Using Gamification to Address Technical Debt: A Case Study at Adyen

Version of February 15, 2023

Mark Meijhuis

Using Gamification to Address Technical Debt: A Case Study at Adyen

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Mark Meijhuis
born in Amsterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Adyen
Simon Carmiggeltstraat 6 - 50
Amsterdam, the Netherlands
www.adyen.com

Using Gamification to Address Technical Debt: A Case Study at Adyen

Author: Mark Meijhuis
Student id: 4389255

Abstract

Technical debt is a term that describes the consequences of taking shortcuts or quick-and-dirty solutions in the software engineering process, in order to gain short term advantages in the development process of software projects. In this paper, we investigate the technical debt present at the fintech company Adyen, and the developers' experience with technical debt. We investigate their behavior towards testing and refactoring, and give insight into module health with the Delta Maintainability Model. With the conclusions from this research, we propose a novel gamification system called 'Code Karma' to address these technical debt issues, give insight into efforts that increase code quality and motivate developers to improve internal software quality. We found that the Delta Maintainability Model may be used to evaluate module health over time. Additionally, developers believe that 'Code Karma' has a positive influence on the internal quality of the system.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen. Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. M.F. Aniche. Faculty EEMCS, TU Delft
Company supervisor:	F. Braz, MSc. Adyen
Committee Member:	Dr. Dr. J.G.H. Cockx. Faculty EEMCS, TU Delft

Preface

Firstly, I would like to thank Arie van Deursen for supervising me, discussing results and coming up with new angles to approach the multi-facetted problem of technical debt. Secondly, I would like to thank Maurício Aniche for being available for feedback on ideas and advice on statistical methods. Lastly, I would like to thank the developers at Adyen for deep insights in the company and the development process, brainstorming new ideas and thinking of angles to approach certain problems and concepts.

Mark Meijhuis
Delft, the Netherlands
February 15, 2023

Contents

List of Figures

Chapter 1

Introduction

Technical debt is a term that describes the additional development work of taking shortcuts or quick-and-dirty solutions in the software engineering process, in order to gain short term advantages in the development process of software projects. These short term advantages, such as reduced cost or time to market, may be beneficial for a short while, but often lead to technical insufficiencies and limitations that may be hard to overcome in the future. Additionally, if developers have to work with or around technical debt, the technical debt might be increased in the process. This is why technical debt is often compared to financial systems and economic theories [5]. The principal of the technical debt is the money and resources that are used to overcome the technical insufficiencies, and the interest is the excess money and resources that are needed to work with the technical debt in the future [24].

Even though code-related technical debt has gained the most attention, other types of technical debt can be present as well [24]. Other types of technical debt can be a lack of testing, which makes production code unstable and the development process harder and more time-consuming due to hidden bugs or lack of tests that catch new bugs while debugging. An inefficient build process can also inhibit the efficiency of the development process. High-level decisions, such as architectural decisions, can be the most important source of technical debt, since it affects the largest proportion of the codebase and the development process [33]. There seems to be a dynamic between software engineers, that want to protect software quality, and business developers, that care about time to market [48].

Unintentional technical debt can also be incurred in multiple ways. Developers may lack competence to create an optimal solution, and design a sub-par system that needs to be worked around. New technologies may exist that better fit the software project. The business goals of the project may change, making the old code not the perfect fit for the product and something that needs to be rewritten or worked around. The company might not have clear coding guidelines, making it harder to work on the codebase due to a lack of consistency. The complex nature of technical debt is summarized in a simple way in Fowler's technical debt quadrant [13], which can be seen in Figure 1.1.

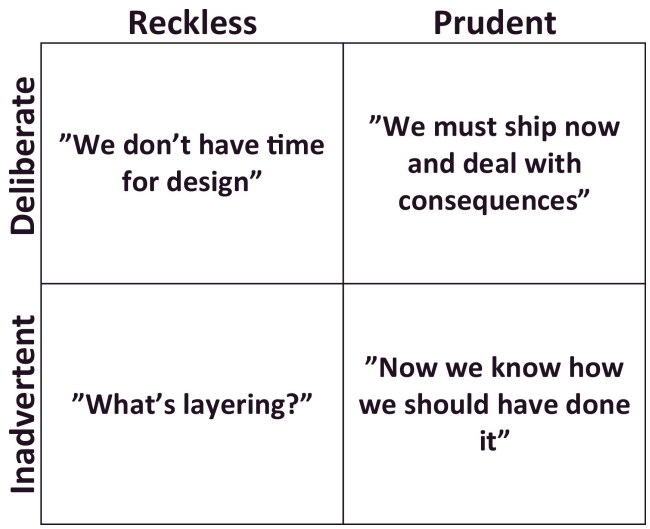

Figure 1.1: Technical debt quadrant.

Technical debt might not always be problematic, since a small amount of technical debt can be beneficial if the business impact is large. Thus, technical debt can be managed if the management pays attention to managing it. Companies can use backlogs for managing the technical debt issues, use static analysis tools to identify debt and reserve time for developers to work on technical debt [48]. Even though the implications of technical debt are significant, and there are ways to manage it, only 7.2% of organizations methodically track technical debt [43].

1.1 Adyen

Adyen is a global company with a large codebase that has been rapidly growing and evolving for 16 years. Both business managers and developers have had to deal with technical debt, which makes this company an interesting target for further research into how a large company accrues, reduces and manages technical debt.

Adyen is a technology company founded in 2006, which specializes in a host of different payment methods, data enhancements and financial products in a single solution[1]. Adyen offers merchants an all-in-one flexible and secure payments platform to accept payments globally. These transactions can occur through e-commerce, mobile and point-of-sale products. The company partners with global card schemes, such as Visa and Mastercard, while also offering local payment solutions, such as iDEAL in the Netherlands. Adyen serves companies of all sizes, from small and medium-sized companies to large global enterprises. The company is still growing and expanding, now counting over 2800 employees world-

[1] https://adyen.com/about

wide. The company went public in 2018.

Adyen has had a long lifespan with rapid growth. The monorepo, which is mainly written in Java, consequently has grown into a large, complex codebase. Technical debt has been incurred over its lifespan, which causes problems in internal code quality and the development process. Refactoring is the main practice related to reducing technical debt [33]. Refactoring is being done by developers, but the reason behind the refactoring is largely unknown. The refactoring could be done to make requirements more easy to implement [30, 39], make code more readable [36, 31] or make the code less fault-prone [31]. Understanding the behavior of developers in doing refactorings is therefore interesting, since it could be the main driver behind avoiding or reducing technical debt.

1.2 Research questions

In this paper we will aim to research technical debt in the context of a large company, and to see if we can use gamification to reduce or avoid technical debt in the system. We will use the following research questions to investigate what problems developers face with regards to technical debt within the company and to define the boundaries of the gamification system.

RQ1: What types of technical debt impede the development process the most?
RQ2: What is the view of developers on refactoring to reduce or avoid technical debt?
RQ3: What is the behavior of developers regarding refactoring and testing efforts?
RQ4: Can we use the Delta Maintainability Model [10] to give insights into module health?
RQ5: Can we create a gamification system that helps to improve internal software quality?

We will aim to research the different aspects of technical debt within Adyen, investigate what problems developers face in their development process and answer RQ1 in Chapter 2. We will investigate what motivates developers to refactor, research their views on refactoring in general and answer RQ2 in Chapter 3. Additionally, we will investigate their refactoring and testing behaviors and answer RQ3 in Chapter 4. We will apply the Delta Maintainability Model [10], investigate whether this can be used to determine module health and answer RQ4 in Chapter 5. Finally, we will create a gamification system using the results from previous chapters to investigate whether we can improve internal software quality and answer RQ5 in Chapter 6. Our general conclusions and suggestions for future work can be found in Chapter 7.

Chapter 2

Developers' experience with technical debt

2.1 Background

Technical debt (TD) is a term first coined by Ward Cunningham [8], which describes the consequences of technical decisions that result in short-term gain, compromising on long term software quality [33]. The term *debt* is used as an analogy to describe the negative results that can occur as a result of not paying off the debt. The *principal* of the debt is the effort and accompanying cost that is needed to eliminate the debt from the system. The *interest* is the excess effort that is needed to maintain or extend the system as a result of the principal. The principal can be measured in many ways and expressed in terms of hours to fix the problems and the labor cost [18, 21]. However, determining the principal and the interest is not an easy task, since the barriers of technical debt are unclear, and many factors might contribute to problems that can be identified as technical debt [42].

Technical debt is not about code only. Software systems are complex and have many aspects that could cause problems and slowdown in the development in the long term. The following aspects illustrate some types of technical debt that might hinder the software system to move forward in the most efficient way [24]. Low code understandability might result in a developer taking more time to understand the code and to build a new feature. Low test coverage might result in more bugs, resulting in more developer time expended on writing tests and making sure the shipped product is bug-free. A long build time might result in slower development process or longer deployment time. The different types of technical debt as described by Li et al [24] can be seen in Table 2.1.

Table 2.1: Types of technical debt [24].

TD type	Definition
Requirements TD	"Refers to the distance between the optimal requirements specification and the actual system implementation, under domain assumptions and constraints"
Architectural TD	"Is caused by architecture decisions that make compromises in some internal quality aspects, such as maintainability"
Design TD	"Refers to technical shortcuts that are taken in detailed design"
Code TD	"Is the poorly written code that violates best coding practices or coding rules. Examples include code duplication and over- complex code"
Test TD	"Refers to shortcuts taken in testing. An example is lack of tests (e.g., unit tests, integration tests, and acceptance tests)"
Build TD	"Refers to flaws in a software system, in its build system, or in its build process that make the build overly complex and difficult"
Documentation TD	"Refers to insufficient, incomplete, or outdated documentation in any aspect of software development. Examples include out-of-date architecture documentation and lack of code comments"
Infrastructure TD	"Refers to a sub-optimal configuration of development-related processes, technologies, supporting tools, etc. Such a sub-optimal configuration negatively affects the team's ability to produce a quality product"
Versioning TD	"Refers to the problems in source code versioning, such as unnecessary code forks"
Defect TD	"Refers to defects, bugs, or failures found in software systems"

How much of a problem is technical debt? Bad quality code is hard to maintain and understand. This is not a trivial problem: technical debt is estimated to waste up to 42% of developers' time, low quality code contains 15 times more defects than high quality code and resolving issues in low quality code takes on average 124% more time in development [43]. It is not even the end of the story once the technical debt is in place, since the technical debt that has been incurred into the codebase results in more bad quality code. This phenomenon is best described by the term 'contagious debt' [21]. Developers feel that they can get away with 'quick-and-dirty' solutions in a dirty codebase [3]. This means that once technical debt is in place, this will result in more technical debt. The reverse is also true, as a clean, well-maintained codebase will result in fewer technical debt incurred as a result. This suggests that technical debt has a psychological effect on developers, and this has been shown in previous studies. Bad quality code results in a decrease in developers' morale [16] and as the ratio of time spent understanding code in comparison to time writing code is over 10 to 1 [49], it is of the utmost importance that the amount of bad code is kept to a minimum.

With clear downsides to bad quality code and significant positive impact of refactoring on productivity, developer morale and general software quality and maintainability, one would expect the industry to pay attention to the technical debt accumulation in their systems. The contrary seems to be the case: only 10% of business managers are involved in managing technical debt, while there is no industry-wide standard for code quality [43]. Developers are frequently forced to introduce new technical debt as new features are deemed more important than quality code [43]. In some cases, this trade-off is made willingly if the consequences are clear beforehand [23]. The developers introducing the debt are often aware of the technical debt that is present, and they indicate it within the comments of the code or in issue trackers [47]. How businesses and developers handle and deal with technical debt is therefore not clear, and should be investigated further.

What is the cause of technical debt? In some cases, technical debt can be leveraged for business advantage if it is taken on with the full knowledge of the possible consequences [23]. From a software development perspective, it can be hard to forget that customer feedback is very important to make accurate decisions on the direction forward for the company or organization [48]. If a different direction is identified, then the system needs to change anyway, and a faster decision can lead to less technical debt in the long term. This approach, however, could allow technical debt to accrue more than necessary, since allowing technical debt to be created is often the path of least resistance. It is also not always possible to know what the long term consequences will be of short term decisions, since business directions change and the maintainers of a software system might not have full knowledge of how the codebase or the business landscape will evolve.

A lack of time for bringing features to market is a recurring theme in the literature as a cause of technical debt [42, 19]. Technical debt is characterized by low visibility [20], which makes it hard to determine the technical debt that is actually in the system, holding back productivity and efficiency. Managers care about time to market, software engineers

care about software quality [48]. Without a good system to manage technical debt, it is hard for managers to determine the business value of resolving technical debt. Different software development organizations have different approaches to managing the debt. Some companies use '20% time' [48]. As the name suggests, 20% of the development time is used to improve internal software quality by, for example, fixing bugs and refactoring. The prioritization of the items can be done in different ways. It can be done on a hunch by developers, since they are the ones editing, restructuring and extending the code. Some organizations use a technical debt backlog in management tools such as JIRA[1] with a distinction between priority of items on the list. The items on the list can be identified by looking at business directions, for example based on the stakeholders' meeting once a month, or they can be identified by static code analysis tools such as SonarQube[2] and CodeScene[3]. These tools detect code smell issues that indicate there is debt that could be repaid. The issues that are brought up by static analyzers can also be displayed using *linters*, such as SonarLint[4]. Another metric that is used in the industry to prioritize debt is code coverage [48]. Even though there is tool support for managing technical debt, only 7.2% of organizations methodically track technical debt [43, 26]. This might be due to a lack of consensus in both academia and industry on what the best approach is to use the tools available [21, 27]. Another popular method of documenting technical debt is self-admitted technical debt (SATD): using words like *TODO*, *fixme* or *hack* in code comments to indicate there are problems in that code that still need to be solved [47, 11]. Practical approaches to prevent technical debt are implementing coding standards, improving the definition of done and make unit tests mandatory for new code pushed to the codebase.

The aforementioned tools are proficient in detecting code debt, which could be due to easily accessible and analyzable data [21]. Metrics such as coupling, cohesion, complexity and general code smells within the code can be easily and automatically identified [18]. Determining what needs to be done where is therefore easily identifiable. Other types of debt, such as architectural debt or technological gaps, are not found by these static analyzers [19, 22]. The most important debt might therefore not be identified, as it is suggested that architectural debt is the most important type of technical debt [33]. Another approach to finding technical debt could therefore be the measurement of process metrics, such as number of distinct developers, owner's experience and normalized lines added/deleted [34]. Madeyski *et al.* suggested that number of distinct committers should be implemented in defect prediction models [25]. In addition, Pantiuchina *et al.* suggest that the number of distinct committers is an important factor in predicting refactoring activities [31]. This could therefore be an important indicator of technical debt in certain files or parts of a software system. Technical debt is a hard problem, since it is multi-facetted and any decision in the technical development process of the software system can lead to a negative impact in the long term. Whatever the type of technical debt, refactoring to a better solution is often the antidote.

[1] https://www.atlassian.com/software/jira/
[2] https://www.sonarqube.org/
[3] https://codescene.com/
[4] https://www.sonarsource.com/products/sonarlint/

2.2 Experimental design

To answer RQ1, *'What types of technical debt impede the development process the most?'*, we did a questionnaire among Adyen developers. A questionnaire was chosen above other research techniques, such as static code analysis, since technical debt has many facets beyond code debt that could stay hidden with such a confined research technique. For example, there might be problems with the build system that would go unnoticed when only performing a static code analysis. With a questionnaire we can quantify certain aspects, such as grading the codebase on certain dimensions. Additionally, we can use open questions, where participants might highlight a diverse palette of specific problems that come to mind in the context of technical debt.

We interviewed 15 Adyen developers that did either an extract method refactoring, an extract class refactoring or both refactorings within a single commit. These refactorings were detected with RefactoringMiner [44]. These refactorings have an impact on the codebase in multiple ways. They could, for example, restructure a part of the application to adhere to the Single Responsibility Principle, they could make the code more testable, they could reduce complexity and they could make code more readable. Silva *et al.* stated that there are 11 motivations for doing an extract method refactoring, calling it the 'swiss army knife' of refactorings [39]. Asking these developers why they chose to do these refactorings could give us insight into what they deem important qualities of the codebase, such that they take the time and effort to do such a refactoring. This, in turn, could give us insight in how they approach avoiding or reducing technical debt.

The developers were interviewed on their views of the current state of technical debt within the codebase, if enough is being done to mitigate the problem and what they think are points that attention should be paid to. The questionnaire is divided in the subjects technical debt, refactoring and the refactoring performed by the developer specifically. The section of the questionnaire about technical debt can be seen in Table 2.2. The sections of the questionnaire about refactoring can be found in Chapter 3, in Table 3.1 and Table 3.2.

The results of the questionnaire will be interpreted as follows. The results to numeric questions will be summarized by the average of the numbers. We will subdivide the answers to open questions into more general themes to draw more general conclusions. Answers to Likert scale questions cannot be easily generalized to the average of the answers when mapping the answers from 'fully disagree' up until 'fully agree' to 1-5, since we cannot assume that the sentimental feeling towards these answers can be quantified in a linear way. Experts have therefore argued that the median should be used for Likert scale questions [41]. However, since the sample size of 15 participants is not large, we will also take the average of the Likert scale question answers into account.

2.3 Questionnaire on technical debt

Question	Answer type
What team are you in within Adyen?	Open
How many years have you worked in the software industry?	Numeric
Do you agree with the following statement: technical debt is a problem within the codebase.	Likert scale (1-5)
What is your grade for the code quality on these characteristics within the codebase: **readability**	Grade (1-10)
What is your grade for [...]: **extendability**	Grade (1-10)
What is your grade for [...]: **reusability**	Grade (1-10)
What is your grade for [...]: **immutability**	Grade (1-10)
What is your grade for [...]: **reliability**	Grade (1-10)
What is your grade for [...]: **testability**	Grade (1-10)
What is your grade for [...]: **documentation**	Grade (1-10)
What is your grade for [...]: **coupling**	Grade (1-10)
What is your grade for [...]: **cohesion**	Grade (1-10)
Do you agree with the following statement: if there was less technical debt in the codebase, I would be more productive.	Likert (1-5)
Do you agree with the following statement: I would feel happier during my development if there was less technical debt.	Likert (1-5)
How prevalent are these types of technical debt within the codebase: **architecture (e.g. problems with modularization)**	1 is not prevalent, 5 is very prevalent

How prevalent are these types of technical debt within the codebase: **design (e.g. wrong design patterns for implementations)**	1 is not prevalent, 5 is very prevalent
How prevalent are these types of technical debt within the codebase: **code (e.g. long code, high complexity, too many parameters in methods, etcetera)**	1 is not prevalent, 5 is very prevalent
How prevalent are these types of technical debt within the codebase: **test (e.g. no testing, not enough coverage within the tests, bad tests)**	1 is not prevalent, 5 is very prevalent
How often do you write tests for your code?	1 (Never) - 10 (Always)
Where do you test most of your code?	Local/Beta/Production
Do you agree with the following statement: I check other people's code thoroughly in the code review on correctness and code quality.	1 (I approve without looking) - 10 (I checkout the code and test every single line)
Do you agree with the following statement: code review is necessary.	Likert (1-5)
Do you agree with the following statement: my team at Adyen spends enough time and effort to resolve technical debt.	Likert (1-5)
Do you agree with the following statement: I add to technical debt in my development process.	Likert (1-5)
Do you agree with the following statement: Adyen should manage technical debt better.	Likert (1-5)
If you could advise Adyen on managing technical debt, what would you say?	Open
What frustrates you the most about technical debt in adyen-main? (in order of importance, most important first)	Open
What would help reduce technical debt in adyen-main? (in order of importance, most important first)	Open
What would improve the most if we solved all technical debt in adyen-main? (in order of importance, most important first)	Open

What causes the most technical debt? (in order of importance, most important first)	Open

Table 2.2: Section 1 of the questionnaire (technical debt).

2.4 Questionnaire results

The answers to the questionnaire in Table 2.2, Table 3.1 and Table 3.2 are confidential and can be found in Appendix **??**. The general results to the questions in Table 2.2 will be discussed here. In total, there were 15 respondents with an average of 10.3 years in software development.

As is to be expected after a system has been developed for 16 years, there is technical debt in the system. The developers agree that the technical debt is a problem. If the technical debt was reduced, then the developers would feel more productive and happier. This is consistent with the literature that says reducing technical debt improves productivity [3], and that technical debt decreases morale [16].

From the responses we can gather that readability and reliability are the best properties of the codebase. In contrast, the properties that could be improved are testability, documentation and immutability. This is an interesting observation, since one would expect reliability to be low when testability is low. Another interesting observation about testing is that technical debt in terms of testing is present, but that almost all developers indicated that they often write tests for their code. They deem code review very important, and do a thorough analysis when they are doing code review. Their responses also indicate that their team tends to spend enough effort to resolve technical debt, even though the technical debt as whole could be managed better. Since the total of 15 respondents come from a total of 12 teams, this is most likely not due to bias by only looking at a well performing team. The teams might overestimate their own efforts for resolving the debt and might underestimate their own share in creating the technical debt.

Existing debt is seen by the respondents as a source of technical debt, which is also called contagious debt [21]. The respondents also indicated that 'tribal knowledge' is a source of technical debt: unwritten knowledge within the company that is not widely known. This also leads to people 'reinventing the wheel': they think they are the first to build something, but an implementation is already present in the codebase. This leads to the same functionality being in multiple places, reducing maintainability.

The respondents also shared their ideas on how technical debt could be reduced. More time should be spent on reducing technical debt, such as 20% time for refactoring or improving internal software quality [48]. Other things that might help are more testing, better code guidelines, more insight into the problem and rewards or recognition for actually reducing

technical debt. They also mentioned that clearer ownership of certain parts of the system are desirable, so people take more responsibility for reducing technical debt in their part of the codebase.

RQ1: What types of technical debt impede the development process?
It seems that tests, testability and documentation could be improved. Developers might keep 'reinventing the wheel' if they do not know that something already exists within the codebase, leading to multiple implementations of the same thing. Solutions to reducing or avoiding technical debt are more testing, better guidelines, more insight into problems, more rewards or recognition for paying off debt and reserving time to pay off technical debt.

Chapter 3

Developers' view on refactoring

3.1 Background

The main practice related to technical debt principal repayment is refactoring [33]. Refactoring is restructuring software code without changing its behavior [12]. The goal of refactoring is to increase code quality and code structure. This goal is achieved by a number of refactoring techniques, such as the technique *'extract method'*, which is seen as the swiss army knife of refactoring [39]. This technique extracts some code from a method and encloses it into a new method, making the code more readable while reducing the complexity. There are many more refactoring methods and techniques, which can be automatically identified by tools such as RefactoringMiner [44].

The benefits of refactoring are supported by the literature. It is widely believed that refactoring improves developers' productivity and software quality [17]. The re-organizing of the source code to create more cohesive components within the system is correlated with higher readability [37]. Higher readability is also correlated with fewer bugs or other issues in the system [36, 3, 17]. When time was spent on refactoring during one development iteration, the productivity becomes significantly higher during the next iteration [3, 28]. The efforts of refactoring to improve the internal quality is even related to improved developers' morale [16]. Refactoring is, however, not only driven by the urge to improve the internal structure. Some research suggests that refactoring is mainly driven by changes in requirements, such as implementing new features and fixing bugs, and not due to the presence of code smells [39, 30].

All these factors suggest that refactoring has major benefits and should be taken serious by all developers in the software development field. However, there are also some drawbacks. If there is no adequate testing of the source code that is being refactored, then the resulting refactored code might be prone to errors. There is even evidence for this phenomenon: Weißgerber and Diehl found that in a case study of open source software systems, a high ratio of refactoring is often followed by an increase in bug reports [46]. On the other hand, Ratzinger *et al.* and Ammerlaan *et al.* found that the number of refactorings was correlated

with less defects in the system [35, 3]. It therefore seems that there is some controversy around refactoring and reducing the number of bugs or defects, but in general refactoring seems to be worthwhile.

To assist the developer in refactoring, and to reduce the effort expended by developers to do refactorings, many tools have been developed. Even the IDE the developer uses can have an impact on the number of refactorings a developer does: IntelliJ IDEA users perform more automated refactorings than Eclipse and Netbeans users [39]. These automated refactorings are often low-impact, and just make the activity of refactoring a tiny increment easier. An example of this is a dialog that extracts the selected code and allows you to give a new name to the created method. Shahidi *et al.* have attempted to automatically identify extract method refactoring opportunities and automatically generate method names for the extracted methods [38]. This can be done by a rule-set, but Zaitsev *et al.* have implemented a machine learning approach to naming methods [49]. These refactoring opportunities are relatively low-level. Higher-level automatic refactoring approaches are mostly in the field of search-based refactoring. Search-based refactoring restructures the refactoring opportunities as combinatorial optimization problems [29]. An example of this is the remodularization of a software system by moving classes to increase cohesion and decrease coupling. A fitness formula is determined and an algorithm optimizes for the highest fitness value, by trying out different refactorings.

To answer RQ2, *'What is the view of developers on refactoring to reduce or avoid technical debt?'*, we will look at the results of the refactoring section of the same questionnaire discussed in Section 2.2.

3.2 Questionnaire on refactoring

3.2.1 Refactoring

Question	Answer type
Refactoring is an important development activity.	Likert (1-5)
When I feel I should refactor code, I do it.	Likert (1-5)
Refactoring improves code readability.	Likert (1-5)
Refactoring improves performance.	Likert (1-5)
Do you agree with the following statement: I am afraid of introducing bugs into the system when I refactor.	Likert (1-5)

How many hours per week do you estimate to spend on refactoring?	Numeric
What are reasons for you to refactor code?	Multiple choices: • Reduce complexity • Increase readability/understandability • Be able to implement a new feature better • Be able to write tests better • Fix bugs • Avoid bugs in the future • Remove code smells • Remove duplicate code • Other

What are reasons NOT to refactor source code?	Multiple choices: • Might introduce bugs • Insufficient tests of the source code • I don't understand all the source code • Understanding all the source code to do the refactoring takes too much time • Do not want to make code reviews for people reviewing my code harder than they have to be • I don't care about technical debt • Not worth the time • I have no time • Don't want to deal with the merge conflicts • Risk of over-engineering • Other.
Do you adhere to the Boy Scout rule: *leave your code better than you found it*.	Likert (1-5)
Do you agree with the Broken Windows Theory: *when there are problems in the codebase, developers are less motivated to write the best code they can*.	Likert (1-5)
If I had a tool that could deliver specific refactoring recommendations, then I would use it.	Likert (1-5)
How do you ensure program correctness after refactoring?	Open
What do you think is the best reason to refactor?	Open

Table 3.1: Section 2 of the questionnaire (refactoring).

3.2.2 Specific refactoring

Question	Answer type
What is the hash of your specific commit? You can find this in my Mattermost message.	Open
What was the motivation behind your specific refactoring? Please be specific.	Open
(Optional) Do you have any other insights in refactoring and technical debt at Adyen that you would like to share?	Open

Table 3.2: Section 3 of the questionnaire (specific refactoring).

3.3 Questionnaire results

The main practice related to technical debt principal repayment is refactoring [33]. The developers that did a refactoring deem refactoring an important development activity and spend an average of 5.1 hours per week on refactoring.

The main reason for refactoring is mainly to improve code structure. The extract method and extract class refactorings were mainly performed to improve readability and to remove/avoid code duplication and to make the classes and methods adhere to the Single Responsibility Principle [4]. This gives the impression that the developers are involved in reducing or avoiding technical debt in their development process.

It is also interesting to note that code review is also a reason for people to do a specific refactoring, showing that multiple people are involved in the process of avoiding deterioration of the codebase. Code review is also seen by these developers as a very important process, making it likely that developers deem reducing technical debt as important.

In some cases, a refactoring is needed to implement a new feature, which has been supported by Silva *et al.* who state that refactoring activity is mainly driven by requirements for new features or bug fixes [39]. However, for Adyen developers, improving code structure seems to be the main driver. This is also supported by their adherence to the *Boy Scout Rule*: they try to leave the code better than they found it. They also strongly agree with the *Broken Windows Theory*: when there are problems in the codebase, developers are less

motivated to write the best code they can. This could also be a good motivator for putting a heavy emphasis on clean code, since a clean codebase motivates clean code creation in the future.

RQ2: What is the view of developers on refactoring to reduce or avoid technical debt?
Developers mainly refactor to improve code structure in terms of readability, to remove/avoid code duplication and to make the classes and methods adhere to the Single Responsibility Principle. The developers deem refactoring an important activity, they refactor when they feel that it is needed and they suggest refactoring in code reviews. There is a fear of introducing bugs when refactoring.

Chapter 4

Investigating refactoring and testing efforts

4.1 Experimental design

To answer RQ3, *'What is the behavior of developers regarding refactoring and testing efforts?'*, we will look at the codebase itself, the activity in the Git repository, the lead time of tickets, and the refactoring and testing efforts. This will give us large resources of quantifiable data where we might be able to discover if there are predictive behaviors for developers to do or not do a refactoring or touch test files.

In the rest of this section, we will describe the methods used to gather data on code committed to the Git repository, files, classes, methods, authors, specific refactorings and tickets that are linked to the commits. We need a few details of the codebase to do a thorough analysis of the refactoring efforts of developers and the evolution of the codebase in general.

- **Commits and commit metadata.** Commit metadata includes hash, date and commit message.
- **Files.** For each commit, we need to know which files have been changed. Each commit is a snapshot that contains file data, such as file length.
- **Methods and classes.** For each commit, we track whether methods were added and what the metrics are of those methods, such as method length and complexity.
- **Authors, teams and module owners.** Every author is connected to a team within Adyen. There are metadata files within the codebase to indicate what teams own what modules. Almost all files within the codebase are connected to a module. With this data, we know whether an author commits code to a file that he or she does not explicitly own.
- **Refactorings.** For each commit, we know whether a certain refactoring has taken place. RefactoringMiner [44] is used to scan the Git history for refactorings.
- **Tickets.** Within Adyen, a system is used to manage tickets and assign them to people to work on them. The most important features of the tickets is the assigned date of the

ticket to investigate the lead time of the ticket.

- **Delta maintainability.** The Delta Maintainability Model indicates for every commit whether the changes are low risk or high risk within a certain domain, such as unit size, unit complexity or unit interfacing. For every commit, these metrics are calculated and stored as a value between 0 and 1. More on the Delta Maintainability Model can be found in Chapter 5.

Since much of the data is relational, we used a SQL database to store and link the data. The data was gathered from the Adyen Git repository from 01-07-2020 to 01-07-2022 *(DD/MM/YYYY)*.

4.1.1 Data acquisition

All data acquisition, preprocessing and entering into the database was done in Python 3.8 [45].

Git repository

The data from the Git repository was gathered with GitPython[1]. The following data was gathered from the Git repository:

- **Commits**: hash, commit message, datetime, author and changed `*.java` files in commit.
- **Authors**: author name and number of commits before the data acquisition starting date.
- **Files**: filename.

File and method metrics

Every `*.java` file change within a commit is evaluated as a new snapshot of the file. For every snapshot, the file and the methods in this file undergo a static code analysis performed by Lizard[2]. The following data was gathered for each snapshot:

- **Files**: number of lines of code and token count.
- **Methods**: number of lines of code, cyclomatic complexity, number of parameters, token count and containing class.

Refactoring data

The open source library RefactoringMiner [44] was used to detect refactorings in the Git history of the Adyen codebase. Every refactoring was characterized by at least a type and a file, and when possible the method that was involved in the refactoring. The refactoring types we will investigate are as follows. The precision and recall metrics are taken from the

[1] https://gitpython.readthedocs.io/en/stable/
[2] https://github.com/terryyin/lizard

official GitHub page[3] as of 17 October 2022.

Refactoring type	Precision	Recall
Extract Method	0.999	0.969
Extract Class	1.000	1.000
Extract Interface	1.000	1.000
Extract Subclass	1.000	1.000
Extract Superclass	1.000	1.000

Table 4.1: Refactoring types

Correlation analysis

To determine if there is a linear correlation between 2 variables, the Pearson correlation coefficient is used [7]. A correlation on the intervals [0.1, 0.4) and (-0.4, -0.1] is considered weak, a correlation on the intervals [0.4, 0.6) and (-0.6, -0.4] is considered moderate, a correlation on the intervals [0.6, 1) and (-0.6, -1] is considered strong and a correlation of 1 or -1 is considered perfect [1].

4.1.2 Schema

The resulting schema of the database can be seen in Figure 4.1.

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

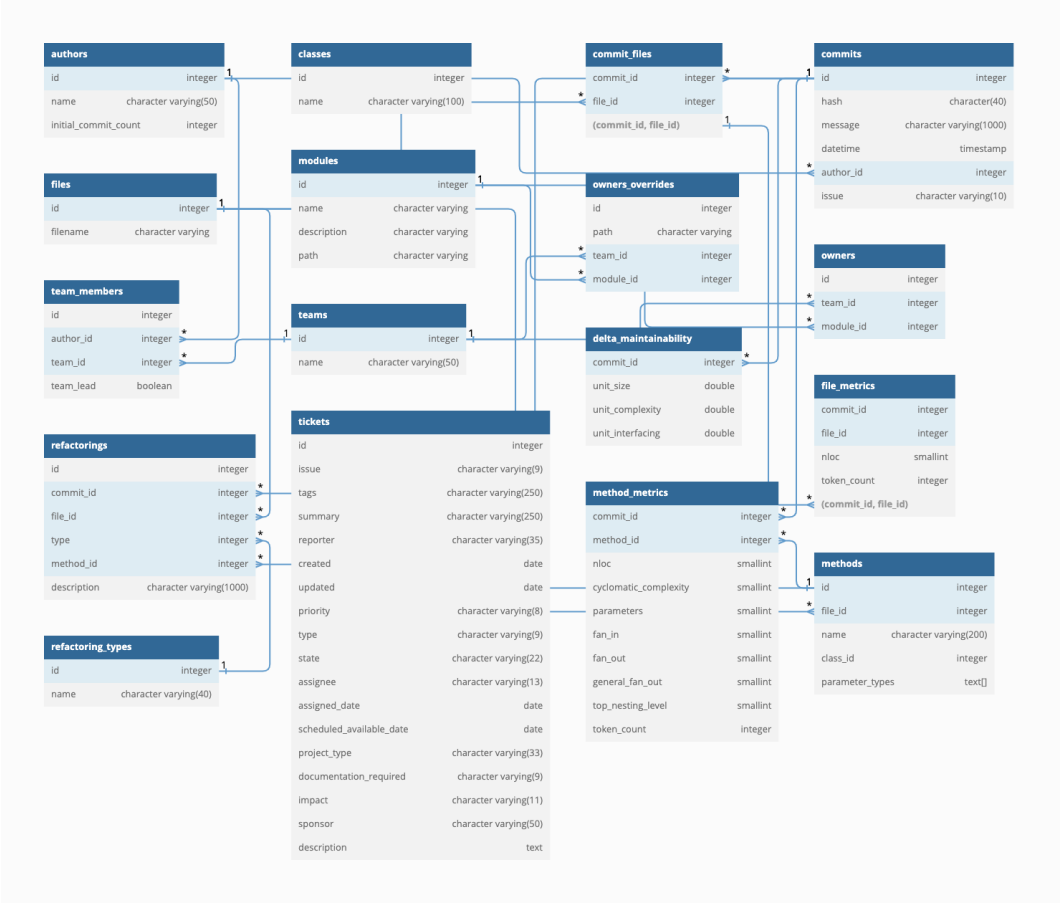

Figure 4.1: Database schema

4.2 Results

In this section, we will look at the efforts of developers in terms of testing and refactoring, according to the refactoring types in Table 4.1.

4.2.1 Refactoring and testing

Figure 4.2 shows the percentage of commits that contains a change to a test file of every author on the y-axis, while the x-axis shows the percentage of commits that contain a refactoring of the same author. The Pearson correlation of these metrics is 0.36, indicating a weak correlation.

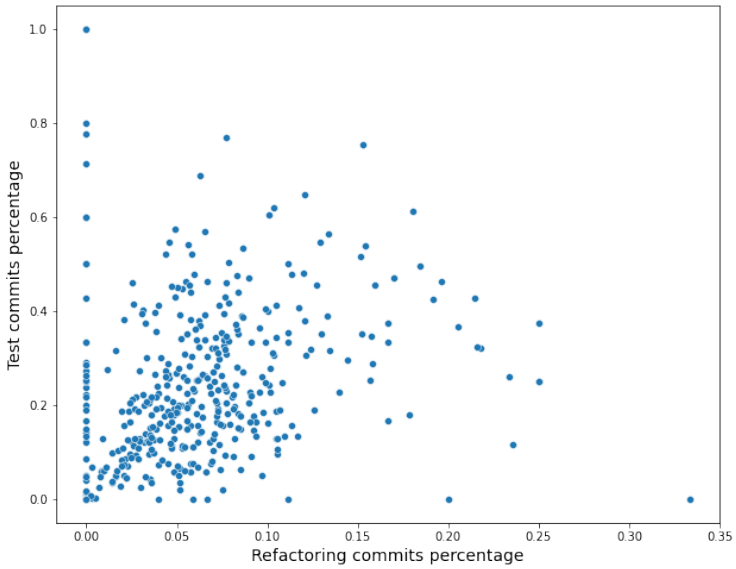

Figure 4.2: The percentage of commits that includes a test or a refactoring for every author.

Since there is a weak correlation, one might say that the developer that tends to write more tests, might also be more inclined to do refactorings.

In the questionnaire discussed in Section 3.1, participants indicated that they write tests to check their refactoring on correctness. Table **??** in confidential Appendix **??** shows the number of commits that contained a refactoring or changed a test file. This table indicates that in nearly half of the commits where there was a refactoring, a test file was also changed. The Chi-Square test indicates that a test is more likely to be changed if a refactoring has occurred, with a p-value < 0.001.

4.2.2 Lead time

Figure 4.3 shows the number of refactorings in a ticket and the time it took to complete the ticket. The Pearson correlation of lead time and number of refactorings in a ticket is 0.20, which indicates a weak correlation.

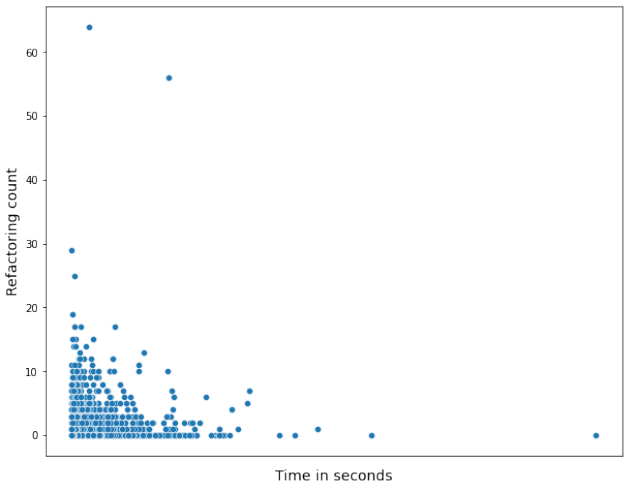

Figure 4.3: Refactoring count and lead time of tickets.

Since there is a weak correlation, there does not seem to be a significant impact on lead time if refactoring is involved. This means that developers might not need to spend more time on a ticket if a refactoring is required. Additionally, if refactorings are required due to bad code or a restructuring of the code, then this does not significantly impact lead time.

A ticket that requires the developers to work in long files might take more time than a ticket that only involves short files. Figure 4.4 shows the lead time with the average file length and the lead time with the maximum file length, respectively.

The Pearson correlation between lead time and average file length is 0.05, while the Pearson correlation between lead time and maximum file length is 0.18. These correlations are weak, so we cannot draw the conclusion that the requirement to work in long files is an indicator that the lead time of a ticket will be increased.

High file complexity might lead to a developer taking longer to make a change or add features to the code. Figure 4.5 shows the lead time with the average file complexity and the lead time with the maximum file complexity, respectively.

The Pearson correlation between the lead time and the average file complexity is 0.06 and the Pearson correlation between the lead time and the maximum file complexity is 0.11. This indicates that there is no relationship between the lead time of a ticket and the com-

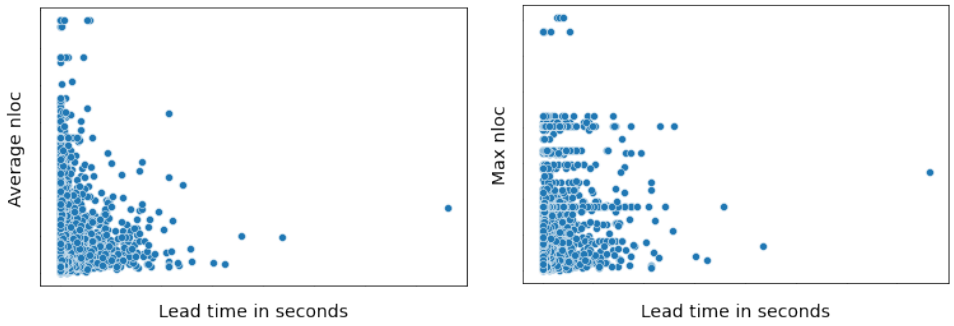

Figure 4.4: Lead time of ticket, along with average and maximum file length.

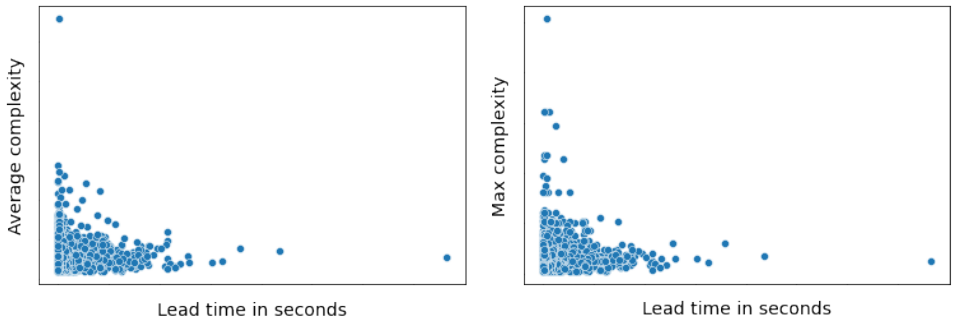

Figure 4.5: Lead time of ticket, along with average and maximum file complexity.

plexity of the code a developer has to work in in order to finish a ticket.

4.2.3 Author experience

Pantiuchina *et al.* state that the fewer experience a developer has with the code he is working on, the lower the chance is that that developer refactors the code [31]. For this analysis, they use the number of commits as the metric for developer experience. Our data shows that the percentage of commits that contains a refactoring is not significantly correlated with the total number of commits to the codebase by that specific author.

The Pearson correlation between commit count and refactoring commit count is 0.74, indicating that there is a strong relationship. This is logical, since the more developers commit code, the more likely it is that a refactoring is among those commits. On average, the developers do a refactoring once every 16 commits.

4.2.4 File changes

One might say that the more a system changes, the more technical debt accrues within that system. This might mean that the more a file changes, the more likely that file is to be refactored. The distribution can be seen in Figure 4.6 and with a Pearson correlation of 0.11, we can say that this is not necessarily the case.

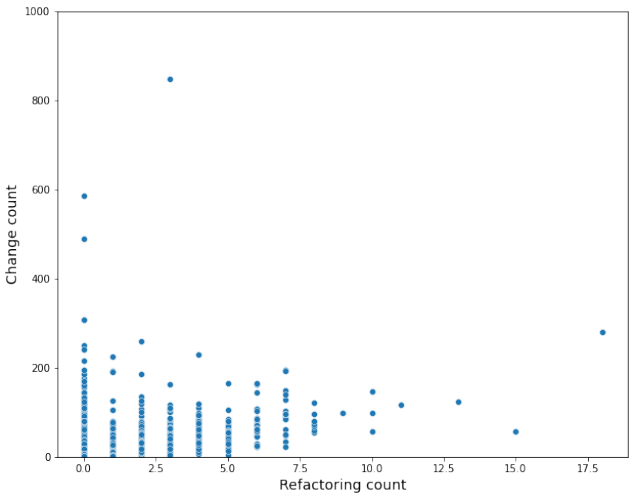

Figure 4.6: Number of file changes and number of refactorings per file.

Another theory is that the more authors a file has, the more complex code can become due to different coding styles or developers not understanding all the code in the file. This might have an impact on how often a file gets refactored. In Figure 4.7, we can see the percentage of commits that had a refactoring as a percentage of the total changes to that file. In the same figure, we can see the number of authors of the file. The Pearson correlation of these variables is 0.04, indicating that an increased number of authors of a file has no predictive value to the number of refactorings in this file.

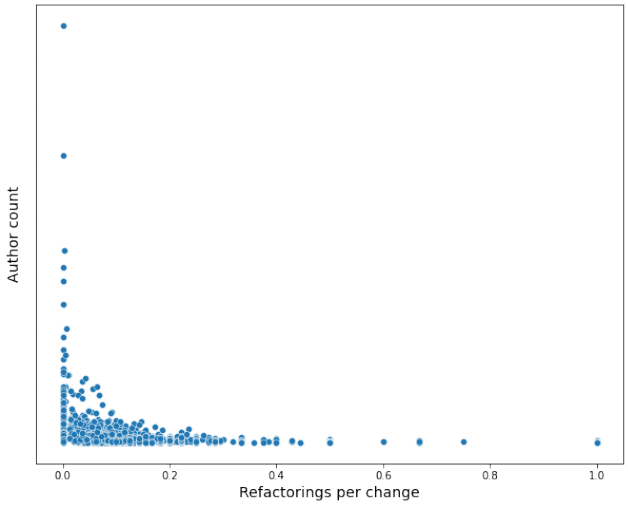

Figure 4.7: Percentage of refactoring changes along with the number of authors of a file.

4.2.5 Method complexity

In Figure 4.8, we can see the average complexity change of a method that occurs along with a refactoring type. Extract method reduces the complexity the most, on average by 2.2.

In Figure 4.9, we can see what the average complexity is of methods where a certain refactoring type occurs. When a method gets extracted from another method, the average complexity of that method is 7.9. We can also see that 'Add Variable Modifier' and 'Remove Variable Modifier' happens relatively often in methods with a high complexity. This is exclusively the addition and the removal of the *final* keyword to variables. 'Add Parameter' and 'Remove Parameter' also occur relatively often in methods with high complexity.

RQ3: What is the behavior of developers regarding refactoring and testing efforts?
There is a weak correlation between writing refactorings and editing test files. When a refactoring occurs, there is a higher chance of a test file being changed. When a refactoring is done, it might indicate that the lead time of the corresponding ticket is higher. The file length or the complexity of the methods in the files that are changed during the development process do not seem to correlate with refactorings.

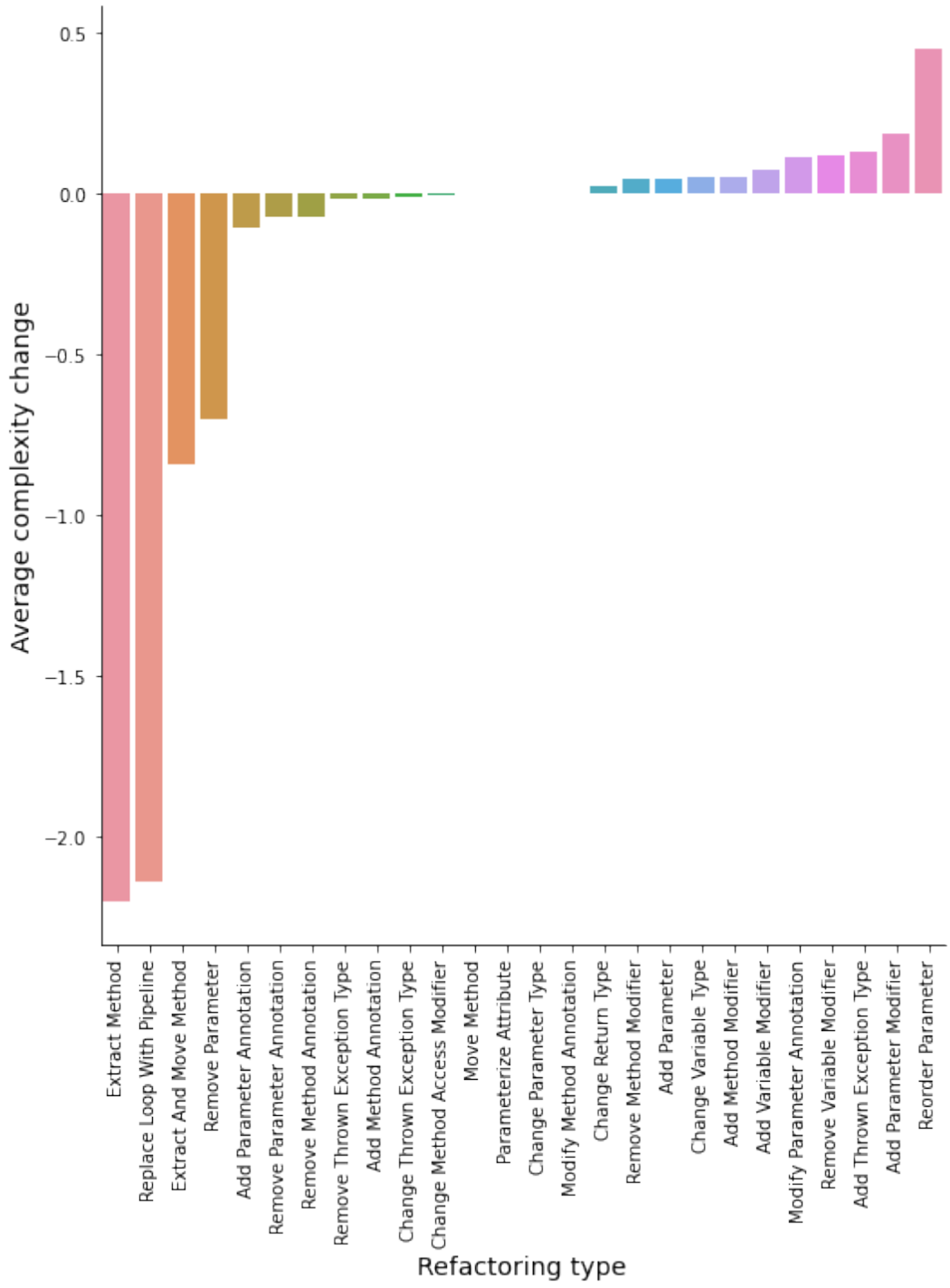

Figure 4.8: Change to complexity of methods by refactoring type.

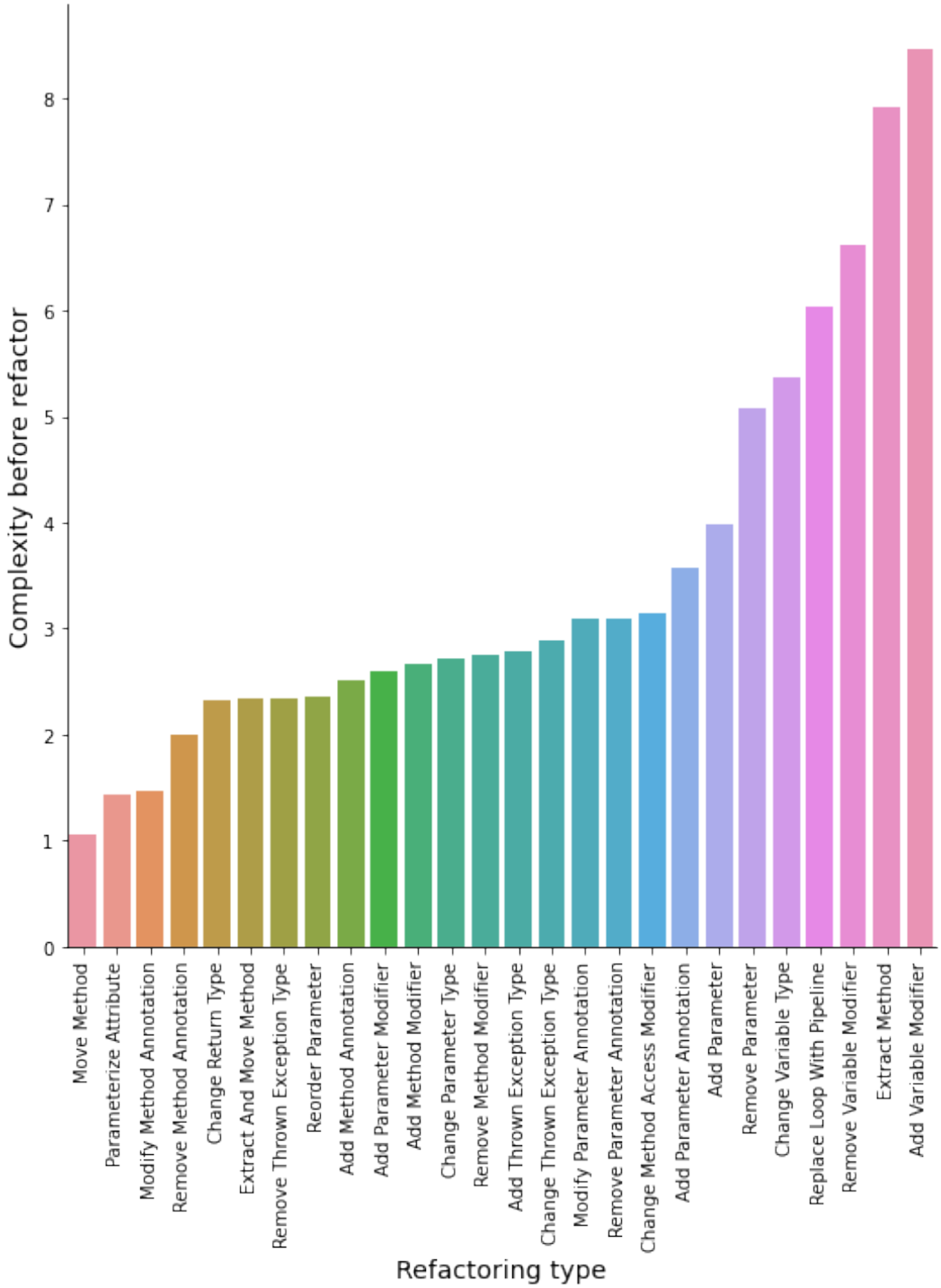

Figure 4.9: The complexity of a method before a refactoring type occurs.

Chapter 5

Insight in module health with the Delta Maintainability Model

5.1 Background

The Delta Maintainability Model (DMM) is a model that is used to assess what the effect of a commit is on the maintainability of the code. The Delta Maintainability Model uses the risk thresholds as described by Alves *et al.* [2] to determine a delta risk profile, which in turn measures whether the changes of the commit can be classified as *good* or *bad* [10]. The delta risk profile takes three properties into account:

- **Unit size:** method length in lines of code. Low risk threshold 15 lines.
- **Unit complexity:** method cyclomatic complexity. Low risk threshold 5.
- **Unit interfacing:** method number of parameters. Low risk threshold 2.

The DMM is used to investigate whether we can see deterioration of the codebase in terms of these properties. For every commit, a value is calculated from 0.0 (all changes are risky) to 1.0 (all changes are low risk). Since this value is only describing of a single commit, we will map the values of these properties to -0.5 to 0.5 and use the accumulation of these values over time to characterize modules within the codebase. We expect the cumulative value to be above 0 if the module has mostly maintainable changes, and below 0 if the module has mostly unmaintainable changes. We would expect any modules with a cumulative DMM property value below 0 to have considerable technical debt.

PyDriller is a Python framework that can be used to mine software respositories [40]. It has an open source implementation of the Delta Maintainability Model to assess the risk profile of commits.

5.2 Applying the DMM to the codebase

This DMM measures risk values of unit size, unit complexity and unit interfacing. These indicators are measured for every commit and measured with a value between 0.0 (high risk commit) and 1.0 (low risk commit). The distribution of the values of these commits is

shown in Figure 5.1.

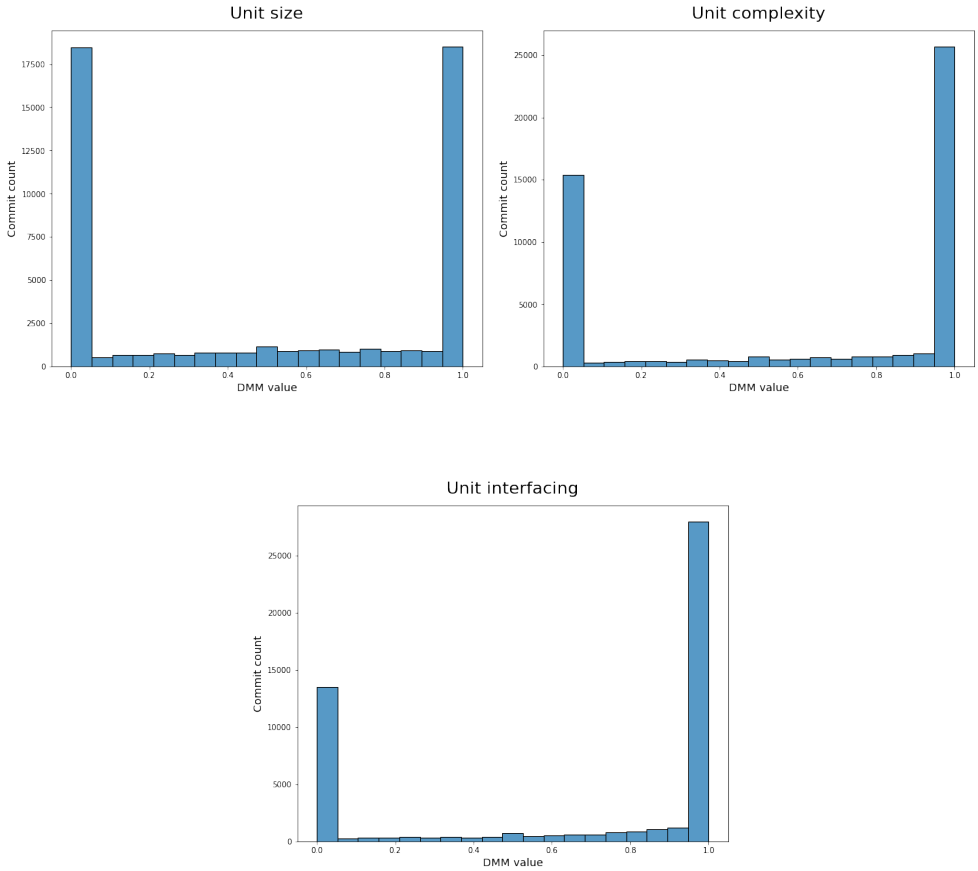

Figure 5.1: Distribution of delta maintainability values over commits.

The distributions in Figure 5.1 show that most of the DMM values are 0 or 1. This indicates that either all changes within the commit are risky, or all changes are non-risky. The distribution for unit size is not skewed to either side, but in terms of unit complexity and unit interfacing we can see that the distribution is skewed into mostly non-risky changes.

5.3 Effect of refactoring on the DMM risk values

When applying the refactoring types from Table 4.1, one can expect that the DMM values would be closer to 1 than 0. This is because these refactoring types should reduce unit size, since some code is extracted away into other methods, classes or interfaces. The complexity should be reduced for the same reason. Overall the code should be more simple and more readable, which could also reduce the unit interfacing. Table 5.1 shows the average unit size, unit complexity and unit interfacing of commits where one of these refactoring types

is applied.

Refactoring type	Occurrences	Avg. size	Avg. complexity	Avg. interfacing
No refactoring	29,857	0.50	0.60	0.67
Extract Method	5,959	0.69	0.80	0.66
Extract Class	493	0.70	0.79	0.74
Extract Interface	57	0.71	0.79	0.78
Extract Subclass	34	0.71	0.82	0.77
Extract Superclass	170	0.69	0.80	0.69

Table 5.1: DMM values in commits with refactorings

Compared to the commits that had no refactoring, it seems like the commits with a specific high-level refactoring made the underlying code more maintainable. This means that the commits with these refactorings improved in terms of the lines of code within methods, cyclomatic complexity of methods and number of parameters of methods.

5.4 Using the DMM as a measurement of module health

If we map the values between 0 and 1 to -0.5 and 0.5, we can aggregate the DMM values to see improvement or deterioration of these metrics within modules of the codebase. The results are in Figure 5.2. The modules have been anonymized in this figure, the real names of the modules can be found in Figure **??** in Appendix **??**.

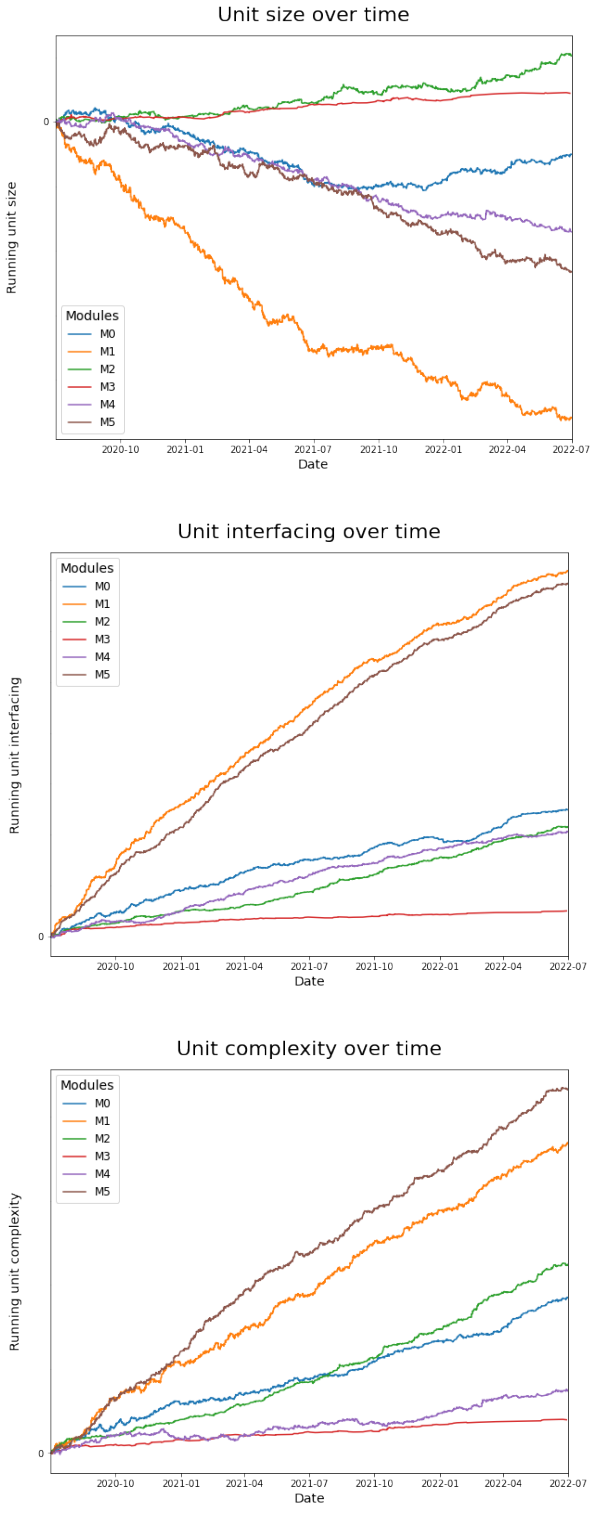

Figure 5.2: Aggregated DMM values over time.

In all the modules that were analysed, unit interfacing and unit complexity went up over time. This is consistent with the distribution in Figure 5.1, as there is an abundance of DMM values of 1 for these metrics. Some modules have a steeper slope, this is because these modules are changed more frequently in commits. In Figure 5.1, we can see that the unit size metrics are more balanced. This results in the different graphs for unit size in Figure 5.2.

Some modules have a positive value on 2022-7, such as M2 and M3. Other modules have a negative value, such as M1 and M5. You can see that M2 and M3 mostly have a positive slope, while M1 and M5 mostly have a negative slope. From this we can conclude that M2 and M3 mostly received commits that were *low risk* to unit size, so the commits did not affect the length of the methods in a negative way. Modules M1 and M5 mostly received commits that were *high risk* to unit size, so the commits impacted the method lengths negatively. The course of the unit size deterioration of module M1 and the course of the unit size being guarded in modules M2 and M3 can be an indication of the *Broken Windows Theory*: when there are problems in the codebase, developers are less motivated to write the best code they can [3].

RQ4: Can we use the Delta Maintainability Model to give insights into module health?
The Delta Maintainability Model seems to give an indication of module health when its unit size risk values are added over time. If the aggregate value is low, then this module might risk changes that are bad for unit size, i.e. edits that make methods longer or edits that add long methods. A higher aggregate unit size might indicate better module health with low method lengths. This might in turn mean that future edits will be of low risk to unit size, which means that developers will put in more effort to not let method size get out of hand.

Chapter 6

Using gamification to improve software quality

6.1 Background

With the lack of a concrete process to manage technical debt and with organizations relying on developer's prioritization based on their subjective feeling [48], gamification might be a solution. Gamification seeks to implement game elements in certain tasks, to improve the user's engagement, motivation and performance [15, 32]. In doing so, the gamification of the process makes the tasks themselves more attractive and might promote more of those activities. A simple way of implementing a gamification system is to assign points to behaviour you would like to see repeated. However, the value of points or a rank is not inherent in the points themselves, it is within the implied reputation, identification with a certain group or social approval that the points represent [9]. Gamification in software engineering could look like earning points for writing documentation, better commit messages and writing more tests. A good example of gamification is StackOverflow[1]: users gain points by answering community questions in a well-formulated and accurate way.

The main point of gamification is to motivate developers to repeat behaviour that is desirable. Research suggests that the motivation of software engineers is reported to have the largest impact on productivity and software quality management, and that this motivation is the hardest to manage [14]. The motivation of software engineers has been studied [6], but people management has still been considered a key issue [15]. If gamification can improve developers' motivation, then it will improve technical debt, improve productivity and reduce turnover within the company.

Gamification in software engineering is at an early stage, and not yet fully mature [15]. This might be due to the nature of gamification, where a company-wide application might be hard to implement both in the systems and in the working procedures of the employees. This might also be due to the infancy of the concept within the field.

[1] https://stackoverflow.com/

6.2 Code Karma

'Code Karma' is a gamification implementation that we propose for the identification of beneficial internal code modifications. The system works by the accumulation of points for certain activities, to see if showing these points in a ranking will increase developers' motivation for these tasks by coupling it to their perceived reputation within the team. In the survey described in Section 2.2, developers indicated that the main problems with technical debt in the codebase are related to testing/testability and documentation. They also indicated that they refactor to improve readability/understandability and to avoid or remove duplicate code. Code debt such as long code or high complexity also seems to be relatively prevalent. Code Karma aims to motivate developers by awarding them points for making modifications to the codebase that improve these properties. The goal of this experiment is to investigate if we improve what we measure, even more so when coupled to a reputation within the team. Additionally we will be able to determine if we can make the effort of improving internal software quality a fun activity. The Code Karma experiment ran for 4 weeks, while reporting the results to the participants every week.

6.2.1 Karma accumulation

Three properties of Git commits are tracked to determine how much karma is awarded for a commit.

- **Test lines added**. The number of lines added to a test file is determined by the lines added to a file with the filename `*Test.java`. Empty lines are not counted. If there are added lines that were moved from another test file, then the lines are not counted. This could happen when a class is extracted and the tests are moved to a new test file. This would not be new test lines, since they test the same functionality.
- **Docblocks written**. Docblocks are identified by an added line that starts with `/*`, which indicates a multi-line docblock. Single line comments are not counted, since there are cases where they do not explain the code or the effect of the code. They could, for example, indicate self-admitted technical debt with comments such as *TODO* [47], or they could indicate characteristic comments in tests such as *given*, *then* and *when*. Moved docblocks are not counted.
- **Refactorings**. The refactorings are identified by using RefactoringMiner [44]. Since the goal of promoting refactoring behavior is mainly to improve readability/understandability and improve code structure, only the refactoring types from Table 4.1 are measured.

No negative karma is awarded to avoid motivating the developers to do things they feel are not necessary. The calculation for karma per commit is as follows.

```
karma = (docblocks * 2) + (test_lines_added / 10) + (refactorings * 5)
```

We tried to award karma for a specific activity in accordance with the effort it takes to do a specific activity. For example, in this calculation, doing a refactoring is deemed harder than writing a docblock.

6.2.2 Achievements

To invoke more motivation within the developers to put effort in the aforementioned activities, achievements were awarded to developers ranking the highest in the activities. The following achievements could be earned.

- First, second and third highest karma earner in a week.
- Most test lines added in a week.
- Most docblocks written in a week.
- Most refactorings done in a week.

6.2.3 Participant groups

Within Adyen, there were 2 teams that participated in the experiment. Group 1 had 20 participants, group 2 had 10 participants. The results were split between the teams, such that team members could only see the anonymized results of team members. Additionally, each team had a karma goal of 750 karma. The progress towards this goal was reported every week.

6.2.4 Displaying the results

The value of points or a rank is not inherent in the points themselves, it is within the implied reputation, identification with a certain group or social approval that the points represent [9]. Therefore we show a ranking of the karma earned within the team, with an indication of where the developer is within the distribution. The names are indicated with random nicknames, such as *Vertex* and *Ripple*, within the distribution, to ensure anonymity of the developers. This is done to avoid negative effects of the experiment, such as damaging a developers' reputation by showing this person low in the rankings. There can be good reasons for a developer to not write tests, docblocks or do refactorings, so the experiment aims to provoke only positive emotions within the participants.

Every week, all participants received a message in the internal company messaging system. In this message, the player received their nickname, the activities they performed in the last week that attributed to their karma, their total karma, their achievements, the progress towards the team goal karma and the distribution of karma across the team. An example of this message is shown in Figure 6.1.

6.3 Questionnaire to evaluate Code Karma

To evaluate the experience of developers with Code Karma and to see what effect it had on the participants, a questionnaire was distributed to the participating developers. The questions in the questionnaire are in Table 6.1.

Code Karma update 🚀

Hi [redacted]

Last week, you did the following things to reduce technical debt:

- You added 220 test lines.
- You wrote 2 docblocks.
- You did 1 refactoring (extract method/extract class/extract interface).

This adds **31.0 karma** to your score! This brings you to a grand total of **65.3 karma**. You can see your position in the rankings below. Your nickname is **Duckling**.

Achievements

- 🥈 2nd place highest karma in a week.
- 🎖️ Most docblocks written in a week.

Team karma goal progress

72.3% (542) 100% (750)

Ranking

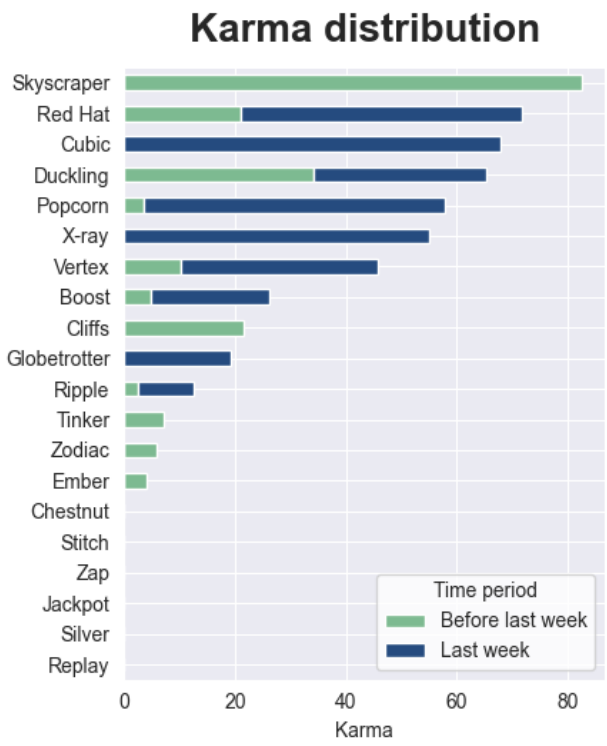

Figure 6.1: Example karma update message to a developer after week 2.

Question	Answer type
What team are you in within Adyen?	Multiple choice
What was your nickname?	Open
How many years have you worked as a software developer?	Open
On a scale of 1-10, how competitive are you?	Grade (1-10)
On a scale of 1-10, how concerned are you with reducing technical debt and improving the quality of the codebase?	Grade (1-10)
Do you agree with the following statement: Code Karma was a positive experience.	Likert (1-5)
(Optional) Elaborate on your previous answer if you wish.	Open
Do you agree with the following statement: the resulting karma distribution was surprising to me.	Likert (1-5)
(Optional) Elaborate on your previous answer if you wish.	Open
Do you agree with the following statement: I expected to obtain more karma than I received.	Likert (1-5)
Do you agree with the following statement: I think Code Karma has a positive impact on the codebase.	Likert (1-5)
Do you agree with the following statement: Code Karma motivated me to write more tests.	Likert (1-5)
Do you agree with the following statement: Code Karma motivated me to write more docblocks.	Likert (1-5)
Do you agree with the following statement: Code Karma motivated me to do more refactorings.	Likert (1-5)
Do you agree with the following statement: the **ranking distribution** motivated me to obtain more karma.	Likert (1-5)

Do you agree with the following statement: the **team karma goal** motivated me to obtain more karma.	Likert (1-5)
Do you agree with the following statement: the **achievements** motivated me to obtain more karma.	Likert (1-5)
Do you agree with the following statement: the formula for obtaining karma is balanced. The formula is as follows: *(docblocks * 2) + (test_lines_added / 10) + (refactorings * 5)*	Likert (1-5)
(Optional) Elaborate on your previous answer if you wish.	Open
Do you agree with the following statement: using nicknames was better than using real names of participants.	Likert (1-5)
Do you agree with the following statement: I would like Code Karma to continue giving the team weekly updates after the end of the experiment.	Likert (1-5)
Do you agree with the following statement: Code Karma should be used more within Adyen.	Likert (1-5)
What do you like about Code Karma?	Open
What do you NOT like about Code Karma?	Open
Are there features that should be added to Code Karma?	Open
Is there any other feedback you would like to give?	Open

Table 6.1: Questionnaire to evaluate Code Karma.

6.4 Results

The Code Karma experiment was evaluated by both a survey and by comparing the refactoring efforts, test lines added and docblocks written in the month of the experiment, compared to the 3 months before the experiment. The results of the teams after 4 weeks are shown in Figure 6.2.

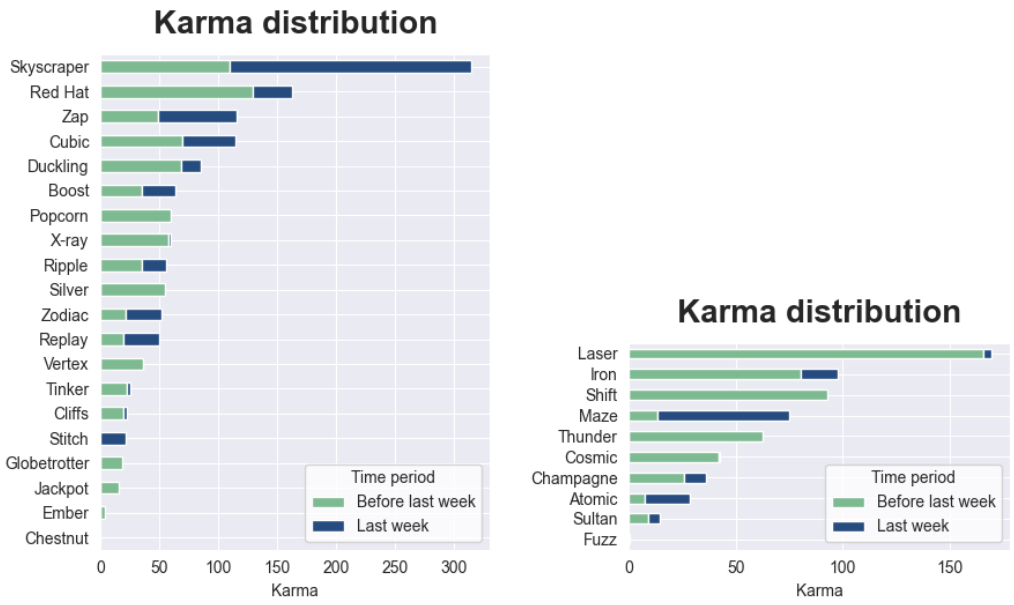

Figure 6.2: Code Karma distributions of 2 teams

6.4.1 Questionnaire results

Only the results of the grade-scale and likert-scale questions are shown here. These results and the answers to the open questions will be discussed after the survey results. These are the results of 17 participants of the code karma experiment. In total 2 teams participated, of which 11 (64.7%) were in team A, and 6 (35.3%) were in team B. All the questions were likert-scale questions (1-5), unless otherwise stated in the question.

Question	Answer type
How many years have you worked as a software developer?	Average 9.9 ± 8.1 years
On a scale of 1-10, how competitive are you?	Average 7.2 ± 1.5
On a scale of 1-10, how concerned are you with reducing technical debt and improving the quality of the codebase?	Average 8.2 ± 1.2
Do you agree with the following statement: Code Karma was a positive experience.	Median 4, average 4.0 ± 0.9

Do you agree with the following statement: the resulting karma distribution was surprising to me.	Median 3, average 3.2 ± 0.9
Do you agree with the following statement: I expected to obtain more karma than I received.	Median 3, average 2.6 ± 1.2
Do you agree with the following statement: I think Code Karma has a positive impact on the codebase.	Median 4, average 3.6 ± 1.0
Do you agree with the following statement: Code Karma motivated me to write more tests.	Median 3, average 3.2 ± 1.2
Do you agree with the following statement: Code Karma motivated me to write more docblocks.	Median 3, average 3.0 ± 1.2
Do you agree with the following statement: Code Karma motivated me to do more refactorings.	Median 3, average 3.2 ± 1.3
Do you agree with the following statement: the **ranking distribution** motivated me to obtain more karma.	Median 4, average 3.1 ± 1.1
Do you agree with the following statement: the **team karma goal** motivated me to obtain more karma.	Median 4, average 3.2 ± 1.3
Do you agree with the following statement: the **achievements** motivated me to obtain more karma.	Median 4, average 3.3 ± 1.4
Do you agree with the following statement: the formula for obtaining karma is balanced. The formula is as follows: *(docblocks * 2) + (test_lines_added / 10) + (refactorings * 5)*	Median 3, average 3.2 ± 1.0
Do you agree with the following statement: using nicknames was better than using real names of participants.	Median 4, average 4.1 ± 1.0
Do you agree with the following statement: I would like Code Karma to continue giving the team weekly updates after the end of the experiment.	Median 4, average 4.1 ± 1.0

Table 6.2: Results of questionnaire to evaluate Code Karma.

The overall experience of Code Karma

Overall, the participants seem to agree that Code Karma was a positive experience for them. They mentioned that this gamification could break the inertia of never having written tests

by motivating the developers to write tests. Additionally, developers mentioned that they were motivated to write better code. It gave them positive reinforcement when making a contribution, and reminding them when they made no contribution. The participants also mentioned that the insights were useful and interesting. Some participants mentioned that it is not a true reflection of contribution, since being concerned with code quality might be subordinate to very high priority work and the Code Karma formula might be too simple to accurately reflect contributions. One participant noted that Code Karma is not very useful and might be more of a distraction from getting work done.

The answers to the questions in Table 6.2 indicate that the participants think that Code Karma has a positive impact on the codebase, that Adyen should use Code Karma more and that they would like Code Karma to continue after the experiment. This indicates that Code Karma might be positive for both quality concerns and having fun while doing development work.

Code Karma's impact on motivation

An interesting finding from the results in Table 6.2 is that the participants seemed neutral about the impact of Code Karma on their motivation for writing tests, adding docblocks or doing refactorings. They do, however, seem to be motivated by seeing the ranking distribution, the team karma goal and the achievements in their results. These results might indicate that the participants are not necessarily motivated to perform a particular code quality activity, but that they are motivated by the rewards and the social implications of their contributions as a whole.

Accuracy of the formula

One of the main critiques of Code Karma is the formula. For example, participants wonder whether refactorings always improve code quality, that there are forms of improving code that are not included, and that adding test lines is more important than adding docblocks. Other activities such as removing lines from a deprecated module or creating a test harness for a class that has existed without tests for a long time might be more important than the simple metrics included in the formula.

Closing feedback

The main positive points of Code Karma were the playful way of showing insights into the code quality contributions of the team. The participants also liked the raised awareness of code quality. The weekly summary was fun to receive and acknowledgement of contributions was nice to receive.

The main criticism is the absence of other metrics that might give deeper insight to the actual contribution to the codebase. There could also be an addition of negative karma when people write code that does not have tests associated with it. It was also mentioned that people might be more concerned with gathering karma points than adding value to the business. Additionally, the insights might cause some conflict between developers.

6.4.2 Comparing karma gained to preceding months

To determine if there is an increase in karma due to the experiment, we look at the karma gained in preceding time frames in comparison to the time frame of the experiment. The experiment lasted for 4 weeks starting on Monday the 7th of November. The preceding time frames are 3 time frames of 4 weeks. In Table 6.3 and Table 6.4, these time frames are denoted as the months of the starting dates. The developers included in this analysis participated in the Code Karma experiment, and were active developers at the start of August. The values of Table 6.3 and Table 6.4 were multiplied by a hidden number to obscure the absolute values.

	August	September	October	November
Docblocks	360	330	810	765
Test lines added	50265	73065	58470	82155
Refactorings	285	345	315	645
Karma	7171.5	9691.5	9042	12970.5
Average karma per person	651	881	822	1179.1

Table 6.3: Karma results per month for Team A

	August	September	October	November
Docblocks	720	630	510	840
Test lines added	24765	54195	23580	33330
Refactorings	180	465	210	90
Karma	4816.5	9004.5	4428	5463
Average karma per person	963.3	1800.9	885.6	1092.6

Table 6.4: Karma results per month for Team B

In Team A, there seems to be an increase in karma in November, the month of the experiment. In Team B, this does not seem to be the case due to the high karma gain in September. We can therefore not draw any conclusions as to whether or not Code Karma assisted in motivating the developers to spend more time on writing documentation, writing tests or doing

refactorings.

RQ5: Can we create a gamification system that helps to improve internal software quality?
The results do not prove that the gamification system leads to more activities to improve the internal code quality by refactoring, writing tests or writing docblocks. However, the participants do believe that the gamification system has a positive impact on the codebase. The karma distribution, achievements and team karma goal all seemed to motivate the participants. Therefore, we can say that there is a good chance that this gamification system, if developed further, will have a positive impact on the internal software quality.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we aimed to investigate technical debt at Adyen. We looked at the perception of developers on technical debt or problems in the codebase that hindered them in their development process. We also looked at the refactoring and testing efforts of the developers and the impact of file length and complexity on their refactoring efforts. We looked at the Delta Maintainability Model to give insight in module deterioration or maintainment. Finally, we created a 'Code Karma' gamification system to motivate developers to improve internal quality.

The developers agree that there is technical debt in the system and that there could be improvements in the management of it. They also think that they themselves do very well in terms of managing and avoiding technical debt. There might be a slight underestimation of the developers themselves on how much they add to the technical debt. The main problems with technical debt seem to be testing/testability, documentation, architecture and problems with the code itself (long code, high complexity, etcetera). Another problem that was mentioned was that the codebase is so large that developers do not know a solution already exists in the codebase. The developers say that more time should be reserved for reducing technical debt. More guidelines and more testing could also improve the internal quality. The developers refactored to improve code structure, let code adhere to the Single Responsibility Principle or when the code reviewer suggested them to. Improving the code structure therefore seems to be an important part of the developers' development process.

It does seem like developers that do more refactoring do more testing. When the developers refactor, they also have a higher chance of writing tests. This makes it seem like some developers are more concerned with code structure and tests than others. We did not find a correlation with the author's experience with the codebase and the number of refactorings they do. File length and complexity also do not seem to be a factor that increases the chance of refactoring. There is a weak correlation between lead time and refactoring. This could be due to complexer requirements of a ticket correlating with higher lead time, but also with

one or multiple refactorings being needed to implement the ticket. The number of times a file has changed and the number of authors of a file does not seem to give a higher chance of refactoring.

The unit size of the Delta Maintainability Model [10] plotted over time seems to support the idea of the *Broken Windows Theory*: when there are problems in the codebase, developers are less motivated to write the best code they can. The modules that have long methods will receive more changes that further aggravate this problem. Modules that have mostly short methods, will continue to receive changes that are good for the length of the methods. Plotting the unit size over time could therefore give more insight in what modules are deteriorating into bad code with long methods.

The 'Code Karma' gamification seems to have a positive effect on both the developers and the motivation to improve code quality. The main criticism of the system was that the formula for karma did not reflect true contribution to code quality, since there are many things that are done to improve code quality that were not captured by this formula. Adding more features to this formula and adding a way to toggle these features to reflect what is most important for the software system at that time might be a better approach. Anonymizing the names was perceived as a positive addition, to keep the developers anonymous and avoid negative emotion towards a developers' place in the ranking.

7.2 Answers to the research questions

In this section, we will repeat the research questions and the answers to the research questions as they were formulated in the chapters.

RQ1: What types of technical debt impede the development process?
It seems that tests, testability and documentation could be improved. Developers might keep 'reinventing the wheel' if they do not know that something already exists within the codebase, leading to multiple implementations of the same thing. Solutions to reducing or avoiding technical debt are more testing, better guidelines, more insight into problems, more rewards or recognition for paying off debt and reserving time to pay off technical debt.

RQ2: What is the view of developers on refactoring to reduce or avoid technical debt?
Developers mainly refactor to improve code structure in terms of readability, to remove/avoid code duplication and to make the classes and methods adhere to the Single Responsibility Principle. The developers deem refactoring an important activity, they refactor when they feel that it is needed and they suggest refactoring in code reviews. There is a fear of introducing bugs when refactoring.

RQ3: What is the behavior of developers regarding refactoring and testing efforts?
There is a weak correlation between writing refactorings and editing test files. When a refactoring occurs, there is a higher chance of a test file being changed. When a refactoring is done, it might indicate that the lead time of the corresponding ticket is higher. The file length or the complexity of the methods in the files that are changed during the development process do not seem to correlate with refactorings.

RQ4: Can we use the Delta Maintainability Model to give insights into module health?
The Delta Maintainability Model seems to give an indication of module health when its unit size risk values are added over time. If the aggregate value is low, then this module might risk changes that are bad for unit size, i.e. edits that make methods longer or edits that add long methods. A higher aggregate unit size might indicate better module health with low method lengths. This might in turn mean that future edits will be of low risk to unit size, which means that developers will put in more effort to not let method size get out of hand.

RQ5: Can we create a gamification system that helps to improve internal software quality?
The results do not prove that the gamification system leads to more activities to improve the internal code quality by refactoring, writing tests or writing docblocks. However, the participants do believe that the gamification system has a positive impact on the codebase. The karma distribution, achievements and team karma goal all seemed to motivate the participants. Therefore, we can say that there is a good chance that this gamification system, if developed further, will have a positive impact on the internal software quality.

7.3 Future work

There are two topics investigated in this paper that deserve a continuation in the form of future research.

7.3.1 Delta Maintainability Model

Firstly, the Delta Maintainability Model to quantify the health of modules within a software system. The conclusion that the aggregation of the unit size value of the Delta Maintainability Model showed the health of a module were mainly verified by the experience of developers within Adyen. To see whether this method works consistently in other systems, it should be tested by showing the results in the same way as in Figure 5.2 and the correctness should be quantified using a survey.

7.3.2 Code Karma

Secondly, the Code Karma system could be improved for both developer experience and developer motivation for the right improvement efforts. Since the participants' main criticism was that the karma system did not show a fair representation of contributions due to the lack of features, these features could be tweaked. For example, moving classes might be an important activity if modularization are a priority for the company. Creating a test harness for a class that does not have any tests might be more impactful than just adding test lines to a file. Integration tests might be worth more points than a regular unit test. People might want to gift others karma if these people were helpful for implementing a feature or fixing a bug, since these efforts are invisible to the system. Negative karma could also be introduced. For example, if a person creates a new class without unit tests.

The focus of a development team might change, so these requirements might change as well. A system that has a backoffice where managers can toggle what things are important and what features are not might be a very accessible way of managing technical debt, that is also a fun experience for the developers.

Bibliography

[1] Haldun Akoglu. User's guide to correlation coefficients. *Turkish Journal of Emergency Medicine*, 18(3), 2018. ISSN 24522473. doi:10.1016/j.tjem.2018.08.001.

[2] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *IEEE International Conference on Software Maintenance, ICSM*, 2010. doi:10.1109/ICSM.2010.5609747.

[3] Erik Ammerlaan, Wim Veninga, and Andy Zaidman. Old habits die hard: Why refactoring for understandability does not give immediate benefits. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, 2015. doi:10.1109/SANER.2015.7081865.

[4] Apostolos Ampatzoglou, Angeliki Agathi Tsintzira, Elvira Maria Arvanitou, Alexander Chatzigeorgiou, Ioannis Stamelos, Alexandru Moga, Robert Heb, Oliviu Matei, Nikolaos Tsiridis, and Dionisis Kehagias. Applying the single responsibility principle in industry: Modularity benefits and trade-offs. In *ACM International Conference Proceeding Series*, 2019. doi:10.1145/3319008.3320125.

[5] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. The financial aspect of managing technical debt: A systematic literature review. In *Information and Software Technology*, volume 64, 2015. doi:10.1016/j.infsof.2015.04.001.

[6] Sarah Beecham, Nathan Baddoo, Tracy Hall, Hugh Robinson, and Helen Sharp. Motivation in Software Engineering: A systematic literature review. *Information and Software Technology*, 50(9-10), 2008. ISSN 09505849. doi:10.1016/j.infsof.2007.09.004.

[7] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson Correlation Coefficient. In *Springer Topics in Signal Processing*, volume 2, pages 1–4. 2009. doi:10.1007/978-3-642-00296-0_5. URL `http://link.springer.com/10.1007/978-3-642-00296-0_5`.

[8] Frank Buschmann. To pay or not to pay technical debt. *IEEE Software*, 28(6), 2011. ISSN 07407459. doi:10.1109/MS.2011.150.

[9] Sebastian Deterding. Gamification: Designing for Motivation. *Interactions*, 19(4), 2012. ISSN 10725520. doi:10.1145/2212877.2212883.

[10] Marco Di Biase, Ayushi Rastogi, Magiel Bruntink, and Arie Van Deursen. The delta maintainability model: Measuring maintainability of fine-grained code changes. In *Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019*, 2019. doi:10.1109/TechDebt.2019.00030.

[11] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, Marcos Kalinowski, and Rodrigo Oliveira Spínola. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology*, 121, 2020. ISSN 09505849. doi:10.1016/j.infsof.2020.106270.

[12] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[13] Martin Fowler. Technical debt quadrant, 2009. URL `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`.

[14] Cesar Franca, Fabio Q.B. Da Silva, and Helen Sharp. Motivation and Satisfaction of Software Engineers. *IEEE Transactions on Software Engineering*, 46(2), 2020. ISSN 19393520. doi:10.1109/TSE.2018.2842201.

[15] Félix García, Oscar Pedreira, Mario Piattini, Ana Cerdeira-Pena, and Miguel Penabad. A framework for gamification in software engineering. *Journal of Systems and Software*, 132, 2017. ISSN 01641212. doi:10.1016/j.jss.2017.06.021.

[16] Hadi Ghanbari, Terese Besker, Antonio Martini, and Jan Bosch. Looking for Peace of Mind? Manage Your (Technical) Debt: An Exploratory Field Study. In *International Symposium on Empirical Software Engineering and Measurement*, volume 2017-November, 2017. doi:10.1109/ESEM.2017.53.

[17] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012*, 2012. doi:10.1145/2393596.2393655.

[18] Makrina Viola Kosti, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Georgios Pallas, Ioannis Stamelos, and Lefteris Angelis. Technical debt principal assessment through structural metrics. In *Proceedings - 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017*, 2017. doi:10.1109/SEAA.2017.59.

[19] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6), 2012. ISSN 07407459. doi:10.1109/MS.2012.167.

[20] Oualid Ktata and Ghislain Lévesque. Designing and implementing a measurement program for scrum teams: What do agile developers really need and want? In *ACM International Conference Proceeding Series*, 2010. doi:10.1145/1822327.1822341.

[21] Valentina Lenarduzzi, Terese Besker, Davide Taibi, Antonio Martini, and Francesca Arcelli Fontana. A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software*, 171, 2021. ISSN 01641212. doi:10.1016/j.jss.2020.110827.

[22] Zengyang Li, Nicolas Guelfi, Peng Liang, Paris Avgeriou, and Apostolos Ampatzoglou. An empirical investigation of modularity metrics for indicating architectural technical debt. In *QoSA 2014 - Proceedings of the 10th International ACM SIGSOFT Conference on Quality of Software Architectures (Part of CompArch 2014)*, 2014. doi:10.1145/2602576.2602581.

[23] Zengyang Li, Peng Liang, and Paris Avgeriou. Architectural Debt Management in Value-Oriented Architecting. In *Economics-Driven Software Architecture*. Morgan Kaufmann, 2014. doi:10.1016/B978-0-12-410464-8.00009-X.

[24] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101, 2015. ISSN 01641212. doi:10.1016/j.jss.2014.12.027.

[25] Lech Madeyski and Marian Jureczko. Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal*, 23(3), 2015. ISSN 15731367. doi:10.1007/s11219-014-9241-7.

[26] Antonio Martini, Terese Besker, and Jan Bosch. Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. *Science of Computer Programming*, 163, 2018. ISSN 01676423. doi:10.1016/j.scico.2018.03.007.

[27] Ana Melo, Roberta Fagundes, Valentina Lenarduzzi, and Wylliams Barbosa Santos. Identification and measurement of requirements technical debt in software development: A systematic literature review. *Journal of Systems and Software*, 194:111483, 2022. ISSN 0164-1212. doi:https://doi.org/10.1016/j.jss.2022.111483.

[28] Naouel Moha, Yann Gaël Guéhéneuc, Laurence Duchien, and Anne Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1), 2010. ISSN 00985589. doi:10.1109/TSE.2009.50.

[29] Michael Mohan and Des Greer. A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development*, 6(1), 2018. doi:10.1186/s40411-018-0046-4.

[30] Ally S. Nyamawe, Hui Liu, Nan Niu, Qasim Umer, and Zhendong Niu. Automated recommendation of software refactorings based on feature requests. In *Proceedings of the IEEE International Conference on Requirements Engineering*, volume 2019-September, 2019. doi:10.1109/RE.2019.00029.

[31] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. Why Developers Refactor Source Code: A Mining-based Study. *ACM Transactions on Software Engineering and Methodology*, 29(4), 2020. ISSN 15577392. doi:10.1145/3408302.

[32] Oscar Pedreira, Félix García, Nieves Brisaboa, and Mario Piattini. Gamification in software engineering - A systematic mapping. In *Information and Software Technology*, volume 57, 2015. doi:10.1016/j.infsof.2014.08.007.

[33] Boris Pérez, Camilo Castellanos, Darío Correal, Nicolli Rios, Sávio Freire, Rodrigo Spínola, Carolyn Seaman, and Clemente Izurieta. Technical debt payment and prevention through the lenses of software architects. *Information and Software Technology*, 140, 2021. ISSN 09505849. doi:10.1016/j.infsof.2021.106692.

[34] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings - International Conference on Software Engineering*, 2013. doi:10.1109/ICSE.2013.6606589.

[35] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactoring and software defects. In *Proceedings - International Conference on Software Engineering*, 2008.

[36] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. In *Journal of Software: Evolution and Process*, volume 30, 2018. doi:10.1002/smr.1958.

[37] Giulia Sellitto, Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea Lucia, Fabio Palomba, and Filomena Ferrucci. Toward understanding the impact of refactoring on program comprehension. 12 2021. doi:10.1109/SANER53432.2022.00090.

[38] Mahnoosh Shahidi, Mehrdad Ashtiani, and Morteza Zakeri-Nasrabadi. An automated extract method refactoring approach to correct the long method code smell. *Journal of Systems and Software*, 187, 2022. ISSN 01641212. doi:10.1016/j.jss.2022.111221.

[39] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we Refactor? Confessions of Github contributors. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 13-18-November-2016, 2016. doi:10.1145/2950290.2950305.

[40] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018. doi:10.1145/3236024.3264598.

[41] Gail M. Sullivan and Anthony R. Artino. Analyzing and Interpreting Data From Likert-Type Scales. *Journal of Graduate Medical Education*, 5(4), 2013. ISSN 1949-8349. doi:10.4300/jgme-5-4-18.

[42] Edith Tom, Aybüke Aurum, and Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6), 2013. ISSN 01641212. doi:10.1016/j.jss.2012.12.052.

[43] Adam Tornhill and Markus Borg. Code red: The business impact of code quality - a quantitative study of 39 proprietary production codebases. In *Proceedings of the International Conference on Technical Debt*, TechDebt '22, page 11–20, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393041. doi:10.1145/3524843.3528091. URL https://doi.org/10.1145/3524843.3528091.

[44] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, 2022. doi:10.1109/TSE.2020.3007722.

[45] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

[46] Peter Weißgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In *Proceedings - International Conference on Software Engineering*, 2006. doi:10.1145/1137983.1138011.

[47] Laerte Xavier, Joao Eduardo Montandon, and Marco Tulio O Valente. Comments or issues: Where to document technical debt? *IEEE Software*, pages 0–0, 2022. doi:10.1109/MS.2022.3170825.

[48] Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander. How do software development teams manage technical debt? – An empirical study. *Journal of Systems and Software*, 120, 2016. ISSN 01641212. doi:10.1016/j.jss.2016.05.018.

[49] Oleksandr Zaitsev, Stephane Ducasse, Alexandre Bergel, and Mathieu Eveillard. Suggesting descriptive method names: An exploratory study of two machine learning approaches. In *Communications in Computer and Information Science*, volume 1266 CCIS, 2020. doi:10.1007/978-3-030-58793-2_8.