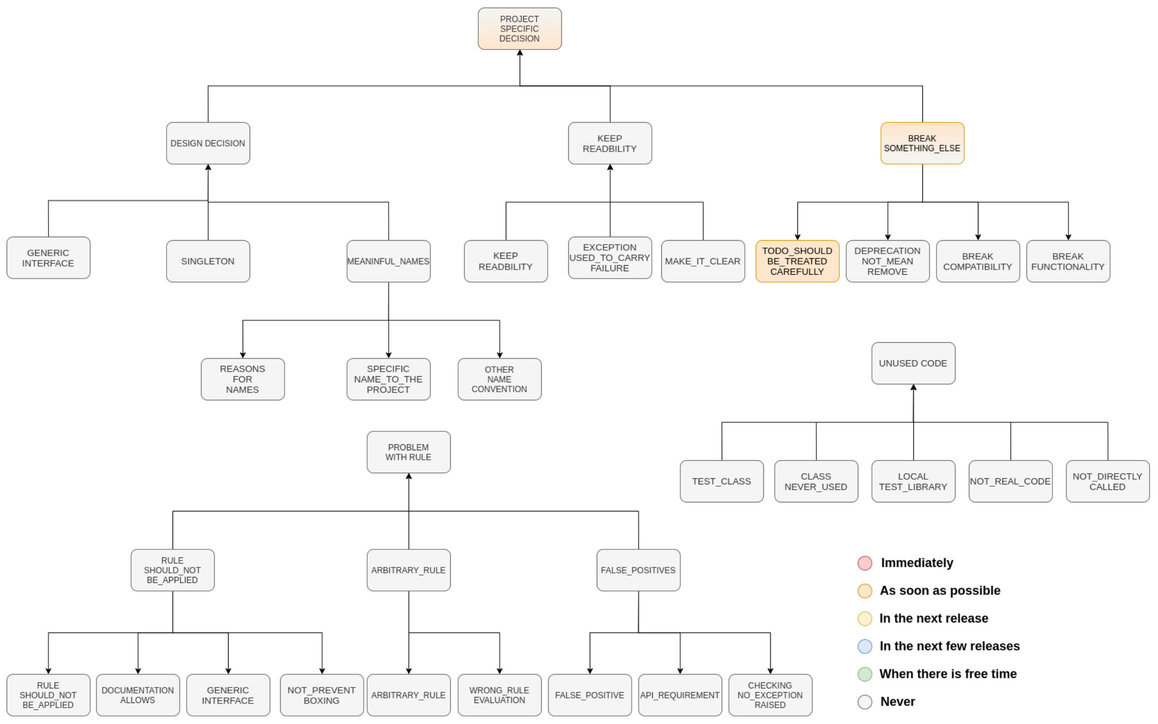


Figure 3. Codes and categories for not paying off technical debt items.

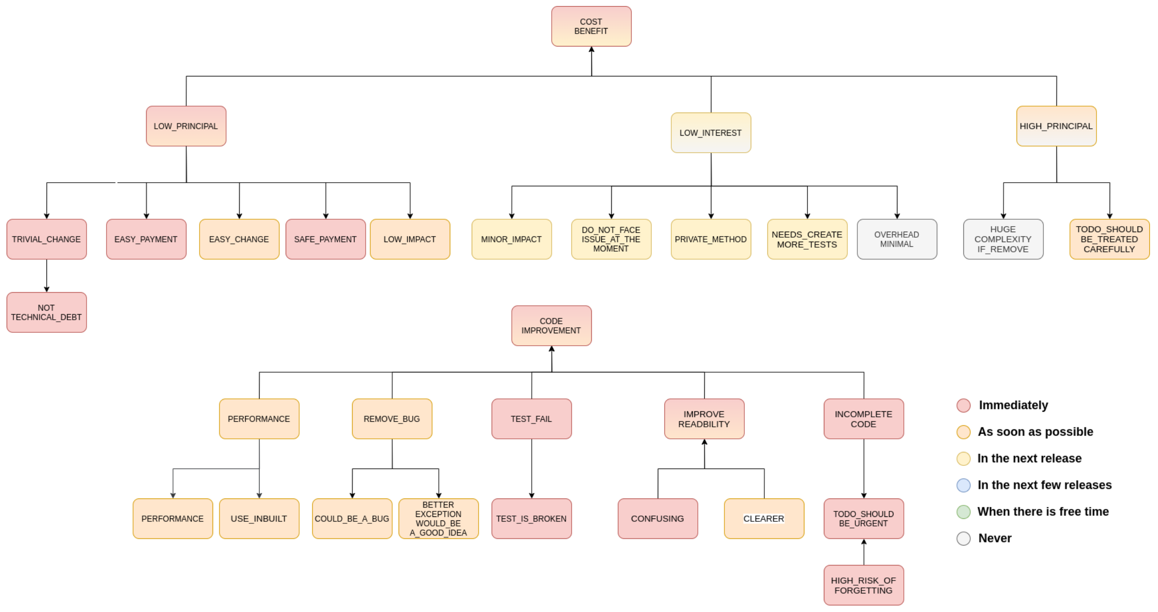


Figure 4. Codes and categories for paying off technical debt items.

When a developer's reason for paying off a technical debt item fell into one of the CODE_IMPROVEMENT categories, there are interesting relationships between the specific reason and how quickly they wanted to pay off the debt (i.e. "immediately" or "as soon as possible"). When the reason to pay off the debt is to improve the performance (PERFORMANCE) or remove bugs (REMOVE_BUGS), developers indicated that it should be paid off as soon as possible. On the other hand, they decided to pay off the technical debt immediately when the reason was a failed test (TEST_FAIL) or incomplete code (INCOMPLETE_CODE). Finally, when the motivation is to improve readability (IMPROVE_READABILITY), they want to pay off the debt immediately when the code is confusing, and as soon as possible when they want to clarify the code.

The other super-category describing cases where the technical debt item should be paid off is COST_BENEFIT. There are also variations in how quickly the debt should be paid off in these cases. When the criteria used were low principal (LOW_PRINCIPAL), respondents immediately chose to pay off the technical debt item because they considered it a trivial change, an easy and safe payment. However, when the debt item was considered an easy change or low impact, they decided to pay it off in the next release. When the criteria used was low interest (LOW_INTEREST), respondents chose to pay off the technical debt item in the next release, except when overhead was minimal (OVERHEAD_MINIMAL), and the complexity to remove it was huge. About (HIGH_PRINCIPAL), they decided not to pay it when there was a huge complexity to do so, and to pay off as soon as possible when there is a TODO tag that should be treated carefully.

5 Discussion of the Results

The developers used a wide array of criteria both to decide whether a technical debt should be paid off and when. They had different motivations when deciding to pay off a technical debt or not in different kinds of situations. These criteria were grouped into three super-categories that define when a technical debt item should not be paid off and two that define when an item should be paid off.

Another interesting observation is that when developers decide to pay a technical debt item off they want to do it soon: immediately, as soon as possible, or in the next release. However, it was not possible to determine through data if this decision was technically based or if personal feelings (worry, anxiety, fear of the reputation) made them choose higher priority actions.

In addition, they used specific criteria for each priority level. This means that when they use the same criterion in different instances, they also choose the same priority level or a neighboring level. For instance, when the technical debt item was about improving readability, participants always decided to pay off those debts immediately or as soon as possible, i.e. they chose neighboring categories. The codes OVERHEAD_MINIMAL and HUGE_COMPLEXITY_IF_REMOVE are the exceptions to that observation, as there was more variety in developers' answers related to these criteria.

Another finding is that each software project needs a specific set of rules to identify technical debt items that are relevant to the project. More than half of the answers were never to pay off the technical debt item. For many of these cases, the respondents explained they were using a different pattern than the one identified as technical debt by the SonarQube detection rules.

We identified categories similar to the decision-making criteria presented by Riegel and Doerr [11] and Ribeiro et al. [10]. Like Leppanen et al. [7], we present a framework to decide on the payment of a technical debt item, but our model is based on the decision criteria of the respondents. The first and fourth findings presented in this discussion confirm the literature. However, the second and third findings are new completely new.

5.1 Implications for Researchers and Practitioners

The results of this study could be used as the basis for researchers to identify other criteria that developers use to decide to pay off or *not* a technical debt item and to choose the payment priority level. This study could be replicated in other software project groups to identify new decisions and prioritization criteria, such as applying it to projects that use other programming languages and non-OSS projects. Besides that, other methods such as interviews could be used to identify and better understand the criteria.

After defining criteria, researchers could study the scenarios in which they are used, that is, to define when and how each criterion is used based on project variables like a programming language, project patterns, project size, development methodology, development team size, and others. With these definitions, it is possible to create guidelines to assist developers to decide the payment priority level for each technical debt item they identify in their software. In addition, researchers can try to relate the criteria to software context, such as code metrics and commit history, to automatically categorize payment of technical debt using, for instance, machine learning.

Practitioners can use the criteria we have found to evaluate and plan technical debt payment pragmatically in their projects. That is, they could verify that the criterion applies to the project, and, if so, they use it to define its priority level of payment. In addition, practitioners who develop technical debt management tools could use the criteria list identified here to improve the technical debt identification tools. After further research, they could use the criteria to tune technical debt management software for each project established on context based in the guidelines.

5.2 Threats to Validity

In this study, we adopted a careful approach to mitigate possible biases and misinterpretations. Below, we describe the steps we took to reduce threats to validity.

Construct validity: We conducted a pilot study by applying the questionnaire to fifteen developers. First, they answered the questions related to their projects, and then they wrote a report about their experience using the questionnaire tool and suggested improvements. Based on the reports, we fixed some issues in the tool and applied improvements to make sure the data items reflected a consistent interpretation of the study constructs.

External validity: This study can be replicated in other software projects. For that, it will be necessary to analyze the project with SonarQube and use the questionnaire tool to identify technical debt items and collect answers. We have collected answers from 21 Java open-source projects with a variety of sizes and features. Future replication should be conducted on a larger number of projects, other programming languages, and non-OSS projects. Applying an interview can also help understand the prioritization criteria and make them more general.

Internal validity: We analyzed the answers one by one and applied codes to them. After that, we tried to improve the codes and interpret them through grouping. Then, we reviewed all of them several times to make sure they accurately represented the answers. We extracted the conclusions and verified that they all were derived from the data. We followed the standard guidelines for qualitative coding reviewed by other authors. The iterative approach helps to mitigate analysis biases. In addition, the survey was designed specifically to capture the relationship between criteria and decisions to pay off or not technical debt items. Thus the conclusions about that relationship come directly from the data.

Reliability: We followed Straussian GT analysis techniques to interpret the data. Initially, one author conducted the coding and analysis. Then, the other authors revised, discussed, and iteratively improved the codes and analysis. All steps were carefully documented.

6 Final Considerations and Future Work

In this study, we performed a survey to collect data on Java open-source software projects hosted on Github to understand which criteria software developers use in practice to prioritize technical debt items, that is, to decide whether and when a technical debt item should be paid off. The participants were asked questions about technical debt items from the projects they had contributed to.

We analyzed the data using Straussian GT techniques to identify developers' criteria to prioritize technical debt. We grouped the criteria into 15 categories. Then, we grouped them into 2 super-categories related to paying off the technical debt item; and 3 super-categories related to not paying off the item.

We observed that some participants decided not to pay off a technical debt that had occurred because of a project decision. However, when participants decide to pay off an item, they want to do it soon. Another observation is that all respondents who use a particular criterion, choose the same priority level or a neighboring level. Finally, we noted that each software project needs a specific set of rules to identify technical debt.

In future work, it will be important to expand the number of projects and participants to cover more kinds of projects and types of technical debt. Furthermore, we intend to train machine learning methods to prioritize technical debt items automatically.

Acknowledgments

This study was financed in part by the Coordenacao de Aperfeicoamento de Pessoal de Nivel Superior – Brasil (CAPES) – Finance Code 001.

References

[1] Christoph Becker, Ruzanna Chitchyan, Stefanie Betz, and Curtis McCord. 2018. Trade-off decisions across time in technical debt management: a systematic literature review. In *Proceedings of the 2018 International Conference on Technical Debt.* ACM, pp. 85–94.

[2] Ward Cunningham. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (1993), pp. 29–30.

[3] Maya Daneva, Egbert Van Der Veen, Chintan Amrit, Smita Ghaisas, Klaas Sikkel, Ramesh Kumar, Nirav Ajmeri, Uday Ramteerthkar, and Roel Wieringa. 2013. Agile requirements prioritization in large-scale outsourced system projects: An empirical study. *Journal of systems and software* 86, 5 (2013), pp. 1333–1353.

[4] Carlos Fernández-Sánchez, Juan Garbajosa, and Agustín Yagüe. 2015. A framework to aid in decision making for technical debt management. In *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on.* IEEE, pp. 69–76.

[5] Yuepu Guo, Rodrigo Oliveira Spínola, and Carolyn Seaman. 2016. Exploring the costs of technical debt management–a case study. *Empirical Software Engineering* 21, 1 (2016), 159–182.

[6] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *Ieee software* 29, 6 (2012).

[7] Marko Leppänen, Samuel Lahtinen, Kati Kuusinen, Simo Mäkinen, Tomi Männistö, Juha Itkonen, Jesse Yli-Huumo, and Timo Lehtonen. 2015. Decision-making framework for refactoring. In *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on.* IEEE, 61–68.

[8] Antonio Martini and Jan Bosch. 2015. Towards prioritizing Architecture Technical Debt: information needs of architects and product owners. In *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on.* IEEE, pp. 422–429.

[9] Antonio Martini and Jan Bosch. 2016. An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on.* IEEE, pp. 31–40.

[10] Leilane Ferreira Ribeiro, Mário André de Freitas Farias, Manoel G Mendonça, and Rodrigo Oliveira Spínola. 2016. Decision Criteria for the Payment of Technical Debt in Software Projects: A Systematic Mapping Study.. In *ICEIS (1).* 572–579.

[11] Norman Riegel and Joerg Doerr. 2015. A systematic literature review of requirements prioritization criteria. In *International Working Conference on Requirements Engineering: Foundation for Software Quality.* Springer, pp. 300–317.

[12] Carolym Seaman and Yuepo Guo. 2011. Measuring and Monitoring Technical Debt. *Advances in Computers* 82, 6810 (2011), 25–46.

[13] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering.* 120–131.

[14] Anselm Strauss and Juliet Corbin. 1994. Grounded theory methodology: An overview. (1994).

[15] Anselm Strauss and Juliet Corbin. 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Sage 2nd Ed.