MASTER'S THESIS 2023

# Evaluating Similarity-Based Refactoring Recommendations

Emma Ericsson


ISSN 1650-2884
LU-CS-EX: 2023-46

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-46

# Evaluating Similarity-Based Refactoring Recommendations

## Utvärdering av likhetsbaserade refaktoriseringsrekommendationer

**Emma Ericsson**

# Evaluating Similarity-Based Refactoring Recommendations

Emma Ericsson
ine15eer@student.lu.se

November 6, 2023

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Markus Borg, markus.borg@cs.lth.se
Emil Aasa, emil.aasa@codescene.com

Examiner: Emma Söderberg, emma.soderberg@cs.lth.se

**Abstract**

Within recommendation systems, there is a well-known problem of not having the required information for making recommendations, *the cold start problem*. CodeScene's refactoring recommendations are based on earlier refactoring within a project and therefore, this problem occurs when almost no refactoring has been carried out in the project. Our goal is to evaluate how existing similarity measures can be used to identify similar source code from open-source software projects that have been improved through refactoring operations. Presenting such examples to developers could prove inspirational to CodeScene, and thus offer a way forward for CodeScene's refactoring recommendation feature. 1,438 refactored methods were collected with CodeScene from top-starred Java projects on GitHub. Four approaches to measuring source code similarity from previous research were evaluated on this dataset. We found Levenshtein distance and code2vec representation of code together with cosine distance to be the most promising. Furthermore, we designed a user study with 10 realistic refactoring tasks accompanied by similar refactorings from other projects in the dataset. Based on interviews with five senior developers, we conclude that there are indications that cross-project similarity-based refactoring recommendations could be useful. Finally, developers find that good recommendations shall be short, concise and self-contained.

**Keywords**: refactoring, refactoring recommendations, similarity, code2vec, Levenshtein distance

# Acknowledgements

I would like to thank:

**Markus Borg**, for being an incredible supervisor. I am grateful for your generous support and also for challenging me. I appreciate that you have taken the time to discuss different aspects of the thesis and given me recurring feedback throughout the whole process.

**Emil Aasa**, for your ideas and data collection.

**Tjaša Heričko**, for inspiring the research.

**CodeScene**, for the opportunity to make this thesis in collaboration with you.

I would also like to thank the CodeScene developers who helped us in the evaluation.

# Contents

# Chapter 1
# Introduction

In this chapter, we will provide context for the thesis. We will firstly present some background (Section 1.1) and then go through the current problem (Section 1.2) and the goal of the thesis, including our research questions (Section 1.3). In the end of the chapter we will go through our contributions (Section 1.4) and the outline (Section 1.5) for the report.

## 1.1 Background

This master's thesis is carried out in collaboration with the company CodeScene. The CodeScene tool is a software engineering intelligence tool that identifies and prioritises technical debt for software projects to improve their efficiency [7]. It highlights a behavioural dimension of the software project by analysing the developers' interaction with the codebase. In the tool, four factors are used to visualise and understand the source code. Those are code health, knowledge distribution, team-code alignment and delivery [10].

Code health is a central concept in the tool. It is a metric that is aggregated based on more than 25 factors that are known to correspond with an increased risk for bugs and higher maintenance costs [9]. CodeScene measures the code health of the codebase and displays it through different perspectives to show the status of the codebase. An example can be seen in Figure 1.1.

In the tool different views can be entered to look into different aspects of the software. For example, in the hotspots tab the files that are changed most often can be seen. In the code health tab the files are classified according to their code health. In the refactoring tab a combination of the hotspots tab and code health tab is showed meaning that the files are classified according to how often they are changed and how complex they are. This view can help out with the prioritisation of the improvement work. The files that are most critical (illustrated in red) can be further examined by identifying issues on the function level. The functions with issues are suitable for refactoring.

Refactoring implies changing the code to improve its inner structure without changing its

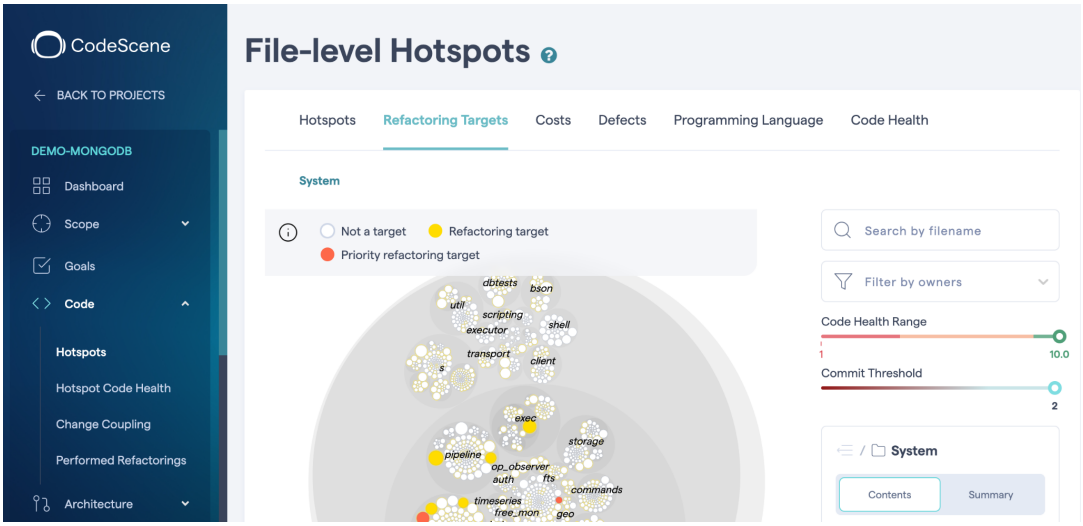



**Figure 1.1:** A snapshot of the tool CodeScene.

external behaviour [14]. By refactoring the code it can be easier to understand and maintain the software and therefore decrease the maintenance costs [25]. More information about refactoring can be found in Section 2.1.

CodeScene carries out automatic code reviews on the code to detect code smells (potential problems in the code, see Section 2.2). When a code smell is detected a refactoring recommendation will be presented. Usually a static pre-determined example will be shown. Though if the project has a lot of history with earlier refactorings the example could instead be picked from there. That is, previous changes in the project that have resulted in improved code health for the same code smell [8]. A supervised machine learning model predicts whether the previous refactoring is likely to be helpful or not.

## 1.2 Problem Description

The refactoring recommendations are based on earlier refactorings performed within the project and therefore the recommendation pool is limited. This is a problem for projects where almost no refactorings have been carried out. This kind of problem has existed for a long time in recommendation systems research. It is called *the cold start problem* [15] and is basically the problem of not having the required information for making recommendations. Gope, J. and Jain, S.K. [15] have examined different solutions to gather the missing information and found that the solutions could be categorised as explicit and implicit solutions. The explicit solutions are solutions that interact with the user to collect the desired information, which could be done by letting the users answer questions in a questionnaire or rate items. The implicit solutions are on the other hand trying to interact as little as possible with the user. Here the information is collected through already existing information as demographics and information from social media.

We will go for an implicit solution which is illustrated in Figure 1.2. Our solution is to increase the pool of refactoring recommendations by finding similar refactored methods in other projects. To find similar methods we will use different similarity measures.

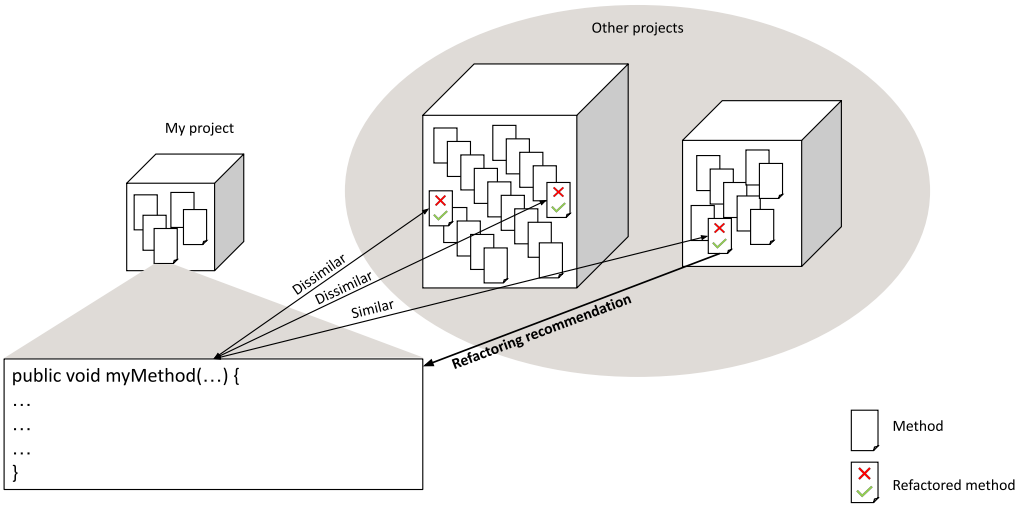



**Figure 1.2:** An illustration of the solution where similar refactored methods in other projects are used as refactoring recommendations.

The solution is further illustrated in Figure 1.3. Our idea is to collect a set of refactored methods with both the source code from before and after the refactoring. We call the methods from before refactoring *pre-methods* and after refactoring *post-methods*. A method could be picked from the set of pre-methods and be compared to the other methods in the set. Then the post-method to the most similar method would serve as a refactoring recommendation.

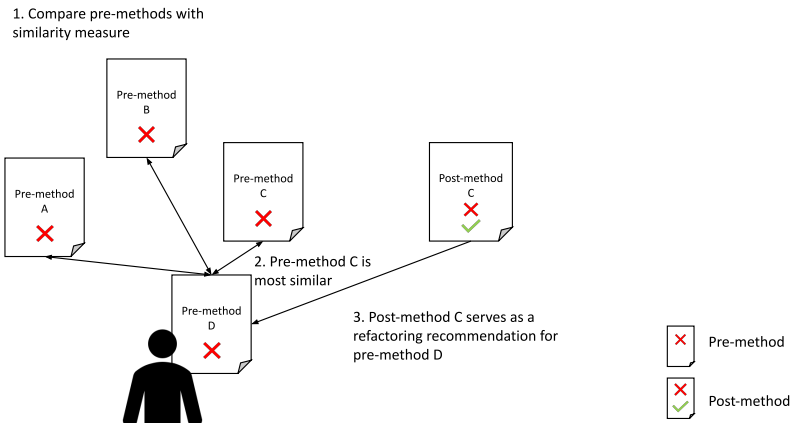



**Figure 1.3:** An illustration of how pre-methods are compared to use the most similar pre-method's post-method as a refactoring recommendation.

## 1.3 Research Question

The goal of this master's thesis is to evaluate how existing similarity measures can be used to identify similar methods from open-source software projects for refactoring recommendations. The goal is also to propose a way forward for CodeScene's refactoring recommendation feature. The following research questions form the basis of the work:

- **RQ1** How well do similarity measures work in the context of CodeScene's refactoring recommendations?
- **RQ2** How do senior developers perceive similarity-based refactoring recommendations?

## 1.4 Contribution

This thesis contributes with an analysis of existing similarity measures and an evaluation of two of them in the context of refactoring recommendations. It also contributes with knowledge about how senior developers perceive similarity-based refactoring recommendations.

The author has carried out the work with guidance from the supervisors. The industrial supervisor helped out with collecting the data with CodeScene. The academic supervisor has supported with feedback throughout the whole process and was part of creating codes and themes in the thematic analysis.

## 1.5 Outline

In Chapter 2, we address the theory needed to understand the report. This covers the three concepts refactoring, code smells and recommendation systems, and the similarity measures that will be used in the thesis. Work related to the thesis on similarity and refactoring recommendations we present in Chapter 3. In Chapter 4 we describe the methodology of the thesis. This includes a pilot experiment, data collection, data processing, similarity calculations and user interviews. The result is presented in Chapter 5 and discussed in Chapter 6 where limitations of the thesis are also described. Chapter 7 contains conclusions from the thesis and suggestions for further work.

# Chapter 2
# Theory

In this chapter, we will go through important concepts to understand the implementation of the thesis. First, we will dig deeper into refactoring (Section 2.1), code smells (Section 2.2) and recommendation systems (Section 2.3). Then we will go through the different similarity measures (Section 2.4) that are used in the thesis.

## 2.1 Refactoring

As the software grows due to the implementation of new features and modifications, it becomes more complex and usually departs from the original design [25]. Therefore the majority of the development cost is spent on the maintenance of the code. To prevent this, the complexity of the code needs to be reduced by restructuring or refactoring the code. According to Fowler refactoring is: "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [14]. This is done by reorganising classes, variables and methods to make forthcoming adaptations and extensions easier [28]. If the refactoring is performed well it improves the quality of the code and makes it easier to understand, change, maintain and evolve.

## 2.2 Code Smells

Code smells are different kinds of design problems that could be recurring in the code [28]. Those code smells often make the software harder to understand, maintain and evolve [33]. The code smells are often introduced due to bad design and implementation practices and could be introduced during either the initial design or the development process [28]. Apart from bad design decisions, they could also be introduced due to ignorance or time pressure. Refactoring is an efficient way of fixing code smells.

In literature, a lot of different code smells can be found but in this thesis, we are focusing on five function-level code smells that CodeScene detects. Those are *bumpy road ahead*, *complex method*, *deep nested complexity*, *excess number of functions* and *large method*. A description of each of them can be found in Table 2.1.

**Table 2.1:** The five CodeScene code smells investigated in the thesis.

| Code Smell | Description |
|---|---|
| Bumpy Road Ahead | The method has several chunks of nested conditional logic [6]. |
| Complex Method | The method consists of many conditional statements as if, while and for. This is measured with cyclomatic complexity [24] with a threshold of 9 for Java. |
| Deep, Nested Complexity | The method has if statements inside loops and/or other if statements. |
| Excess Number of Function Arguments | This indicates either that the function has too many responsibilities (low cohesion) or that it misses an abstraction that encapsulates the arguments. [11] The threshold for Java is four function arguments. |
| Large Method | The method contains many lines of code. The threshold for Java is 70 lines of code. |

## 2.3 Recommendation Systems

A recommendation system is a system that through recommendations assists the user in their decision-making when being faced with a lot of information [35]. By knowing the preferences of the users the system can navigate in the large information space and recommend items of interest. The system can offer novelty, surprise and relevance.

In software engineering this could be about a developer trying to find a desired class from hundreds of libraries. Recommendation systems for software engineering are used for several activities as for reuse of code, writing effective bug reports and, as in our case, refactoring. To help out with the refactoring task, a recommendation system can provide a user with examples of code improvements done by other developers, potentially in other projects, to inspire similar actions for the task at hand.

## 2.4 Similarity Measures

In this thesis four different measures are used to identify how similar methods are. Those measures are code2vec together with cosine distance (Section 2.4.1), JPlag (Section 2.4.2), Jaccard distance (Section 2.4.3) and Levenshtein distance (Section 2.4.4).

### 2.4.1 Code2vec

Code2vec is a neural model that represents snippets of code as vectors [2]. One snippet of code is represented as one fixed-length vector with 384 dimensions. The vectors demonstrate the semantic characteristics of the code meaning that semantically similar code snippets are closer to each other in vector space. This is demonstrated in the paper by Alon et al. [2] by the model predicting names for methods. The code of a method is used as input and then code2vec outputs predicted names. The task of predicting names was evaluated by training the model on 12 million methods. When Alon et al. compared their approach to previous approaches using the same dataset, they saw an improvement of 75 percent.

To get the code vector, the code snippet is first parsed into an abstract syntax tree (AST). Then the paths between the leaves in the AST are collected. Every path is represented as a vector, a *path vector*. The tokens where the path begins and ends are represented as two different *token vectors*. The path vector and the two token vectors are then combined into a vector, one for every path vector in the AST. With some further processing these become vectors called *path-contexts*. In the paper Alon et al. give an example of how the path-context could look like for the expression `x = 7`. The arrows indicate up and down links in the tree:

⟨x, (NameExpr ↑ AssignExpr ↓ IntegerLiteralExpr), 7⟩

To get a vector for the whole code snippet, an attention approach is used, meaning a weighted average of the path-contexts. This final vector is the *code vector*.

#### Cosine Distance

To measure how close different code vectors are to each other in the vector space we used cosine similarity. Cosine similarity is the cosine of the angle between two vectors. The vectors must be the same size and can be calculated according to the following formula:

$$cosine_s(A, B) = \frac{A \cdot B}{||A||\,||B||} \tag{2.1}$$

If the vectors are identical the angle will be 0 degrees and therefore the cosine similarity will be 1. If the angle is 90 degrees the cosine similarity will be 0 and -1 for an angle of 180 degrees.

To get the cosine distance the cosine similarity is subtracted from one as in the following formula:

$$cosine_d(A, B) = 1 - cosine_s(A, B) \tag{2.2}$$

In the rest of this thesis, we will treat code2vec and cosine distance as one measure and thereby refer to it as code2vec + cosine distance (or c2v+cos).

### 2.4.2 JPlag

JPlag [31] is a plagiarism tool that compares a set of programs through pairwise comparison. It has been used for finding plagiarism within student groups.

When running JPlag, the input is a set of programs. Every program is parsed and converted into token strings. JPlag tries to capture the program's structure and ignores elements

as comments, whitespace and identifiers as well as putting semantic information into tokens where it is possible. For every program pair the token strings are pairwise compared. This is done as *Greedy String Tiling* [44] which means that JPlag tries to cover a token string with substrings from the other token string. The similarity value is how much of the token strings that can be covered, expressed in percentage.

### 2.4.3 Jaccard Distance

Jaccard index was introduced by Paul Jaccard under the name *coefficient de communauté* [29] in 1901. It is defined as the intersection of two sets divided by the union of the sets [43] as in the following formula:

$$Jaccard_i(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.3}$$

Jaccard index has a value between 1 and 0 where the sets are very similar if the index is close to 1 and very dissimilar if the index is close to 0.

For strings, n-grams can be used to make up the sets. For example the string `similarity` with 1-gram is {`s`,`i`,`m`,`i`,`l`,`a`,`r`,`i`,`t`,`y`}.

While Jaccard index shows similarity Jaccard distance shows dissimilarity [43]. To get the Jaccard distance the Jaccard index is subtracted from one as for the cosine distance:

$$Jaccard_d(A, B) = 1 - Jaccard_i(A, B) \tag{2.4}$$

### 2.4.4 Levenshtein Distance

Levenshtein distance calculates the distance between two strings by how many substitutions, deletions and insertions that have to be done to change one string to the other [41]. An example could be illustrated with the strings `dissimilar` and `similar`. The first three letters have to be deleted to change `dissimilar` into `similar` and therefore the Levenshtein distance is 3.

# Chapter 3
# Related Work

In this chapter, we will review work related to the thesis to broaden the context. First, we will give an overview of existing similarity approaches for comparing code (Section 3.1). Then, we will also highlight studies that are related to refactoring recommendations (Section 3.2).

## 3.1 Similarity

There are a lot of different ways to measure similarity in code. Ragkhitwetsagul et al. [32] have in their study compared 30 similarity tools and techniques. They have classified different similarity techniques according to what approach they are based on. These approaches are: metric-based, text-based, token-based, tree-based and graph-based.

### 3.1.1 Metric-Based Approaches

The metric-based approaches compare source code using different metrics [20] [47]. Ottenstein [27] used this approach in a tool for plagiarism detection which was based on Halstead complexity measures [16]. Those measures are the number of unique operators, the number of unique operands, the total number of occurrences of operators and the total number of occurrences of operands. Kasper and Godfrey [19] compared a metric-based method to a parameterized string matching method for code cloning and concluded that the clones from the parameterized string matching method were more useful. The accuracy of this approach is low since too much information about the structure disappears [47].

### 3.1.2 Text-Based Approaches

The text-based approaches are based on comparison of sequences of strings [32]. These approaches are able to find identical copies of source code but do not effectively detect similar

source code with semantic and syntactic modifications. An example of a tool that uses text comparison is NiCad [36] as well as the measure Levenshtein distance [41].

### 3.1.3 Token-Based Approaches

In token-based approaches the source code is transformed into tokens before comparison [26]. Ragkhitwetsagul et al. [32] give an example of this where the tokens are on word level, meaning that every word is replaced by a token, `W`. In this case both the expression `int x = 0;` and the expression `String s = "Similarity";` will be represented as `W W = W`. By defining the tokens differently a different abstractions level will be captured. CCFinder [18] and SourcererCC [37] are examples of clone detection tools that use token-based approaches. Both of them detect clones with different identifier names and have been shown to be efficient on million lines of code. Here we also find JPlag [17] as well as Jaccard distance [29].

### 3.1.4 Tree-Based Approaches

Tree-based approaches transform the source code into a tree structure and then sub-trees are compared [32]. Usually Abstract Syntax Trees (ASTs) are used. These approaches focus on structural similarity and can therefore avoid lexical differences and formatting. Though, they have high computational complexity. An example of a clone detection tool that uses ASTs is CloneDR [3]. This tool tackles the problem with high computational complexity by categorising sub-trees with hash values.

### 3.1.5 Graph-Based Approaches

Graph-based approaches transform the source code into graphs before comparison [47]. Commonly used graphs are program dependency graphs (PDG) and control flow graphs (CFG). Apart from structure, graph-based approaches also capture the semantics [32]. Those approaches do not work well on large systems [4] and as for the tree-based approaches the graph-based approaches have high computational complexity [32]. Examples of tools with graph-based approaches are one invented by Krinke [22] and another invented by Komondoor and Horwitz [21].

### 3.1.6 Other Approaches

Karakatič et al. [20] use Hausdorff distance on code2vec embeddings to estimate semantic similarity between Java-based libraries. Their study showed that the semantic similarity could effectively be captured by this approach. This study was a big inspirational source for the thesis.

## 3.2 Refactoring Recommendations

In this section, we will highlight four studies that are related to refactoring recommendations. One of them is using code2vec and one Jaccard distance, which are measures that we also will

use.

Ouni et al. [28] introduce an automated search-based refactoring recommendation tool called MORE. The tool is based on the non-dominated sorting genetic algorithm NSGA-II and gives recommendations from the perspective of improving software quality, fixing code smells and introducing design patterns.

Alizadeh et al. [1] introduce an interactive refactoring recommendation tool that dynamically adapts and gives recommendations based on developers' feedback. This tool also uses NSGA-II in its recommendations. NSAG-II is used to find good refactoring solutions that improve the code quality while still trying to stay close to the initial design. From those solutions interesting similarities are collected, as frequently occurring refactorings. Based on these, refactorings are proposed, in ranked order, to the developer who can accept, decline or modify the refactoring. The feedback from the developer is used to change the current ranking and for future refactoring recommendations. When evaluating the tool on eight open-source projects and two industrial projects, the tool performed significantly better than four search-based refactoring approaches and one tool that is not based on heuristic search.

Tsantalis and Chatzigeorgiou [40] carried out a study about Move Method refactoring. Move method refactoring means moving the method from the current class to the class where it is commonly used [23]. This is one of the most popular refactoring types. In the study Tsantalis and Chatzigeorgiou are using Jaccard distance to decide the distance between a method and a class. They concluded that using Jaccard distance to measure how well methods are placed in classes can be helpful as a basis for ranking refactoring recommendations for Move Method refactoring. This is one of the studies that the Eclipse plug-in tool Jdeodorant [39] is based on.

Kurbatova et al. [23] use code2vec to recommend Move Method refactorings. Potential methods and classes were represented as vectors with code2vec and then the current method were concatenated with one class vector at a time. The concatenated vector were fed through a probability classifier to decide whether the method should be moved to the class. Kurbatova et al. evaluated their approach on one dataset of well-known open-source projects where the methods were moved manually and one dataset of projects with automatically injected code smell instances. This showed that their approach recommended accurate refactorings and outperformed state-of-the-art tools as JDeodorant and JMove.

# Chapter 4
# Method

In this chapter, we will go through the methodology for solving the defined problem in Section 1.2. An overview of this phase can be seen in Figure 4.1. Initially, we examined manually created example methods to figure out how the different similarity measures performed on those (Section 4.1). To examine how the different similarity measures performed on real methods, we collected data from open source Java projects (Section 4.2). From this data, we retrieved methods (Section 4.3) which were used for making similarity calculations (Section 4.4). Some of those methods were also used for evaluating how well the similarity measures work for refactoring recommendations and to explore how developers perceive similarity-based refactoring recommendations. This was done in user interviews (Section 4.5).

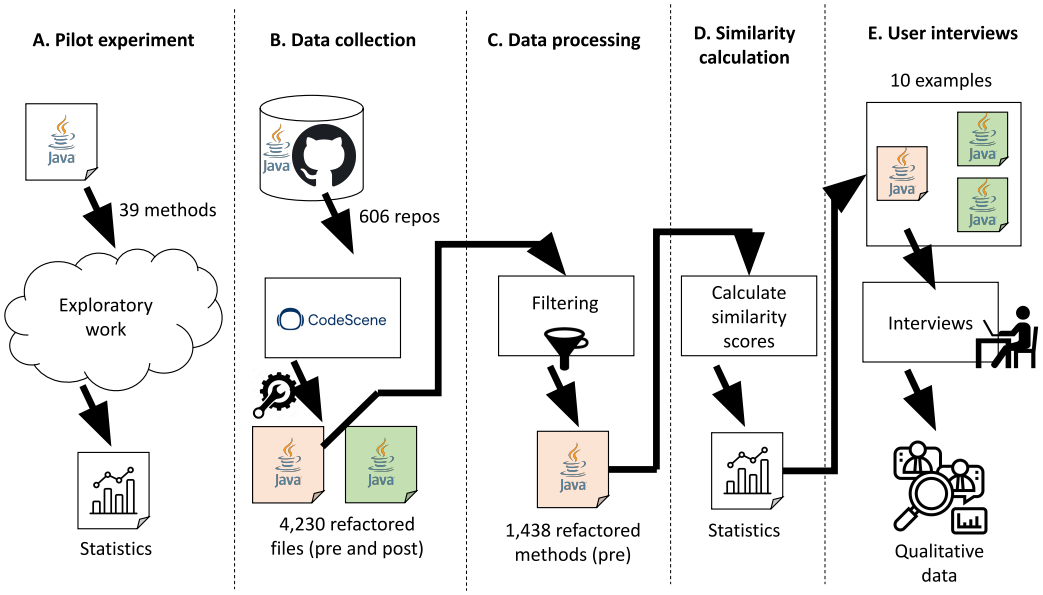



**Figure 4.1:** Overview of the method process

## 4.1 Pilot Experiment

As a first step, we used the similarity measures on our own manually created example methods to examine how the different similarity measures performed and if they classified the same methods as similar.

To do this we created 39 small methods which can be seen in Appendix A. They were all derived from a method `addition` which simply calculates the addition of two input numbers. To this method, different things were added as a for loop, if statement or a `try...catch` block. In some methods the method or parameters were renamed. There were also methods that had multiple for loops or if statements or a combination. The content in for loops and if statements could also be different and the order of the for loops and if statements.

We ran the test in two versions. In Test 1, we used the methods that only were changed in one way meaning, adding a for loop or adding an if statement etc. Those are referred to as M1-M16 in the appendix. All methods were compared to each other. In Test 2, methods with several changes were added as several for loops, one for loop and one if statement etc. Those are referred to as M1-M39 which means that all methods were included in Test 2.

To allow comparisons, we encoded our intuitive understanding of source code similarity. This can be seen in Figure 4.2. We posit that something is almost similar if only something small is changed, such as a renaming of the method or a parameter. Then it is a little bit more dissimilar if something is changed in for example a for loop. Even more so if more identical for loops are added. One step further away if something in turn is changed within those added for loops. If we change a for loop to something similar, for example a while loop, we take another step away from similarity. If we change the for loop to something less similar, such as a `try...catch` block, we would take another step away from similarity. The goal of the pilot experiment was to investigate to what extent the selected similarity measures reflect our similarity intuition.

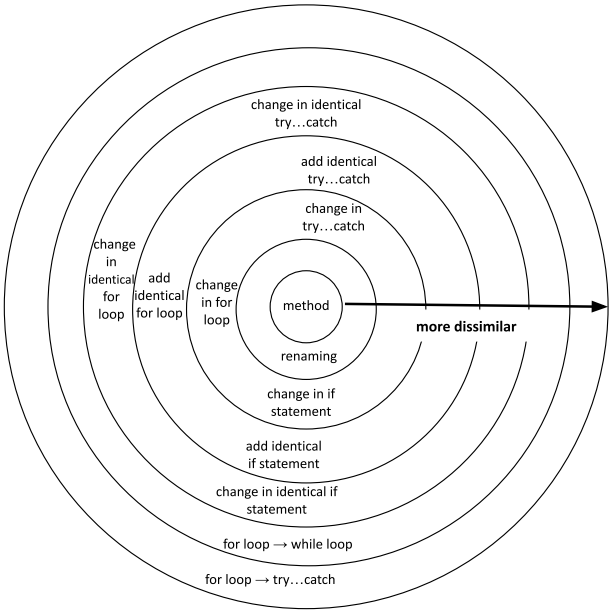



**Figure 4.2:** The intuitive understanding.

### 4.1.1 Similarity Measures

We compared the methods using different similarity measures. Those were code2vec + cosine distance, the similarity score provided by JPlag, Jaccard distance and Levenshtein distance. More information about those can be found in Section 2.4.

We selected measures that differed from each other in various ways to see results from different interpretations of similarity. Levenshtein distance is text-based and code2vec is a neural model which captures semantic similarity. Jaccard distance and JPlag are both token-based tools. JPlag differed in the way that it is a plagiarism tool and thus refactoring recommendations is not its usual area of use.

#### Code2vec + Cosine Distance

For code2vec + cosine distance, the Java methods were transformed into vectors by code2vec and then compared with cosine distance. We used the pre-trained model that is referred to in code2vec's readme on GitHub [38]. This has been trained for 8 epochs on a pre-processed dataset with around 14 million Java methods. Before using code2vec, we made some changes. Firstly, we changed code2vec so we got the code vectors as output instead of the name predicting vectors. Then we changed how the input was processed. Instead of changing an input file we made code2vec parse all the files in a folder structure. We also changed the output format. The code vectors were saved to a Pandas dataframe, which was saved as a csv file.

We examined the similarity of the vectors by using the cosine distance between the vectors. The dataframes from the csv files were imported and converted to numpy arrays. Then we calculated the cosine distance between every vector. For this we used SciPy's cosine distance.

#### JPlag

For JPlag we cloned the JPlag repository [17] in Google Colab and used Conda to use Java version 17. The input format for JPlag is a class, therefore we encapsulated every method in `public class Example`. We put all the files in a folder and compared them to each other. When running JPlag we used the default sensitivity.

#### Levenshtein Distance and Jaccard Distance

For Jaccard distance and Levenshtein distance we used multidistances.jl [34] We ran the program with the default settings. The runs were made locally on a computer through Docker. Here we also compared every file to each other.

For every similarity measure, we calculated the distances between every Java method and every other Java method. From this, we made one distribution for every similarity measure. For every measure apart from JPlag, we made a distance matrix which includes the distances between every method and every other method, including the method itself. This was used for a mantel test to see the correlations between measures. We used `scikit-bio`'s mantel test with Spearman's correlation. The result can be seen in Section 5.1.

## 4.2 Data Collection

To examine how the different similarity measures performed on real Java methods, we collected a dataset of Java refactoring commits that fix code smells. An illustration of the data collection process can be seen in Figure 4.3. The refactored methods were collected from the 606 Java repositories with the most stars on GitHub. First, the repositories were identified. This was done by a curl query that collected the repositories from GitHub with the language Java, in descending order, with the most stars on May 31, 2023.

Then, CodeScene was run on the repositories. When doing this we used several criteria based on previous experience within the company. The refactoring should add more than zero lines of code and less than 100. The refactoring should also delete fewer lines of code than 100. Further, the refactoring should result in a code health improvement that is more than 0.1.

From the repositories, we collected both pre-methods and post-methods. We also collected the names of the methods, what kind of code smell was fixed for every method, what repository it belonged to, how many lines had been added and deleted in the different refactorings, the date of the commits and an ID for the refactoring. In total, we collected 4,230 Java refactorings.

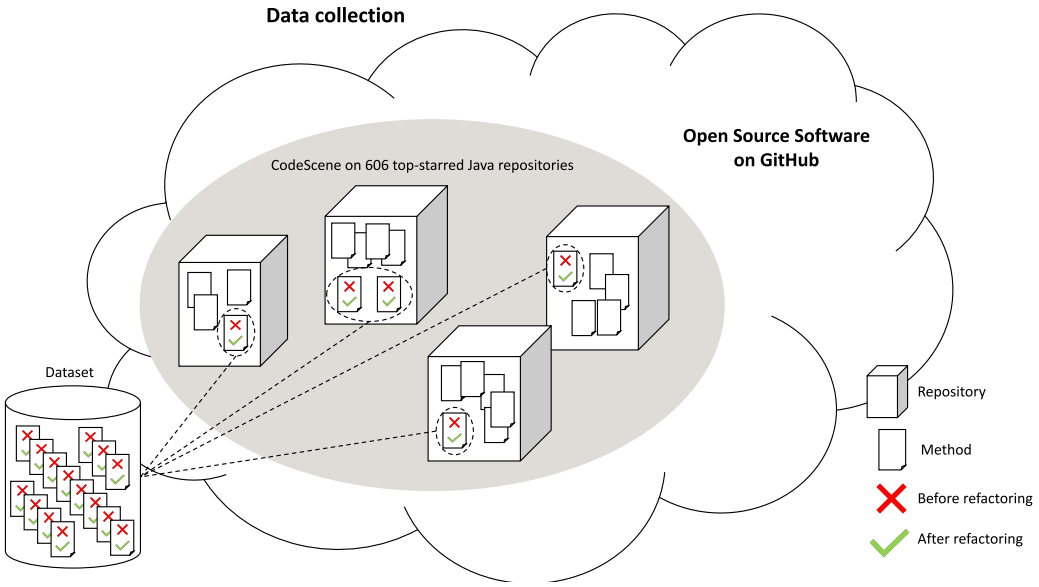



**Figure 4.3:** Data collection process

## 4.3 Processing of Data

To retrieve the methods we went through different stages which can be seen in Figure 4.4.

The processing of the data and analysis were carried out in Google Colab. First, we discarded refactorings that did not belong to the code smells in Table 2.1. Then, we removed all but the last refactoring of every method. We decided to never include a method more than once, i.e., methods that were refactored several times only appear once in the dataset. This

decision was made to avoid including different versions of the same method multiple times in the dataset – we wanted to avoid this bias in the dataset. After this step, we had 2,475 methods left.

The whole file for the methods was collected so we had to separate the desired methods from the files. The files with the methods were decoded from base64. Then we identified the desired method in every file with javalang [5] and saved the code. Javalang is a lexer and a parser for Java source code. Every method was saved to a Java file, named by the ID of the method. Methods that could not be identified were discarded. After this stage we had 1,861 methods.

Methods that could not be converted to code2vec vectors were discarded. Files that could not be converted were rejected by code2vec due to reasons such as invalid characters and lexical errors. This could for example be that a method was not properly parsed or begun with an asterisk. After this, we had 1,660 methods. Some methods were divided into several vectors by code2vec. Since we couldn't associate a refactoring in a method with a specific vector when multiple vectors were present, we chose to exclude methods with multiple vectors. After that we had 1,466 methods. For some projects there were identical methods or code clones in different places of the project. To have a set of unique methods we removed those which led to a set of 1,438 unique methods.

The methods were categorised according to what code smell they contained. This led to 438 methods with *bumpy road ahead*, 587 with *complex method*, 359 with *deep, nested complexity*, 203 with *excess number of function arguments* and 44 with *large method*. Since a method can contain several code smells there is some overlap between the code smells. The overlap can be seen in the GitHub repository for the thesis [12]. The methods come from 163 projects.

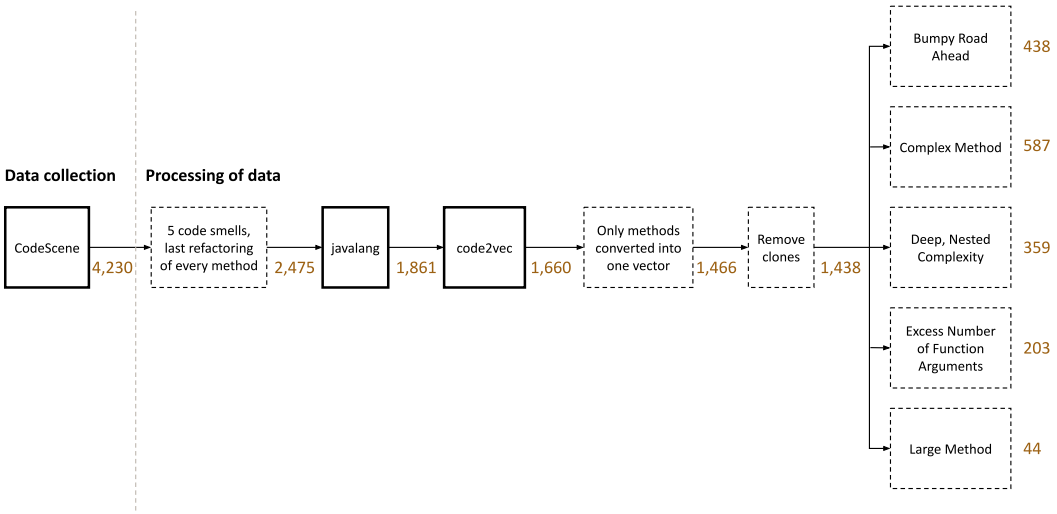



**Figure 4.4:** Number of methods at every stage.

## 4.4 Similarity Calculations

In this phase, we made calculations on our collected pre-methods. As for the manually created methods in the pilot experiment, we investigated the distributions for every similarity measure and analysed how the different similarity measures correlate. At this step, we used the measures code2vec + cosine distance, Jaccard distance and Levenshtein distance. To analyse the correlations, we used SciPy's `stats.skewtest`. The result can be seen in Section 5.2.

## 4.5 User Interviews

In this phase, we wanted to evaluate the similarity measures and learn how senior developers perceive refactoring recommendations selected based on different similarity measures. The evaluation consisted of five individual interviews with developers from CodeScene where the interviewees were told to express their thoughts about different refactoring recommendations. The interviews were conducted remotely and relied on screen sharing.

For the task, the interviewees got an example of some code that they were going to get familiar with. This was a pre-method. After that, they got two Examples of Refactoring Recommendations (ExRR). One of the ExRRs was the refactoring of a pre-method that was close to the example according to code2vec + cosine distance and the other ExRR was the refactoring of a pre-method that was close to the example according to Levenshtein distance. The ExRRs were designed as git commits where lines that had been removed were marked in red and lines that had been added in green. Then we asked which one of the ExRRs was the most helpful as a refactoring recommendation. In total, ten pre-method examples were displayed (two from each code smell). The experimental examples were prepared in a slide deck using standard syntax highlighting of the source code. The two ExRRs were presented in random order.

When carrying out the interviews, we followed a manuscript that can be seen in Appendix B. We started by giving the interviewees a brief background of the thesis project, i.e., we explained that we studied refactoring recommendations but did not mention code similarity measures. We also asked for their consent to record the interview and use their answers for the thesis. Then we asked about their Java experience and their high-level understanding of refactoring. Before presenting any examples, we told the interviewees to have the perspective that it is Friday and a clean-up day at the office and that they have the whole day to improve some code that they have not had time to refactor before. At the end of the interview, after discussing the 10 examples, we asked what they thought was a good refactoring recommendation and if they wanted to add something that came up during the interview.

To validate our design, we conducted a pilot experiment to check whether the 10 examples were a suitable amount of tasks for an interview session. The thesis student and the academic supervisor independently conducted the pilot experiment using their own laptops. We concluded that ten examples was an appropriate number, more than ten methods would be hard to assess in one single interview session.

The interviews followed a *Think Aloud Protocol* which means that the interviewees were asked to share their thoughts while carrying out the tasks [45]. The interviews were conducted remotely via Google Meet. Three of the interviews were held in Swedish and two in English. All interviews were recorded and those conducted in English were also automatically

transcribed by Google's provided service.

Information about the interviewees can be found in Table 4.1. They had between 26 and 6 years of professional development experience. Four of the developers were men and one was female. For confidentiality reasons, we do not share more information.

**Table 4.1:** Overview of the interviewees.

| Interviewee | Role | Years of experience | Java Experience |
|---|---|---|---|
| P1 | Senior developer | 15 | Substantial |
| P2 | Tech lead | 19 | Limited |
| P3 | Tech lead | 26 | Some |
| P4 | Senior Developer | 15 | Substantial |
| P5 | Senior Developer | 6 | Limited |

After the interviews, the Swedish interviews were transcribed and the English transcriptions from Google were corrected. The transcriptions were sent to the interviewees for validation. Each interviewee's answer was summarised for every example and ExRR and put in a mind map. General things that the interviewees pointed out were also transferred to the mind map. From the mindmap, we conducted a thematic analysis [13]. We analysed the statements and started creating low-level codes that emerged in the process. We refined the set of codes in a coding workshop involving the thesis student and the academic supervisor. Once we had coded all transcripts once, we revisited them a second time to ensure that all statements were consistently coded using the final version of the codes. The codes and their descriptions can be seen in Table 4.2. After the coding, we created high-level themes. Those are presented in Section 6.2.

### 4.5.1 Selecting Examples

In this section, we describe how we selected the ten examples and their ExRRs.

When selecting the ten examples, we selected pre-methods carefully to mitigate confounding factors related to code comprehensibility [46]. We started by looking at the length of the methods. We selected methods between 30 and 35 lines of code except for the two examples for large method (which inevitably had to be longer). We wanted to have a fixed length for the methods to eliminate its impact while comparing the examples. Further, methods between 30 and 35 lines easily fit on a single screen which eliminates the need for scrolling.

Before selecting ExRRs we had to create their visual presentations. To do this, we first pushed the pre-methods to GitHub and then the post-methods. Then we used the git differences as ExRRs. When selecting the ExRRs, we tried to choose examples that were as close to the example as possible (according to code2vec respectively Levenshtein distance) while they should 1) not be too long, 2) not have been changed in too many places (fragmented changes) and 3) be reasonably simple (based on the first author's subjective opinion). Since we wanted to examine the possibility of using refactoring recommendations from other projects, all ExRRs were selected from projects different from the example provided.

The selected examples and their ExRRs are presented in the results in Section 5.3.

**Table 4.2:** Low-level codes.

| Low-level codes | Description |
|---|---|
| Explain example | When an interviewee explains what the interviewee sees in one of the ten examples. |
| Explain expectation | When an interviewee expresses what the interviewee thinks or expects to receive as a refactoring recommendation for the given example. |
| Conclude example | When an interviewee concludes their thoughts. |
| Function extraction | When an interviewee talks about extracting a part of the method. |
| Structural reasoning/control flow | When an interviewee talks about the structure of the code or the control flow. |
| Missing context/unclear consequences or actions | When an interviewee expresses that context is missing or the interviewee does not understand the refactoring recommendation or its consequences. |
| Inspiration/some value | When an interviewee expresses that the refactoring recommendation is inspirational, add some value or points in the right direction but does not meet the expectations. |
| Trivial example/no value/irrelevant | When an interviewee expresses that the refactoring recommendation is too trivial to help, adds no value or is irrelevant. |
| Code removal | When the interviewee states that code has been removed. |
| Size of recommendation/presentation | When an interviewee talks about the size of the refactoring recommendation or how it is presented. |
| Not applicable here | When an interviewee expresses that the refactoring recommendation does not fit the current example. |
| Functionality change | When an interviewee expresses that the functionality has been changed. |
| Concise and easy to comprehend | When an interviewee expresses that a refactoring recommendation is concise and easy to comprehend. |
| Good recommendation | When an interviewee expresses that a refactoring recommendation is a good recommendation for the current example |
| Similarity | When an interviewee sees similarities between the example and the refactoring recommendation. |
| Parameter object | When an interviewee talks about creating a parameter object. |

# Chapter 5
# Result

In this chapter, the results from the pilot experiment, similarity calculations and user interviews will be presented. First, the result from the pilot experiment (Section 5.1) is presented where the manually created methods were analysed (see M1-M39 in Appendix B). The pilot experiment is also presented with a comparison between the similarity measures for the most similar method pairs among the manually created example methods according to the intuitive understanding (Figure 4.2). Then the result from the similarity calculations on the collected methods from the Java projects are presented (Section 5.2). Those parts are presented with distributions and correlations. Finally, the results from the user interviews are presented (Section 5.3). In this part, each example from the interviews will be described with a summary of the interviewees' answers.

## 5.1 Pilot Experiment

In this section, we present the different distributions for Test 1 and Test 2 (Figure 5.1). In those we compared the manually created example methods to each other. For Test 1 M1-M16 are included and for Test 2 M1-M39 are included. Then, we present the ranking of the different similarity measures for the most similar method pairs according to our intuitive understanding (Table 5.1). We will also present the correlation between the different measures (Table 5.2).

In Figure 5.1, the result from code2vec + cosine distance can be seen. For the first test, the distances between the methods appears spread, but as more methods with more changes are added, the distribution becomes skewed toward lower scores, i.e., more similar methods.

In Figure 5.2, the result from Jaccard distance can be seen. For both the first and the second test, the distances between the methods are spread. Though the methods appears to be more dissimilar according to Jaccard distance than code2vec + cosine distance in the first test.

In Figure 5.3, the result from Levenshtein distance can be seen. For the first test, the

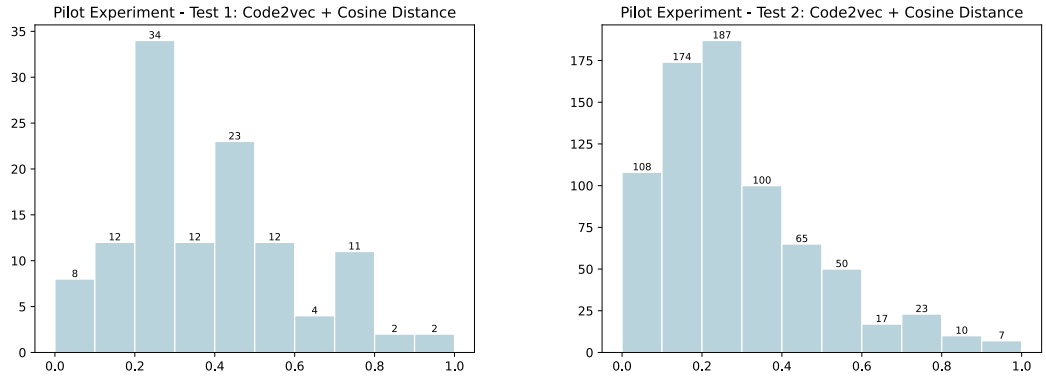



**Figure 5.1:** The normalised distributions of the methods in the pilot experiment for Test 1 and Test 2 when using code2vec + cosine distance as a similarity measure. The x-axis shows the intervals for the distances between every method on a scale from zero to one. The y-axis shows the frequency of value occurrences.

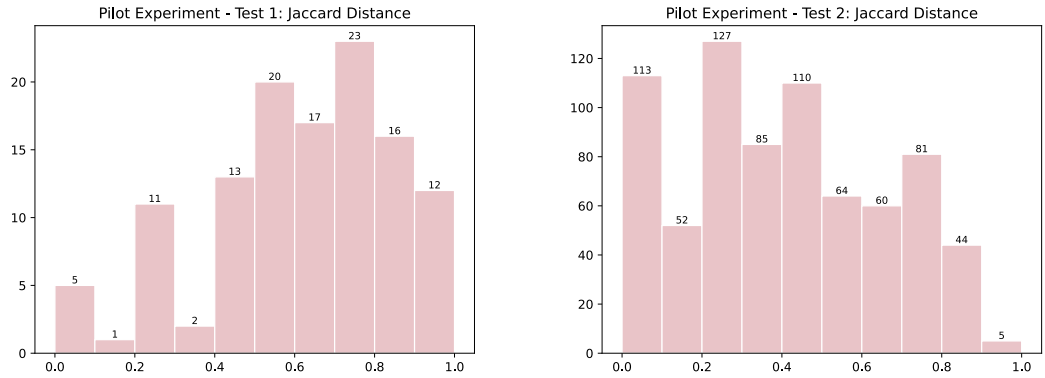



**Figure 5.2:** The normalised distributions of the methods in the pilot experiment for Test 1 and Test 2 when using Jaccard distance as the similarity measure. The x-axis shows the intervals for the distances between every method on a scale from zero to one. The y-axis shows the frequency of value occurrences.

distribution appears to be skewed toward lower scores, i.e., more similar methods. For the second test, the distribution has no clear shape.

In Figure 5.4, the result from JPlag can be seen. Instead of a distance, JPlag outputs the percentage of plagiarism for every method pair in the pairwise comparison. Four Java methods (M1, M13, M15 and M16), were not accepted by JPlag since they contain fewer tokens than the allowed minimum. Therefore, they are not part of the distributions. According to JPlag most methods are considered as dissimilar. The output format from JPlag was not suitable for further analysis and therefore, we chose to continue without JPlag.

In Table 5.1, the most similar method pairs, with their ranking, can be seen according to the intuitive understanding in Figure 4.2. The higher ranking (lower number) of a method pair, the smaller is the distance between them. Those are also ranked according to the intu-

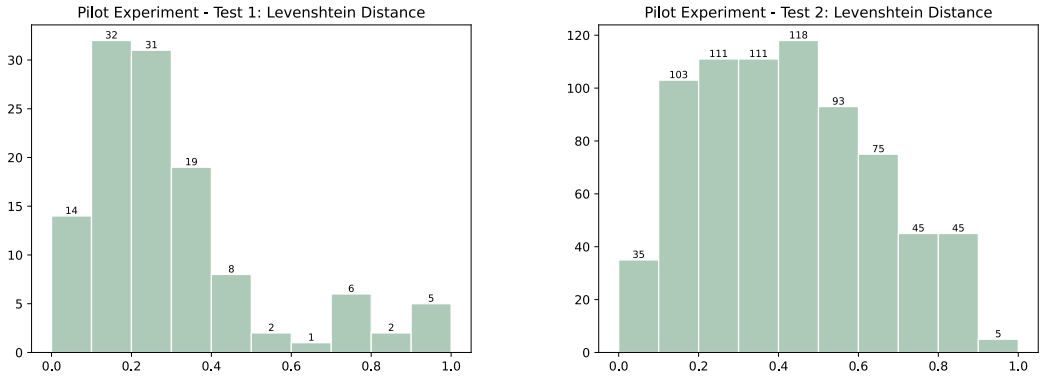



**Figure 5.3:** The normalised distributions of the methods in the pilot experiment for Test 1 and Test 2 when using Levenshtein distance as the similarity measure. The x-axis shows the intervals for the distances between every method on a scale from zero to one. The y-axis shows the frequency of value occurrences.

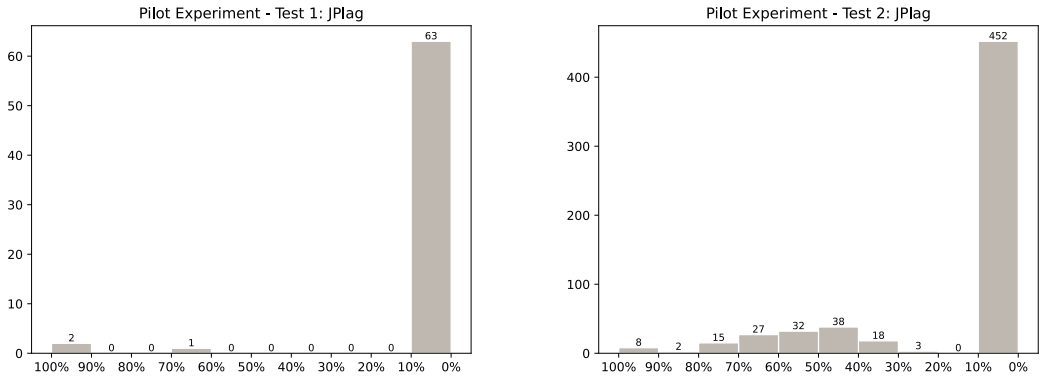



**Figure 5.4:** The distributions of the methods in the pilot experiment for Test 1 and Test 2 (except M1, M13, M15 and M16) when using JPlag as the similarity measure. The x-axis shows the intervals for the percentage of plagiarism for every method pair. The y-axis shows the frequency of value occurrences.

itive understanding, code2vec + cosine distance, Jaccard distance and Levenshtein distance. For Test 1, the lowest ranking was 120 and for Test 2, the lowest ranking was 741.

When looking at code2vec + cosine distance, renaming of a method is ranked first for both tests, as for the intuitive understanding. The method pair *addForLoop* and *addForLoopV2* is ranked as number two in both tests. Therefore, this change of a for loop is also considered as a small change according to code2vec + cosine distance. What stands out is the changes of a parameter. Renaming, adding and deleting a parameter gets a very low ranking compared to the intuitive understanding which means that code2vec + cosine distance consider those actions as a big change while the intuitive understanding does not. Other than that, the ranking for code2vec + cosine distance is not that different from the intuitive understanding in the first test while it differs more in the second test.

Jaccard distance also ranked the method pair *addForLoop* and *addForLoopV2* as number two but Jaccard distance's number one is the change from five lines of printing statements to ten lines of printing statements. This change, from five lines to ten, stands out for Levenshtein distance. It is considered as a big change compared to the intuitive understanding and the other method pairs for Levenshtein distance. Other than that, Levenshtein distance consider the current method pairs as highly ranked in both tests apart from *addForLoopV2X3+IfStatementX3* and *addForLoopV3X3+IfStatementX3* and *addForLoopX3+IfStatementV3X3* and *addForLoopX3+IfStatementX*.

**Table 5.1:** The most similar method pairs according to the intuitive understanding, with ranking for the intuitive understanding, code2vec + cosine distance, Jaccard distance and Levenshtein distance. The higher ranking (lower number) of a method pair, the smaller is the distance between them.

| **Test 1** | | | | | |
|---|---|---|---|---|---|
| **method 1** | **method 2** | **Intuitive Underst.** | **C2v+cos Dist.** | **Jacc Dist.** | **Lev Dist.** |
| M1: addition | M15: renameMethod | 1 | 1 | 45 | 3 |
| M1: addition | M16: renameParameter | 1 | 99 | 18 | 3 |
| M4: addForLoop | M5: addForLoopV2 | 3 | 2 | 2 | 1 |
| M6: addIfStatement | M7: addIfStatementV2 | 3 | 7 | 6 | 2 |
| M1: addition | M8: addParameter | 5 | 29 | 12 | 5 |
| M1: addition | M13: deleteParameter | 5 | 70 | 19 | 5 |
| M3: addFiveLines | M10: addTenLines | 5 | 5 | 1 | 102 |
| M4: addForLoop | M12: addWhileLoop | 8 | 16 | 14 | 13 |
| M5: addForLoopV2 | M12: addWhileLoop | 8 | 15 | 13 | 13 |
| **Test 2** | | | | | |
| **method 1** | **method 2** | **Intuitive Underst.** | **C2v+cos Dist.** | **Jacc Dist.** | **Lev Dist.** |
| M1: addition | M15: renameMethod | 1 | 1 | 453 | 5 |
| M1: addition | M16: renameParameter | 1 | 676 | 279 | 5 |
| M4: addForLoop | M5: addForLoopV2 | 3 | 2 | 37 | 1 |
| M6: addIfStatement | M7: addIfStatementV2 | 3 | 44 | 122 | 2 |
| M25: addForLoopV2X3 +IfStatementX3 | M24: addForLoopV2X3 +IfStatementV2X3 | 3 | 19 | 88 | 5 |
| M25: addForLoopV2X3 +IfStatementX3 | M26: addForLoopV3X3 +IfStatementX3 | 3 | 99 | 81 | 70 |
| M30: addForLoopX3 +IfStatementV2X3 | M31: addForLoopX3 +IfStatementV3X3 | 3 | 46 | 132 | 118 |
| M30: addForLoopX3 +IfStatementV2X3 | M32: addForLoopX3 +IfStatementX3 | 3 | 21 | 81 | 5 |
| M31: addForLoopX3 +IfStatementV3X3 | M32: addForLoopX3 +IfStatementX3 | 3 | 29 | 116 | 97 |

In Table 5.2, the correlations between the different similarity measures can be seen, when

they are applied to the manually created example methods (M1-M39). The correlation between code2vec + cosine distance and Jaccard distance is moderate while the correlation between Levenshtein distance and Jaccard distance is weak. The correlation between code2vec + cosine distance and Levenshtein distance is very weak.

**Table 5.2:** Correlation between the similarity measures in the pilot experiment.

| Similarity measures | Correlation coefficient | P-value |
|---|---|---|
| Levenshtein - Jaccard | 0.347 | 0.001 |
| Code2vec + Cosine - Jaccard | 0.444 | 0.001 |
| Code2vec + Cosine - Levenshtein | 0.152 | 0.015 |

## 5.2 Similarity Calculations

In this section, we will present the different distributions that we received when comparing all of the pre-methods to every other pre-method with the different similarity measures (Figure 5.5). In total, the number of pairwise comparisons presented in the following figures are 1,033,203. We will also present the correlations between the similarity measures (Table 5.3).

In Figure 5.5, the distribution for code2vec + cosine distance can be seen. The z-score for the skewtest is -204 and the p-value is 0. Therefore, the distribution is negatively skewed. In this case, there are a few pairs of methods that are more similar than others.

In Figure 5.6, the distribution for Jaccard distance can be seen. The z-score for the skewtest is 60 and the p-value is 0. Therefore, this distribution is slighlty positively skewed.

In Figure 5.7, the distribution for Levenshtein distance can be seen. The z-score for the skewtest is 673 and the p-value is 0. Therefore, this distribution is a positively skewed which means that the majority of the methods are considered similar to each other according to Levenshtein distance.

We also calculated the correlation between the different similarity measures. Those can be seen in Table 5.3. The correlation between Levenshtein distance and Jaccard distance is negative and very weak. The correlation between cosine distance and Jaccard distance is positive and very weak, even though it is a bit higher. There is no significant correlation between code2vec + cosine distance and Levenshtein distance.

**Table 5.3:** Correlation between similarity measures when using methods from top-starred Java projects on GitHub.

| Similarity measures | Correlation coefficient | P-value |
|---|---|---|
| Levenshtein - Jaccard | -0.089 | 0.001 |
| Code2vec + Cosine - Jaccard | 0.139 | 0.001 |
| Code2Vec + Cosine - Levenshtein | 0.006 | 0.511 |

For the evaluation, we wanted a set-up that resembles the envisioned future user experience in CodeScene. Therefore, we wanted to show one ExRR at a time. To be able to compare

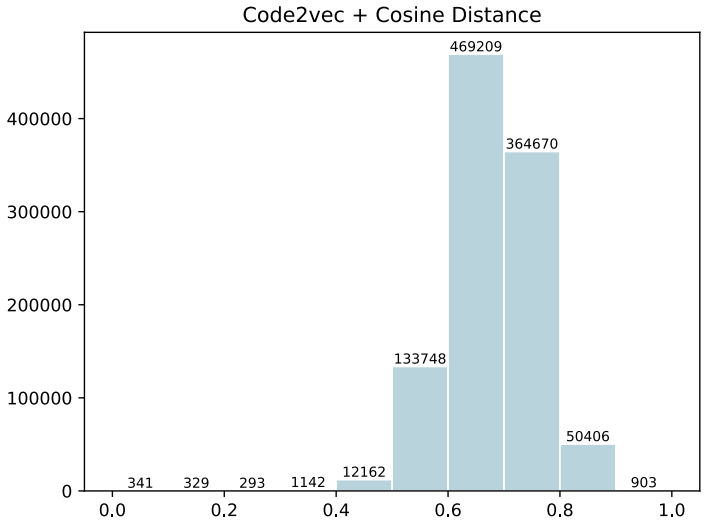



**Figure 5.5:** The normalised distribution of the pre-methods when using code2vec + cosine distance as similarity measure. The x-axis shows the intervals for the distances between every pre-method on a scale from zero to one. The y-axis shows the frequency of value occurrences.

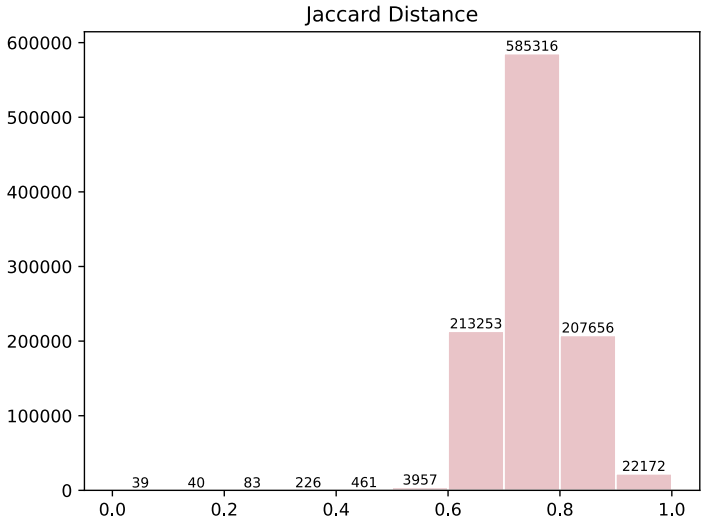



**Figure 5.6:** The normalised distribution of the pre-methods when using Jaccard distance as the similarity measure. The x-axis shows the intervals for the distances between every pre-method on a scale from zero to one. The y-axis shows the frequency of value occurrences.

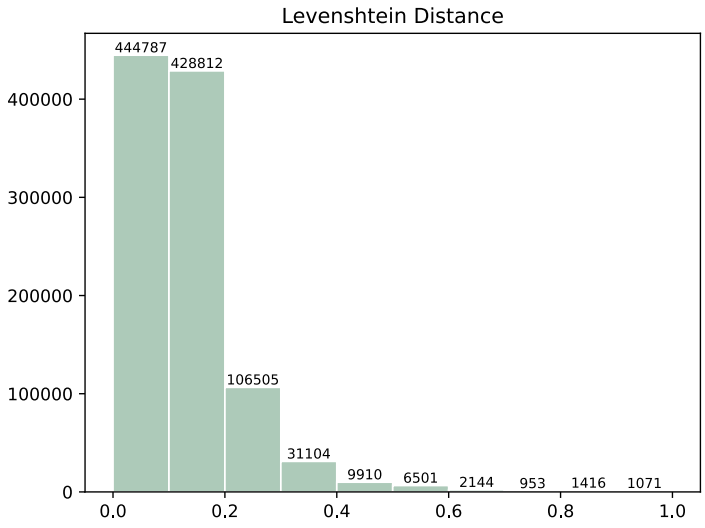



**Figure 5.7:** The normalised distribution of the pre-methods when using Levenshtein distance as the similarity measure. The x-axis shows the intervals for the distances between every pre-method on a scale from zero to one. The y-axis shows the frequency of value occurrences.

ExRRs without having them side by side, we thought that two ExRRs would be manageble to keep in mind. We wanted to continue with two measures that interpreted similarity differently. Code2vec + cosine distance and Levenshtein distance seemed to interpret similarity differently since the distribution for code2vec + cosine distance is clearly negatively skewed while the distribution for Levenshtein distance is clearly positively skewed. Thus, those were good candidates. Code2vec + cosine distance and Jaccard distance had the strongest correlation for both the GitHub methods and for the pilot experiment. Therefore, we wanted only one of them. According to this, and that code2vec is considered as state-of-the-art for measuring code similarity at the time, we chose to continue with code2vec + cosine distance and Levenshtein distance.

## 5.3 User Interviews

In this section, we will present all ten examples that were part of the user interviews. For every example, we will first present the distances between the example and the ExRRs. Then, we will summarise the interviewees' answers regarding ExRR A, then regarding ExRR B and then regarding who preferred which ExRR. In the end of every example we will also reflect on the findings.

In the end of the section, we will summarise the findings (Section 5.3.11) and address what the interviewees thought was a good refactoring recommendation (Section 5.3.12).

### 5.3.1 Example 1

The first example contains the code smell *bumpy road ahead* and can be seen in Figure 5.8.

```
private CompletableFuture<Channel> connectToAddress(InetAddress ipAddress, int port, InetSocketAddress sniHost) {
    CompletableFuture<Channel> future = new CompletableFuture<>();
    // if proxy is configured in pulsar-client then make it thread-safe while updating channelInitializerHandler
    if (isSniProxy) {
        bootstrap.register().addListener((ChannelFuture cf) -> {
            if (!cf.isSuccess()) {
                future.completeExceptionally(cf.cause());
                return;
            }
            Channel channel = cf.channel();
            try {
                channelInitializerHandler.initChannel(channel, sniHost);
                channel.connect(new InetSocketAddress(ipAddress, port)).addListener((ChannelFuture channelFuture) -> {
                    if (channelFuture.isSuccess()) {
                        future.complete(channelFuture.channel());
                    } else {
                        future.completeExceptionally(channelFuture.cause());
                    }
                });
            } catch (Exception e) {
                log.warn("Failed to initialize channel with {}, {}", ipAddress, sniHost, e);
                future.completeExceptionally(e);
            }
        });
    } else {
        bootstrap.connect(ipAddress, port).addListener((ChannelFuture channelFuture) -> {
            if (channelFuture.isSuccess()) {
                future.complete(channelFuture.channel());
            } else {
                future.completeExceptionally(channelFuture.cause());
            }
        });
    }
    return future;
}
```

**Figure 5.8:** Example 1 in the user interviews.

Option A is a close recommendation based on Levenshtein distance and can be seen in Figure 5.9. The Levenshtein distance between this ExRR's pre-method and the example is 1,084 and the code2vec + cosine distance is 0.885. It is ranked as number 14,967 for Levenshtein distance and 25,083 for code2vec + cosine distance. The total number of ranked pairs is 93,434, which represents the number of pairwise combinations between methods within the code smell *bumpy road ahead* that are not originating from the same project.

Option B can be seen in Figure 5.10. It is a close recommendation according to code2vec + cosine distance and has the code2vec + cosine distance 0.594. The Levenshtein distance is 1,497. It is ranked as number 200 for code2vec + cosine distance and 35,687 for Levenshtein distance.

P1 liked A because it *"follows some pattern that seems to remove stuff or move logic to a more appropriate place"*. P3 and P5 also liked that A extracts parts in the method. P2 and P4 did not think A was helpful. P2 thought that A did not solve the problem with *bumpy road ahead* and that *"it probably was part of a bigger refactoring"*. P4 thought that some code was removed and would rather have seen a function extraction.

> "*/...Extract a small piece and give it a name and reduce the complexity of this function.../*" - P3 about A in the first example

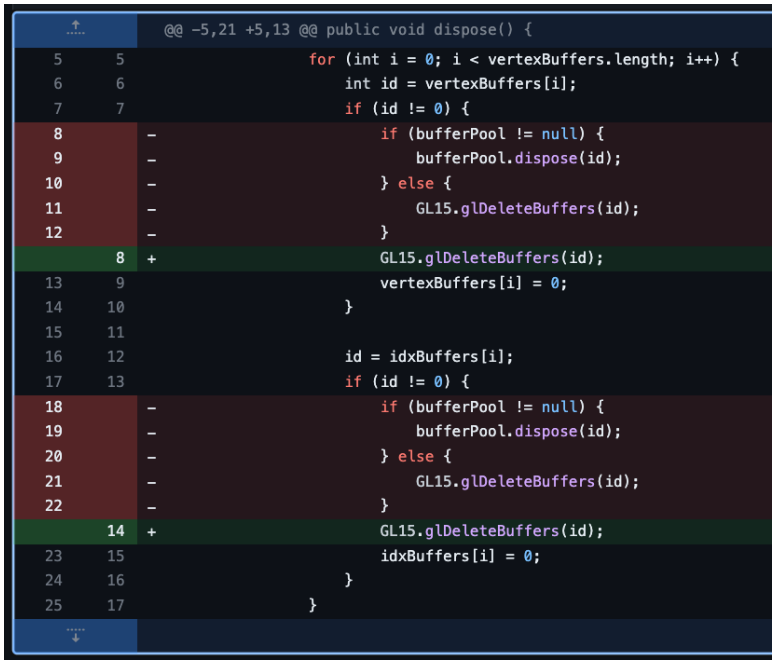



**Figure 5.9:** Option A for the first example. This was selected as a close recommendation based on Levenshtein distance. The Levenshtein distance is 1,084 and the code2vec + cosine distance is 0.885.

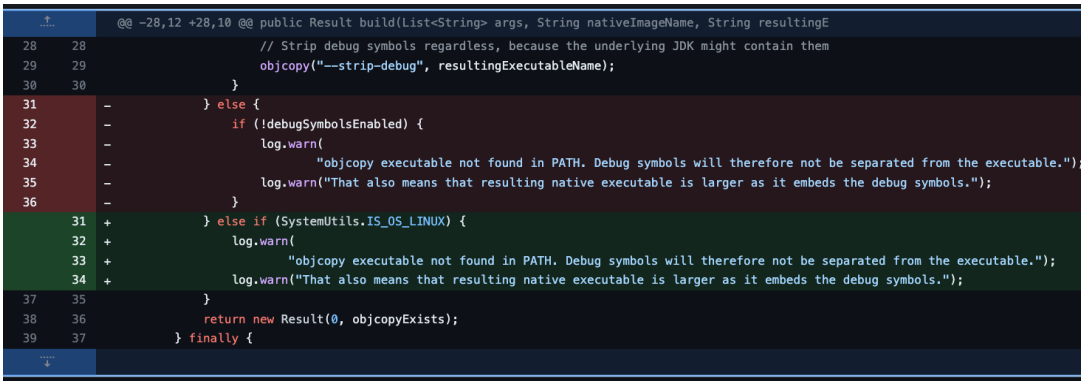



**Figure 5.10:** Option B for the first example. This was selected as a close recommendation based on code2vec + cosine distance. The code2vec + cosine distance is 0.594 and the Levenshtein distance is 1,497.

> *"/…Here, it feels like it just removes a bit of the condition in the example in the previous one and it's not at all sure that it's fully applicable to what you can do in the actual code here. Because in it, I think that there you probably get more work to do with breaking out functions…/"* - P4 about A in the first example

For B, both P3 and P1 claimed that the change in B is not a big change, or basically the same. P2 thought that B gives you an idea of where to start but that it is a bit too simple to solve the problem in the example. P4 thought that B is a good refactoring but not applicable to this example.

> "/…structurally it looks good, we have changed an else and a nested if statement that was empty to an else if, it will be a little better but is really the same thing…/" - P1 about B in the first example

> "/…the example that we have with the bumpy road is much more complex and so this might help you get started down the right path, I guess, but it's quite a bit simpler than what you need for solving the problem in the example…/" - P2 about B in the first example

P1 and P3 preferred A whereas P2 and P5 preferred B. When forced to make a choice, P4 preferred A.

In the first example, the interviewees were divided about which ExRR was the best one. Three interviewees preferred A and two B. In A three people perceived that code had been extracted while one perceived that it only had been removed. In B we interpret that two of the interviewees considered B too trivial to be helpful.

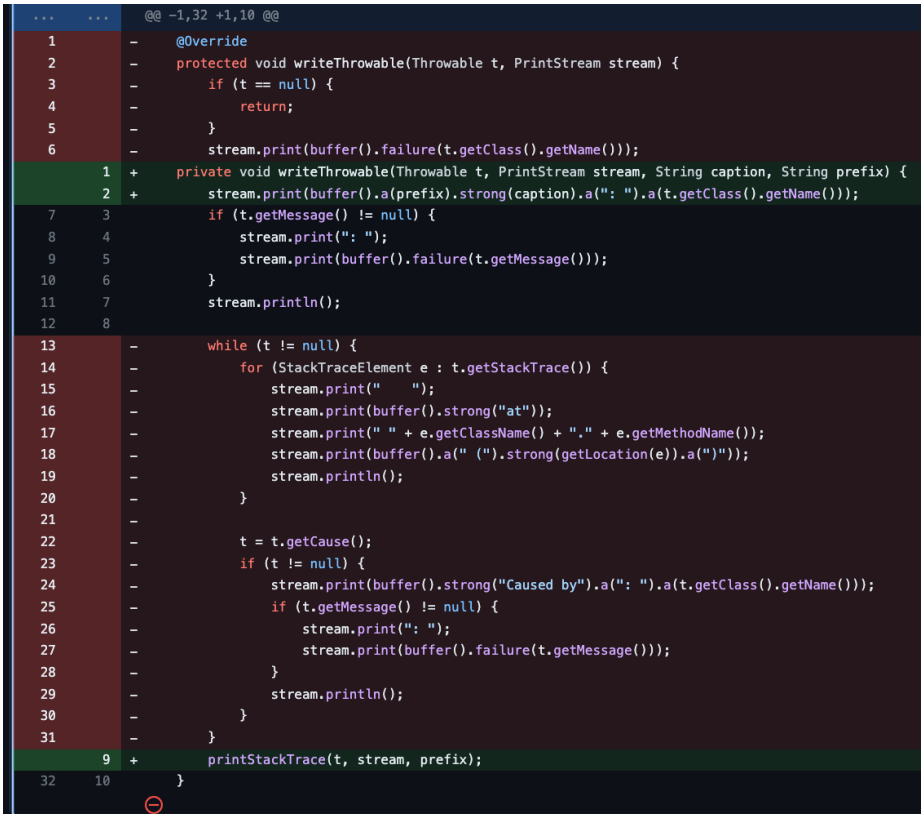
### 5.3.2 Example 2

The second example also contains the code smell *bumpy road ahead* and can be seen in Figure 5.11.

```
@Override
protected Iterator<E> flatMap(final Traverser.Admin<Vertex> traverser) {
    Iterable<? extends JanusGraphElement> result;

    if (useMultiQuery) {
        if (multiQueryResults == null || !multiQueryResults.containsKey(traverser.get())) {
            prefetchNextBatch(traverser); // current batch is exhausted, fetch new batch
        }
        result = multiQueryResults.get(traverser.get());
    } else {
        final JanusGraphVertexQuery query = makeQuery((JanusGraphTraversalUtil.getJanusGraphVertex(traverser)).query());
        result = (Vertex.class.isAssignableFrom(getReturnClass())) ? query.vertices() : query.edges();
    }

    if (batchPropertyPrefetching && txVertexCacheSize > 1) {
        Set<Vertex> vertices = new HashSet<>();
        for(JanusGraphElement v : result){
            vertices.add((Vertex) v);
            if (vertices.size() >= txVertexCacheSize) {
                break;
            }
        }

        // If there are multiple vertices then fetch the properties for all of them in a single multiQuery to
        // populate the vertex cache so subsequent queries of properties don't have to go to the storage back end
        if (vertices.size() > 1) {
            JanusGraphMultiVertexQuery propertyMultiQuery = JanusGraphTraversalUtil.getTx(traversal).multiQuery();
            ((BasicVertexCentricQueryBuilder) propertyMultiQuery).profiler(queryProfiler);
            propertyMultiQuery.addAllVertices(vertices).preFetch();
        }
    }

    return (Iterator<E>) result.iterator();
}
```

**Figure 5.11:** Example 2 in the user interviews.

Option A is a close recommendation based on Levenshtein distance and can be seen in Figure 5.12. The Levenshtein distance between this ExRR's pre-method and the example is



```
@@ -1,32 +1,10 @@
-    @Override
-    protected void writeThrowable(Throwable t, PrintStream stream) {
-        if (t == null) {
-            return;
-        }
-        stream.print(buffer().failure(t.getClass().getName()));
+    private void writeThrowable(Throwable t, PrintStream stream, String caption, String prefix) {
+        stream.print(buffer().a(prefix).strong(caption).a(": ").a(t.getClass().getName()));
         if (t.getMessage() != null) {
             stream.print(": ");
             stream.print(buffer().failure(t.getMessage()));
         }
         stream.println();

-        while (t != null) {
-            for (StackTraceElement e : t.getStackTrace()) {
-                stream.print("    ");
-                stream.print(buffer().strong("at"));
-                stream.print(" " + e.getClassName() + "." + e.getMethodName());
-                stream.print(buffer().a(" (").strong(getLocation(e)).a(")"));
-                stream.println();
-            }
-
-            t = t.getCause();
-            if (t != null) {
-                stream.print(buffer().strong("Caused by").a(": ").a(t.getClass().getName()));
-                if (t.getMessage() != null) {
-                    stream.print(": ");
-                    stream.print(buffer().failure(t.getMessage()));
-                }
-                stream.println();
-            }
-        }
+        printStackTrace(t, stream, prefix);
     }
```

**Figure 5.12:** Option A for the second example. This was selected as a close recommendation based on Levenshtein distance. The Levenshtein distance is 1,113 and the code2vec + cosine distance is 0.992.

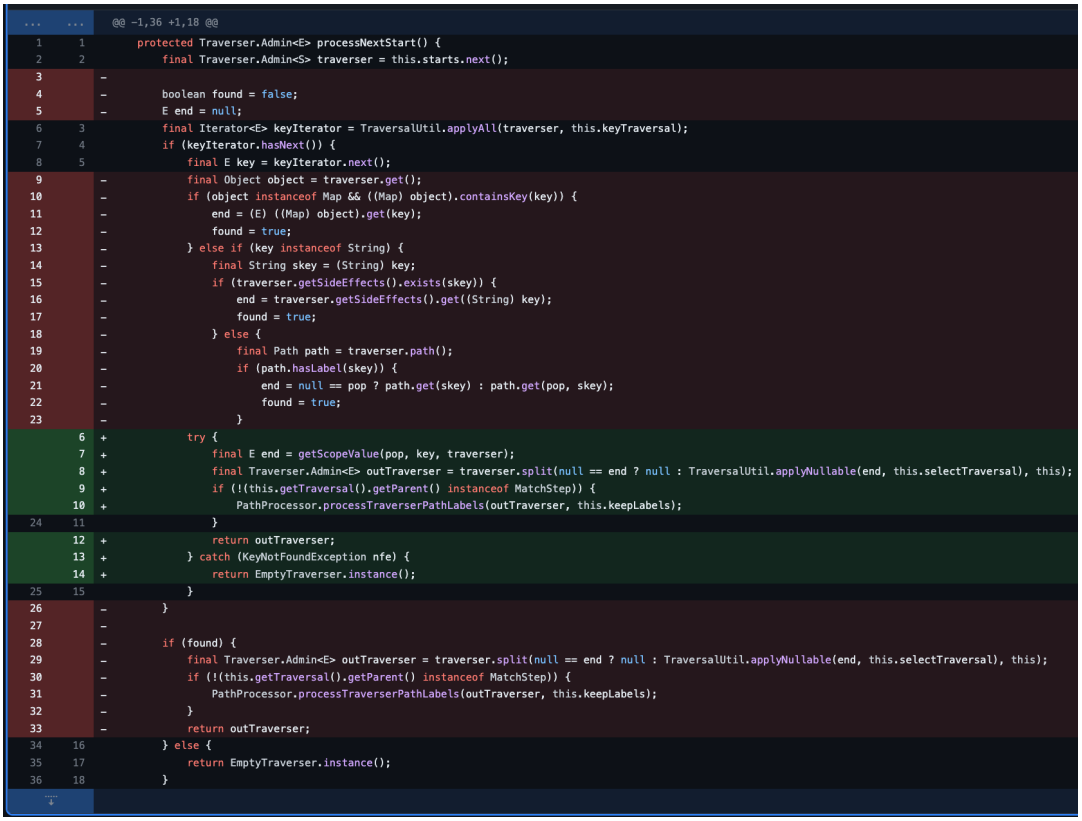
1,113 and the cosine distance is 0.992. It is ranked as number 16,181 for Levenshtein distance and 62,033 for code2vec + cosine distance.

Option B can be seen in Figure 5.13. It is a close recommendation according to code2vec + cosine distance and has a code2vec + cosine distance of 0.424. The Levenshtein distance is 1,178. It is ranked as number 26 for code2vec + cosine distance and 18,945 for Levenshtein distance.

In A P1, P3, P4 and P5 pointed out that a part of the code is extracted. P3 and P4 expressed that something similar could be done in the example. P1 thought this *could* be good and P4 thought that this *is* good. P4 said that: *"It was still, spontaneously, quite a good example of what can be broken out and simplified."*. P2 thought that A would be a bit hard to apply to the example. P2 thought that A could be used as inspiration but it does not tell the developer where to begin and therefore it can be difficult to use if the developer is a novice developer.

> "/…*So, we removed 15 lines of code and extracted it to a method which is good…*/" - P5 about A in the second example

> "/…*I would say this is kind of inspirational in the sense that people should see this and start looking for ways to do the same thing. It doesn't really provide you with, I'm thinking if someone is new to refactoring or might not know, this doesn't really tell them where to start, but it does kind of give you an idea…*/" - P2 about A in the second example



```
@@ -1,36 +1,18 @@
    protected Traverser.Admin<E> processNextStart() {
        final Traverser.Admin<S> traverser = this.starts.next();
-
-       boolean found = false;
-       E end = null;
        final Iterator<E> keyIterator = TraversalUtil.applyAll(traverser, this.keyTraversal);
        if (keyIterator.hasNext()) {
            final E key = keyIterator.next();
-           final Object object = traverser.get();
-           if (object instanceof Map && ((Map) object).containsKey(key)) {
-               end = (E) ((Map) object).get(key);
-               found = true;
-           } else if (key instanceof String) {
-               final String skey = (String) key;
-               if (traverser.getSideEffects().exists(skey)) {
-                   end = traverser.getSideEffects().get((String) key);
-                   found = true;
-               } else {
-                   final Path path = traverser.path();
-                   if (path.hasLabel(skey)) {
-                       end = null == pop ? path.get(skey) : path.get(pop, skey);
-                       found = true;
-                   }
-               }
+           try {
+               final E end = getScopeValue(pop, key, traverser);
+               final Traverser.Admin<E> outTraverser = traverser.split(null == end ? null : TraversalUtil.applyNullable(end, this.selectTraversal), this);
+               if (!(this.getTraversal().getParent() instanceof MatchStep)) {
+                   PathProcessor.processTraverserPathLabels(outTraverser, this.keepLabels);
                }
+               return outTraverser;
+           } catch (KeyNotFoundException nfe) {
+               return EmptyTraverser.instance();
            }
-       }
-
-       if (found) {
-           final Traverser.Admin<E> outTraverser = traverser.split(null == end ? null : TraversalUtil.applyNullable(end, this.selectTraversal), this);
-           if (!(this.getTraversal().getParent() instanceof MatchStep)) {
-               PathProcessor.processTraverserPathLabels(outTraverser, this.keepLabels);
-           }
-           return outTraverser;
        } else {
            return EmptyTraverser.instance();
        }
```

**Figure 5.13:** Option B for the second example. This was selected as a close recommendation based on code2vec + cosine distance. The code2vec + cosine distance is 0.424 and the Levenshtein distance is 1,178.

P2 thought that B *" looks a lot more similar to the example"* while P1 did not think that the loop structure in B matched the loop structure in the example. P2 thought that B generally is a good example of refactoring *bumpy road ahead* but pointed out that the developer have to think a lot itself. P1 pointed out that B is a lot to read and P5 pointed out that a lot of code has been removed. P3 stated that the behaviour also had been changed in B.

> *"/…This is not only a refactoring but it has also slightly changed the behaviour in some way, error handling which has been introduced here. In this case, I would have preferred the first one. It's a little cleaner refactoring. This one I have to sit down and think about what is actually refactored here. What is refactoring and what is handling exceptions?…/"* - P3 about B in the second example

P3, P4 and P5 preferred A and P2 preferred B. P1 did not think that any of the examples were helpful but had a slight preference for A. P5 preferred A since P5 understood A better since it contains less code. P4 also expressed that it was hard to translate an unrelated recommendation to solve the problem, even though it solved the same code smell.

In this example, four interviewees preferred A while one preferred B. Four of the interviewees thought that a part of the code was extracted in A. Those four were the same four that preferred A. For B the reasoning was quite divided.

### 5.3.3 Example 3

The third example contains the code smell *complex method* and can be seen in Figure 5.14.

```
public IColumnType processTypeConvert(GlobalConfig globalConfig, String fieldType) {
    String t = fieldType.toLowerCase();
    if (t.contains("char")) {
        return DbColumnType.STRING;
    } else if (t.contains("bigint")) {
        return DbColumnType.LONG;
    } else if (t.contains("smallint")) {
        return DbColumnType.BASE_SHORT;
    } else if (t.contains("int")) {
        return DbColumnType.INTEGER;
    } else if (t.contains("date") || t.contains("time")
        || t.contains("year") || t.contains("timestamp")) {
        return DbColumnType.DATE;
    } else if (t.contains("text")) {
        return DbColumnType.STRING;
    } else if (t.contains("bit")) {
        return DbColumnType.BOOLEAN;
    } else if (t.contains("decimal")) {
        return DbColumnType.BIG_DECIMAL;
    } else if (t.contains("clob")) {
        return DbColumnType.CLOB;
    } else if (t.contains("blob")) {
        return DbColumnType.BLOB;
    } else if (t.contains("binary")) {
        return DbColumnType.BYTE_ARRAY;
    } else if (t.contains("float")) {
        return DbColumnType.FLOAT;
    } else if (t.contains("double")) {
        return DbColumnType.DOUBLE;
    } else if (t.contains("json") || t.contains("enum")) {
        return DbColumnType.STRING;
    }
    return DbColumnType.STRING;
}
```

**Figure 5.14:** Example 3 in the user interviews.

Option A can be seen in Figure 5.15. It is a close recommendation according to code2vec + cosine distance and has the code2vec + cosine distance 0.524. The Levenshtein distance is 988. It is ranked as number 481 for code2vec + cosine distance and 22,347 for Levenshtein distance. The total rank is 167,850, which represents the number of pairwise combinations between methods within the code smell *complex method* that are not originating from the same project.

Option B is a close recommendation based on Levenshtein distance and can be seen in Figure 5.16. The Levenshtein distance between this ExRR's pre-method and the example is 990 and the code2vec + cosine distance is 0.852. It is ranked as number 22,507 for Levenshtein distance and 26,665 for code2vec + cosine distance.

P3 and P4 noted that the switch statement is replaced with some kind of datatype. P3 pointed out that this was a good idea but was not sure if P3 would have done this type of refactoring in all situations. P4 thought that A is good, simple, and that something similar should be done in the example. P5 stated that A is very good and readable. P2 thought that A is what the example should look like when the refactoring is done but it is up to the developer refactoring to come up with how to get there. P2 addressed that some knowledge and skills are required for that.

```
@@ -1,24 +1,3 @@
-    private EdgeEventActionType edgeTypeByActionType(ActionType actionType) {
-        switch (actionType) {
-            case ADDED:
-                return EdgeEventActionType.ADDED;
-            case UPDATED:
-                return EdgeEventActionType.UPDATED;
-            case ALARM_ACK:
-                return EdgeEventActionType.ALARM_ACK;
-            case ALARM_CLEAR:
-                return EdgeEventActionType.ALARM_CLEAR;
-            case DELETED:
-                return EdgeEventActionType.DELETED;
-            case RELATION_ADD_OR_UPDATE:
-                return EdgeEventActionType.RELATION_ADD_OR_UPDATE;
-            case RELATION_DELETED:
-                return EdgeEventActionType.RELATION_DELETED;
-            case ASSIGNED_TO_EDGE:
-                return EdgeEventActionType.ASSIGNED_TO_EDGE;
-            case UNASSIGNED_FROM_EDGE:
-                return EdgeEventActionType.UNASSIGNED_FROM_EDGE;
-            default:
-                return null;
-        }
+    public static EdgeEventActionType edgeTypeByActionType(ActionType actionType) {
+        return EdgeEventActionType.valueOf(actionType.toString().toUpperCase(Locale.ENGLISH));
     }
```

**Figure 5.15:** Option A for the third example. This was selected as a close recommendation based on code2vec + cosine distance. The the code2vec + cosine distance is 0.524 and Levenshtein distance is 988.

```
@@ -4,7 +4,7 @@ public IWizardPage getPage(String name) {
            if (pageName.equals(name)) {
                return page;
            }
-           if (page instanceof ICompositeDialogPage && !(page instanceof ConnectionPageSettings)) {
+           if (page instanceof ICompositeDialogPage) {
                final IDialogPage[] subPages = ((ICompositeDialogPage) page).getSubPages(false);
                if (subPages != null) {
                    for (IDialogPage subPage : subPages) {
```

**Figure 5.16:** Option B for the third example. This was selected as a close recommendation based on Levenshtein distance. The Levenshtein distance is 990 and the code2vec + cosine distance is 0.852.

> "*Okay, so this is really good, this is really readable. So you have not so much repetition of the code and you have exactly the action types. You can see them really clearly. And the final method, the edgeTypeByActionType is so simple. So this is a good example of refactoring.*" - P5 about A in the third example

P1, P2, P3 and P4 thought that B was not relevant or not applicable to the example. P2 thought that B does not fix the real problem and pointed out that it could be used for only a part of the code. P3 thought that the example probably has another cyclomatic complexity. P5 was not sure why this was a refactoring example at all.

> "/… *That didn't say much, I must say.…/*" - P4 about B in the third example

Everyone preferred A. P2 preferred A since P2 thought it points in the right direction while P2 thought that B is more detailed.

For example 3, everyone agreed that A was more helpful. However, there was a difference in how much they liked A. We interpret that everyone found B irrelevant. We interpret that everyone was slightly positive to A and quite negative to B, i.e., A was a clear winner.

### 5.3.4 Example 4

The fourth example also contains the code smell *complex method* and can be seen in Figure 5.17.

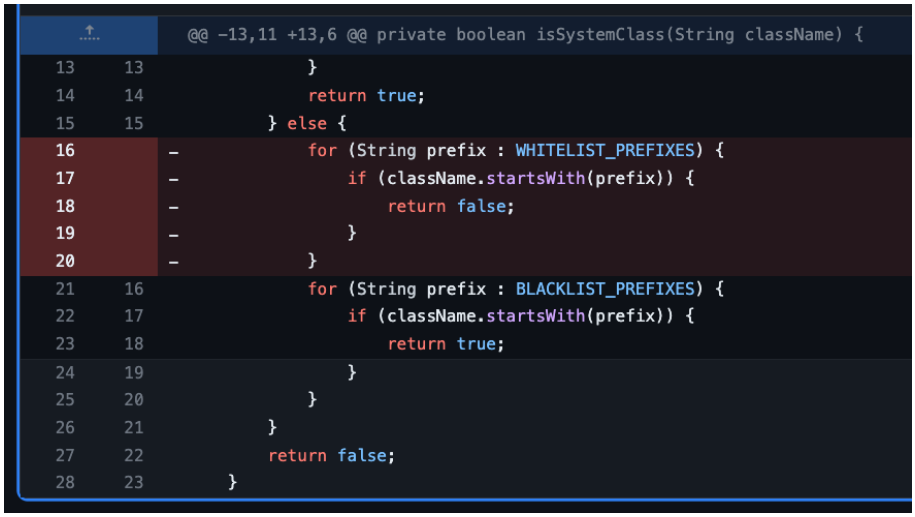
```
private static HttpEntity asHttpEntity(byte[] data, Exchange exchange) throws Exception {
    AbstractHttpEntity entity;

    String contentEncoding = null;
    if (exchange != null) {
        contentEncoding = exchange.getIn().getHeader(HttpConstants.CONTENT_ENCODING, String.class);
    }

    if (exchange != null && !exchange.getProperty(Exchange.SKIP_GZIP_ENCODING, Boolean.FALSE, Boolean.class)) {
        boolean gzip = GZIPHelper.isGzip(contentEncoding);
        if (gzip) {
            InputStream stream = GZIPHelper.compressGzip(contentEncoding, data);
            entity = new InputStreamEntity(
                    stream, stream instanceof ByteArrayInputStream
                            ? stream.available() != 0 ? stream.available() : -1 : -1);
        } else {
            // use a byte array entity as-is
            entity = new ByteArrayEntity(data);
        }
    } else {
        // create the Repeatable HttpEntity
        entity = new ByteArrayEntity(data);
    }
    if (exchange != null) {
        if (contentEncoding != null) {
            entity.setContentEncoding(contentEncoding);
        }
        String contentType = ExchangeHelper.getContentType(exchange);
        if (contentType != null) {
            entity.setContentType(contentType);
        }
    }
    return entity;
}
```

**Figure 5.17:** Example 4 in the user interviews.

Option A is a close recommendation based on Levenshtein distance and can be seen in Figure 5.18. The Levenshtein distance between this ExRR's pre-method and the example is 947 and the code2vec + cosine distance is 0.930. It is ranked as number 19,371 for Levenshtein distance and 66,045 for code2vec + cosine distance.

Option B can be seen in Figure 5.19. It is a close recommendation according to code2vec + cosine distance and has the code2vec + cosine distance 0.474. The Levenshtein distance is 1,405. It is ranked as number 420 for code2vec + cosine distance and 58,342 for Levenshtein distance.

P2 did not think that A was that useful since it was hard to understand if the code was only simplified or if the behaviour also was changed. P3 did not think that A is helpful since



```
@@ -13,11 +13,6 @@ private boolean isSystemClass(String className) {
13  13              }
14  14              return true;
15  15          } else {
16      -           for (String prefix : WHITELIST_PREFIXES) {
17      -               if (className.startsWith(prefix)) {
18      -                   return false;
19      -               }
20      -           }
21  16              for (String prefix : BLACKLIST_PREFIXES) {
22  17                  if (className.startsWith(prefix)) {
23  18                      return true;
24  19                  }
25  20              }
26  21          }
27  22          return false;
28  23      }
```

**Figure 5.18:** Option A for the fourth example. This was selected as a close recommendation based on Levenshtein distance. The Levenshtein distance is 947 and the code2vec + cosine distance is 0.930.

```
@@ -1,13 +1,7 @@
1   1   private List<Message<byte[]>> getMessageFromHttpResponse(String topic, Response response) throws Exception {
2   2
3   3       if (response.getStatus() != Status.OK.getStatusCode()) {
4       -       if (response.getStatus() >= 500) {
5       -           throw new ServerErrorException(response);
6       -       } else if (response.getStatus() >= 400) {
7       -           throw new ClientErrorException(response);
8       -       } else {
9       -           throw new WebApplicationException(response);
10      -       }
        4   +   throw getApiException(response);
11  5       }
12  6
13  7       String msgId = response.getHeaderString("X-Pulsar-Message-ID");
```

**Figure 5.19:** Option B for the fourth example. This was selected as a close recommendation based on code2vec. The code2vec + cosine distance is 0.474 and the Levenshtein distance is 1,405.

functionality *has been removed* and the cyclomatic complexity decreased as a result of that. P4 thought that A is somehow relevant but P4 would rather oneself figure out how to extract code. P5 expressed confusion as to what was happening in A.

> "*This one is not very helpful because this one has only removed code. Removed a functionality that was there before. Has reduced its cyclomatic complexity, but it has been reduced because some functionality has been removed here, I think. Of course, it could have been moved somewhere else, but it's hard to tell from just the small code snippet, so it's not that helpful.*" - P3 about A in the fourth example

> "*Yes. Fairly relevant, but I wouldn't have looked so much at that example if I had it shown to myself, but I would probably have sat and figured out for myself how to break out. How to create that entity in this code primarily, and focus on that instead.*" - P4 about A in the fourth example

P2 thought that B was a clear example and that it was an extract method refactoring. P3, P4 and P5 also noted that a block of logic is extracted. Therefore P5 liked B more. P3 and P5 noted that this change could also be applied to the example.

> "*A logical block. Instead of having it here, we just extracted it to a method and do all the logic there. So I like this one. And this all logic will be probably in this getApiException.../*" - P5 about B in the fourth example

Everyone preferred B. P1 did not like neither A nor B but found B slightly better. P3 pointed out that P3 preferred B since it is a refactoring, but also said that that it is only inspirational, while A was only removing a snippet of code.

In this example, everyone preferred B but as for the last example there was a difference in how much they liked B. Two interviewees raised that functionality was removed both in A and in B. Four interviewees raised that code had been extracted. When looking at that reasoning, we think that the interviewees choice is aligned with the definition of a refactoring, that the external behaviour should not be changed.

### 5.3.5 Example 5

The fifth example contains the code smell *deep, nested complexity* and can be seen in Figure 5.20.

```
private void addDiscoveredRoutes(CamelContext camelContext, List<RoutesBuilder> routes) throws Exception {
    // sort routes according to ordered
    routes.sort(OrderedComparator.get());

    if (modeLine) {
        ModeLineFactory factory = resolveModelineFactory(camelContext);
        if (factory != null) {
            List<Resource> resources = new ArrayList<>();
            // gather resources for modeline
            for (RoutesBuilder builder : routes) {
                if (builder instanceof RouteBuilder) {
                    resources.add(((RouteBuilder) builder).getResource());
                }
            }
            for (Resource resource : resources) {
                LOG.debug("Parsing modeline: {}", resource);
                factory.parseModeLine(resource);
            }
        }
    }

    // first add the routes configurations as they are globally for all routes
    for (RoutesBuilder builder : routes) {
        if (builder instanceof RouteConfigurationsBuilder) {
            RouteConfigurationsBuilder rcb = (RouteConfigurationsBuilder) builder;
            LOG.debug("Adding routes configurations into CamelContext from RouteConfigurationsBuilder: {}", rcb);
            camelContext.addRoutesConfigurations(rcb);
        }
    }
    // then add the routes
    for (RoutesBuilder builder : routes) {
        LOG.debug("Adding routes into CamelContext from RoutesBuilder: {}", builder);
        camelContext.addRoutes(builder);
    }
}
```

**Figure 5.20:** Example 5 in the user interviews.

Option A is a close recommendation based on Levenshtein distance and can be seen in Figure 5.21. The Levenshtein distance between this ExRR's pre-method and the example is 1,021 and the code2vec + cosine distance is 0.983. It is ranked as number 5,645 for Levenshtein

distance and 39,557 for code2vec + cosine distance. The total rank is 62,640, which represents the number of pairwise combinations between methods within the code smell *deep, nested complexity* that are not originating from the same project.

```
@@ -1,26 +1,36 @@
     private void dispatch(boolean isBegin, String log) {
-        synchronized (listeners) {
-            for (LooperDispatchListener listener : listeners) {
-                if (listener.isValid()) {
-                    if (isBegin) {
-                        if (!listener.isHasDispatchStart) {
-                            if (listener.historyMsgRecorder) {
-                                messageStartTime = System.currentTimeMillis();
-                                latestMsgLog = log;
-                                recentMCount++;
-                            }
-                            listener.onDispatchStart(log);
-                        }
-                    } else {
-                        if (listener.isHasDispatchStart) {
-                            if (listener.historyMsgRecorder) {
-                                recordMsg(log, System.currentTimeMillis() - messageStartTime, listener.denseMsgTracer);
-                            }
-                            listener.onDispatchEnd(log);
-                        }
+        if (isBegin) {
+            if (historyMsgRecorder) {
+                messageStartTime = System.currentTimeMillis();
+                latestMsgLog = log;
+                recentMCount++;
+            }
+            synchronized (listeners) {
+                for (LooperDispatchListener listener : listeners) {
+                    listener.onDispatchStart(log);
+                }
+            }
+            synchronized (listenersMap) {
+                for (DispatchListenerWrapper listener : listenersMap.values()) {
+                    if (listener.isValid() && !listener.isHasDispatchStart) {
+                        listener.onDispatchBegin(log);
+                    }
+                }
+            }
+        } else {
+            if (historyMsgRecorder) {
+                recordMsg(log, System.currentTimeMillis() - messageStartTime);
+            }
+            synchronized (listeners) {
+                for (LooperDispatchListener listener : listeners) {
+                    listener.onDispatchEnd(log);
+                }
+            }
+            synchronized (listenersMap) {
+                for (DispatchListenerWrapper listener : listenersMap.values()) {
+                    if (listener.isValid() || listener.isHasDispatchStart) {
+                        listener.onDispatchEnd(log);
                     }
-                } else if (!isBegin && listener.isHasDispatchStart) {
-                    listener.dispatchEnd();
                 }
             }
         }
```

**Figure 5.21:** Option A for the fifth example. This was selected as a close recommendation based on Levenshtein distance. The Levenshtein distance is 1,021 and the code2vec + cosine distance is 0.983.

Option B can be seen in Figure 5.22. It is a close recommendation according to code2vec + cosine distance and has the code2vec + cosine distance 0.441. The Levenshtein distance is 1,222. It is ranked as number 5 for code2vec + cosine distance and 9,887 for Levenshtein distance.

```
@@ -2,11 +2,9 @@ protected synchronized List<Route> getRoutes(String ... excluded_sites) {
 2  2          List<Route> retval=new ArrayList<>(routes.size());
 3  3          for(List<Route> list: routes.values()) {
 4  4              for(Route route: list) {
 5     -            if(route != null) {
 6     -                if(!isExcluded(route, excluded_sites)) {
 7     -                    retval.add(route);
 8     -                    break;
 9     -                }
    5  +            if (route != null && !isExcluded(route, excluded_sites)) {
    6  +                retval.add(route);
    7  +                break;
10  8              }
11  9          }
12 10      }
```

**Figure 5.22:** Option B for the fifth example. This was selected as a close recommendation based on code2vec. The code2vec + cosine distance is 0.441 and the Levenshtein distance is 1,222.

P1 stated that: *"Here we also have quite the same structure"* and P2 thought that A was a nice recommendation since *"it does show how they managed to eliminate several layers of conditionals"*. P3 noted that A is less nested but have about the same cyclomatic complexity as the example. P3 did not like A since P3 thought that more should be done in the recommendation and pointed out parts to extract. P4 also mentioned that A is less nested but that it was: *"a lot to take in to translate that into what you would like to do with this code that you have in front of you"*. P5 also thought that A is too long and pointed out that refactoring recommendations should make it easier and therefore you should not have to spend much time on analysing them.

> "*I have to do some thinking when I look at this example. Has it gotten easier? It's less nesting. Obvious. Cyclomatic complexity is about the same. I think neither before nor after is particularly good. The example after has an if (isBegin) and a big block that is completely independent of the big block down there. It would have been very easy to split these blocks into two functions in that case. And then we have things that look very similar, synchronised (listenersMap), in the example after the refactoring, is refactored but very obvious that you can make things even simpler. That was my reflection. So I would have looked at that and thought, well, this was no good.*" - P3 about A in the fifth example

P1 questioned the helpfulness of B. P2 found B (compared to A) closer to what should be done in the example but that it is only a small step on the way. P2 thought that somehow you would want to show both examples. P3 thought that B is a very simple way of reducing the nesting and thus it does not bring much inspiration. P4 thought that B is short and concise which P4 found good. P5 thought it was good to combine the if statements instead of having two since there is no else clause.

> "*/...I would have liked, generally if you refactor these kinds of methods, you want to separate iterating over something from doing something on what you iterate over. So I would have looked at this one and gone one step further. Picked out that green in a function. So that the for loops were one thing and what you did with what you iterated over was another thing.../*" - P3 about B in the fifth example

> *"It was a little kinder. Short and concise. It gets plus points for that alone and also shows how to keep in nesting as much as possible. So of these two, I definitely think B was the most useful."* - P4 about B in the fifth example

P1 and P2 preferred A. On the contrary, P3, P4 and P5 preferred B. Though P3 would go further with the refactoring than what is presented in B.

In this example, two interviewees preferred A while three interviewees preferred B. The reasoning was quite divided but for A, two people raised that the refactoring made the method less nested and two people raised that they thought that the recommendation was too long. For B, three interviewees thought it provided little or no inspiration while a few interviewees were positive to the recommendation.

### 5.3.6 Example 6

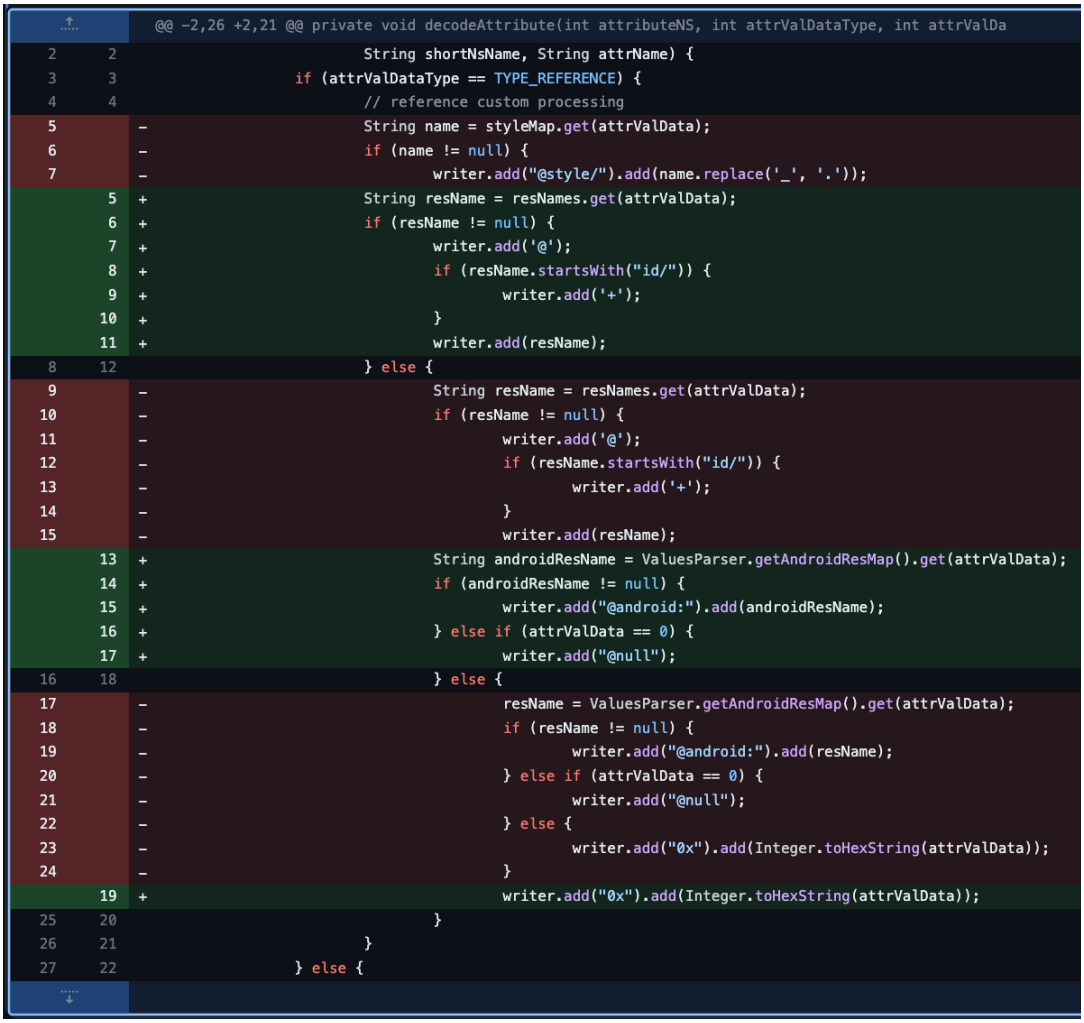
The sixth example also contains the code smell *deep, nested complexity* and can be seen in Figure 5.23.

```
private void updateIfAssertIsPresentTrueOnOptional(
    Context context,
    MethodInvocationNode node,
    Types types,
    AccessPath.AccessPathContext apContext,
    AccessPathNullnessPropagation.Updates bothUpdates) {
  Node receiver = node.getTarget().getReceiver();
  if (receiver instanceof MethodInvocationNode) {
    MethodInvocationNode receiverMethod = (MethodInvocationNode) receiver;
    Symbol.MethodSymbol receiverSymbol = ASTHelpers.getSymbol(receiverMethod.getTree());
    if (methodNameUtil.isMethodAssertThat(receiverSymbol)) {
      // assertThat will always have at least one argument, So safe to extract from the arguments
      Node arg = receiverMethod.getArgument(0);
      if (arg instanceof MethodInvocationNode) {
        // Since assertThat(a.isPresent()) changes to
        // Truth.assertThat(Boolean.valueOf(a.isPresent()))
        // need to be unwrapped from Boolean.valueOf
        Node unwrappedArg = ((MethodInvocationNode) arg).getArgument(0);
        if (unwrappedArg instanceof MethodInvocationNode) {
          MethodInvocationNode argMethod = (MethodInvocationNode) unwrappedArg;
          Symbol.MethodSymbol argSymbol = ASTHelpers.getSymbol(argMethod.getTree());
          if (optionalIsPresentCall(argSymbol, types)) {
            updateNonNullAPsForOptionalContent(
                context, bothUpdates, argMethod.getTarget().getReceiver(), apContext);
          }
        }
      }
    }
  }
}
```

**Figure 5.23:** Example 6 in the user interviews.

Option A can be seen in Figure 5.24. It is a close recommendation according to code2vec + cosine distance and has the code2vec + cosine distance 0.560. The Levenshtein distance is 1,219. It is ranked as number 49 for code2vec + cosine distance and 9,819 for Levenshtein distance.

Option B is a close recommendation based on Levenshtein distance and can be seen in Figure 5.25. The Levenshtein distance between this ExRR's pre-method and the example is



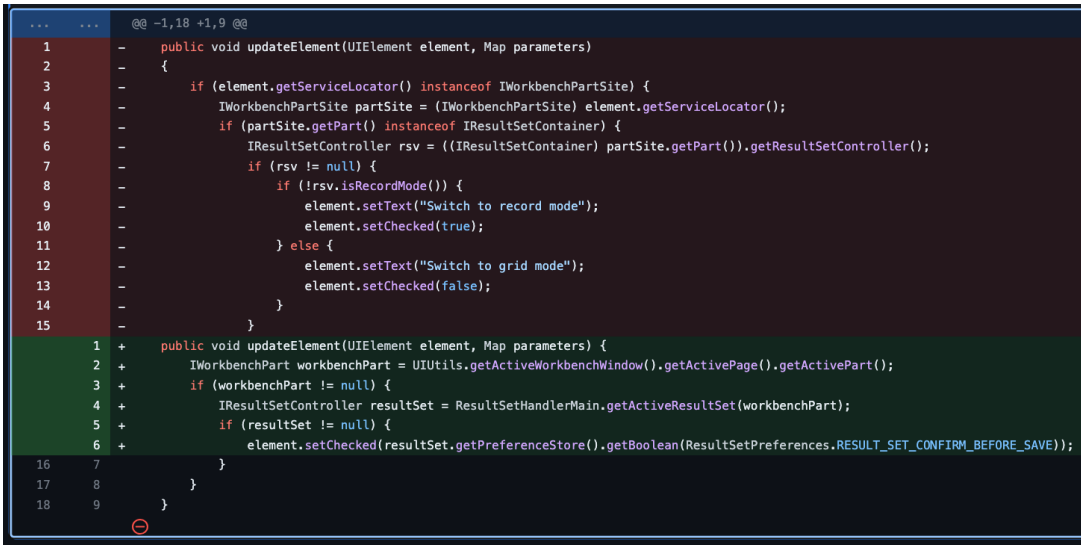
```
@@ -2,26 +2,21 @@ private void decodeAttribute(int attributeNS, int attrValDataType, int attrValDa
                        String shortNsName, String attrName) {
                if (attrValDataType == TYPE_REFERENCE) {
                        // reference custom processing
-                       String name = styleMap.get(attrValData);
-                       if (name != null) {
-                               writer.add("@style/").add(name.replace('_', '.'));
+                       String resName = resNames.get(attrValData);
+                       if (resName != null) {
+                               writer.add('@');
+                               if (resName.startsWith("id/")) {
+                                       writer.add('+');
+                               }
+                               writer.add(resName);
                        } else {
-                               String resName = resNames.get(attrValData);
-                               if (resName != null) {
-                                       writer.add('@');
-                                       if (resName.startsWith("id/")) {
-                                               writer.add('+');
-                                       }
-                                       writer.add(resName);
+                               String androidResName = ValuesParser.getAndroidResMap().get(attrValData);
+                               if (androidResName != null) {
+                                       writer.add("@android:").add(androidResName);
                                } else if (attrValData == 0) {
                                        writer.add("@null");
                                } else {
-                                       resName = ValuesParser.getAndroidResMap().get(attrValData);
-                                       if (resName != null) {
-                                               writer.add("@android:").add(resName);
-                                       } else if (attrValData == 0) {
-                                               writer.add("@null");
-                                       } else {
-                                               writer.add("0x").add(Integer.toHexString(attrValData));
-                                       }
+                                       writer.add("0x").add(Integer.toHexString(attrValData));
                                }
                        }
                } else {
```

**Figure 5.24:** Option A for the sixth example. This was selected as a close recommendation based on code2vec. The code2vec + cosine distance is 0.560 and the Levenshtein distance is 1,219.

1,076 and the code2vec + cosine distance is 0.933. It is ranked as number 6,719 for Levenshtein distance and 27,668 for code2vec + cosine distance.

P1 did not think that A was helpful with the nesting in the example. P2 thought that A was a good example and that A is not entirely solving the same problem as in the example but gives you a general idea. Both P3 and P5 thought that A was a bit hard to absorb due to the diff view. P3 also thought that A before refactoring is quite trivial and probably a case where it is easy to reduce the nesting while A after the refactoring is rather complex. P4 and P5 thought that A was big.

> "*/… This example removes some nesting, it does, but it was quite a lot to take in. Relatively large. Lots of ifs and so on too. Spontaneously, a bit difficult to translate it to, to apply it to the code example.*" - P4 about A in the sixth example



```
@@ -1,18 +1,9 @@
-    public void updateElement(UIElement element, Map parameters)
-    {
-        if (element.getServiceLocator() instanceof IWorkbenchPartSite) {
-            IWorkbenchPartSite partSite = (IWorkbenchPartSite) element.getServiceLocator();
-            if (partSite.getPart() instanceof IResultSetContainer) {
-                IResultSetController rsv = ((IResultSetContainer) partSite.getPart()).getResultSetController();
-                if (rsv != null) {
-                    if (!rsv.isRecordMode()) {
-                        element.setText("Switch to record mode");
-                        element.setChecked(true);
-                    } else {
-                        element.setText("Switch to grid mode");
-                        element.setChecked(false);
-                    }
-                }
+    public void updateElement(UIElement element, Map parameters) {
+        IWorkbenchPart workbenchPart = UIUtils.getActiveWorkbenchWindow().getActivePage().getActivePart();
+        if (workbenchPart != null) {
+            IResultSetController resultSet = ResultSetHandlerMain.getActiveResultSet(workbenchPart);
+            if (resultSet != null) {
+                element.setChecked(resultSet.getPreferenceStore().getBoolean(ResultSetPreferences.RESULT_SET_CONFIRM_BEFORE_SAVE));
             }
         }
     }
```

**Figure 5.25:** Option B for the sixth example. This was selected as a close recommendation based on Levenshtein distance. The Levenshtein distance is 1,076 and the code2vec + cosine distance is 0.933.

> "*/…it's a big example again. So you have to analyse what's happening here and then you have to see what's deleted and what's added. Maybe a side by side view would be better than the combined one.*" - P5 about A in the sixth example

P1 did not think that B was a perfect match but could bring inspiration for delegating to another class. P2 thought that B was really suitable due to the reduction from four levels of conditionals to two. P3 stated that B is more relevant and more similar to the example. P3 explained that instead of having several if statements in the example, those could be extracted into a function and thereby separated from the inner logic. P3 thought that something similar was done in B. P4 thought that B is more comprehensible than A and the interviewee would have done something similar with the example. P5 thought that B was more readable than A but was not sure if it removed functionality or not.

> "*Yes, I think the second one is more relevant because even there, there is some type of delegation to another class, maybe you could be inspired by it a little here but not a spot on example here either, but it is probably difficult to recommend something good for this that has five ifs in a row with business logic, I don't expect to be able to get any good recommendations for this either.*" - P1 about B in the sixth example

> "*/…they had four levels of conditionals and now they've got it down to two. So this is exactly what needs to happen in six. I'm not totally sure how they did it…/*" - P2 about B in the sixth example

Everyone preferred B.

In this example, everyone agreed that B was the better refactoring recommendation. The reasoning about A differs quite much but some interviewees thought that A was hard to understand due to the diff view and/or the size of the recommendation. The reasoning for B also differs.

### 5.3.7 Example 7

The seventh example contains the code smell *excess number of function argument* and can be seen in Figure 5.26.

```
public static HeroicConfig create(
    Optional<String> id,
    Duration startTimeout,
    Duration stopTimeout,
    Optional<String> host,
    Optional<Integer> port,
    List<JettyServerConnector> connectors,
    boolean enableCors,
    Optional<String> corsAllowOrigin,
    FeatureSet features,
    ClusterManagerModule cluster,
    MetricManagerModule metric,
    MetadataManagerModule metadata,
    SuggestManagerModule suggest,
    CacheModule cache,
    IngestionModule ingestion,
    List<ConsumerModule> consumers,
    Optional<ShellServerModule> shellServer,
    AnalyticsModule analytics,
    CoreGeneratorModule generator,
    StatisticsModule statistics,
    QueryLoggingModule queryLogging,
    Optional<ConditionalFeatures> conditionalFeatures,
    TracingConfig tracing,
    UsageTrackingModule usageTracking,
    String version,
    String service,
    String commit
) {
    return new AutoValue_HeroicConfig(id, startTimeout, stopTimeout, host, port, connectors,
        enableCors, corsAllowOrigin, features, cluster, metric, metadata,
        suggest, cache, ingestion, consumers, shellServer, analytics, generator, statistics,
        queryLogging, conditionalFeatures, tracing, usageTracking, version, service, commit);
}
```

**Figure 5.26:** Example 7 in the user interviews.

Option A is a close recommendation based on Levenshtein distance and can be seen in Figure 5.27. The Levenshtein distance between this ExRR's pre-method and the example is 986 and the code2vec + cosine distance is 0.727. It is ranked as number 7,591 for Levenshtein distance and 1,153 for code2vec + cosine distance. The total rank is 19,840 which represents the number of pairwise combinations between methods within the code smell *excess number of function arguments* that are not originating from the same project.

Option B can be seen in Figure 5.28. It is a close recommendation according to code2vec + cosine distance and has the code2vec + cosine distance 0.595. The Levenshtein distance is 4,283. It is ranked as number 54 for code2vec + cosine distance and 18,849 for Levenshtein distance.

P1 liked A since P1 thought that introducing a context object would be a good idea. P2 and P3 assumed that *context* contains the parameters that were removed. Therefore P3 thought that A was a good recommendation. P2 pointed out that this kind of solution is used in those cases. P4 also thought that A was a good example but pointed out that you still have to insert all this information somewhere. P5 did not think that A was a good recommendation since it only showed a change from six parameters to two and did not show what *context* is.

```
@@ -1,23 +1,13 @@
-    public <T extends Method> List<SelectedMethod<T>> getMatchingMethods(Method mappingMethod,
-                                                                         List<T> methods,
-                                                                         List<Type> sourceTypes,
-                                                                         Type mappingTargetType,
-                                                                         Type returnType,
-                                                                         SelectionCriteria criteria) {
+    public <T extends Method> List<SelectedMethod<T>> getMatchingMethods(List<T> methods,
+                                                                         SelectionContext context) {

         List<SelectedMethod<T>> candidates = new ArrayList<>( methods.size() );
         for ( T method : methods ) {
             candidates.add( new SelectedMethod<>( method ) );
         }

         for ( MethodSelector selector : selectors ) {
-            candidates = selector.getMatchingMethods(
-                mappingMethod,
-                candidates,
-                sourceTypes,
-                mappingTargetType,
-                returnType,
-                criteria );
+            candidates = selector.getMatchingMethods( candidates, context );
         }
         return candidates;
     }
```

**Figure 5.27:** Option A for the seventh example. This was selected as a close recommendation based on Levenshtein distance. The Levenshtein distance is 986 and the code2vec + cosine distance is 0.727.

> "*Um, exactly. This makes quite a bit of sense. Squeeze multiple things into objects. Then in cases like this, you always end up having to create a selection context or somehow still get all this information in somewhere. But it's a good suggestion, I think.*" - P4 about A in the seventh example

P1 and P3 thought that B was about the same type of refactoring as A. However, P3 described B more as a chain since the input object is also passed on to something else. P2 thought that B also works for the task and pointed out that there are not many ways of reducing the number of function arguments. P5 likes B better since it is shown what is in the *request*.

> "*Yes, and this is probably something similar, but this is a bit more of a chain because here you pass on the object that you receive to something else as well. That should have been mainly done in the left-hand method as well (the example). Is also an okay example of what you can do. Which is the best of these then? I do not know. The left method there doesn't do much, it just forwards everything. I think both work well here in this case. Take A if I have to choose. Was a little cleaner.*" - P3 about B in the seventh example

P1, P3 and P4 preferred A and P2 and P5 preferred B. P2 preferred B since the original object was passed down instead of creating a new kind of context object. P3 thought that A was a bit cleaner than B and therefore preferred A.

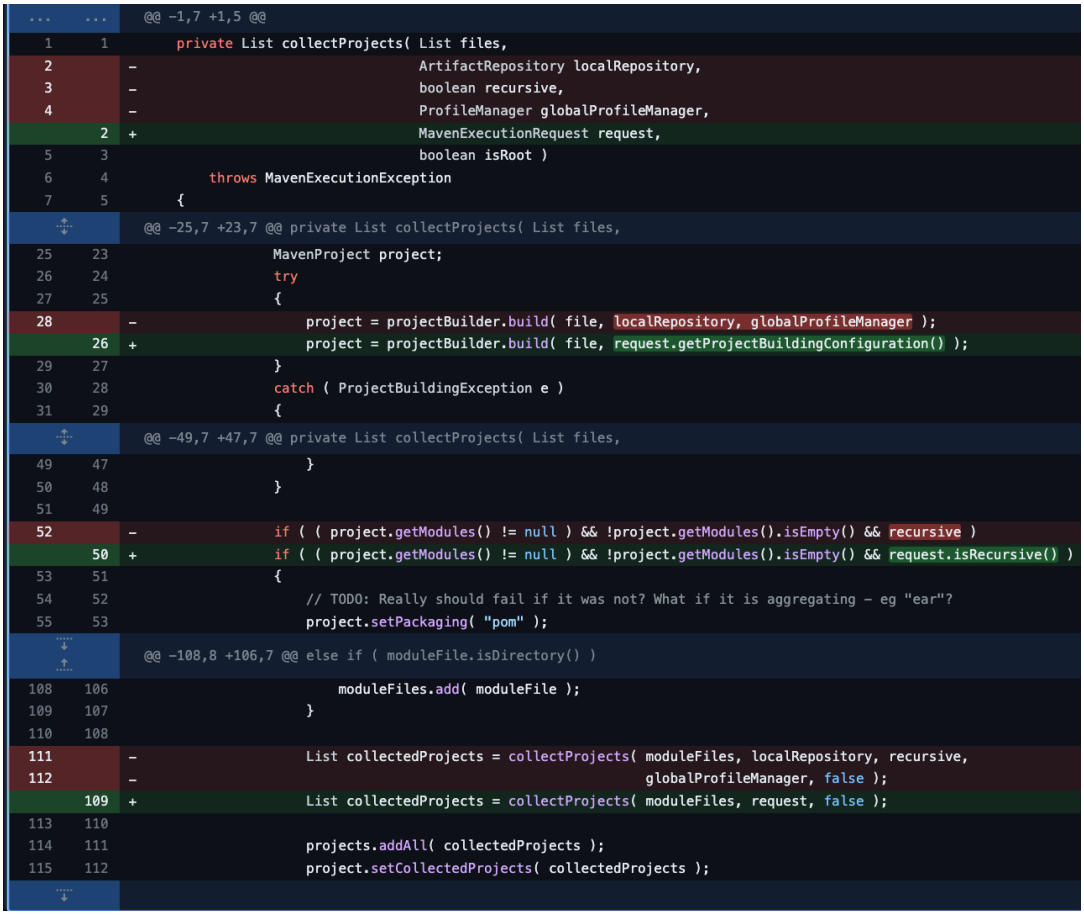



**Figure 5.28:** Option B for the seventh example. This was selected as a close recommendation based on code2vec. The code2vec + cosine distance is 0.595 and the Levenshtein distance is 4,283.

> *"/…both are good. The first one is better. It actually points to the only solution to this that I know of. When you have something like this, or rebuild the entire system."* - P1 about A and B in the seventh example

In this example, the interviewees disagreed on which was most helpful but everyone somehow talked about encapsulating parameters in an object.

### 5.3.8 Example 8

The eighth example also contains the code smell *excess number of function arguments* and can be seen in Figure 5.29.

Option A can be seen in Figure 5.30. It is a close recommendation according to code2vec + cosine distance and has the code2vec + cosine distance 0.591. The Levenshtein distance is 2,647. It is ranked as number 44 for code2vec + cosine distance and 17,134 for Levenshtein distance.

Option B is a close recommendation based on Levenshtein distance and can be seen in Figure 5.31. The Levenshtein distance between this ExRR's pre-method and the example is 921 and the code2vec + cosine distance is 0.870. It is ranked as number 6,831 for Levenshtein

```
public void mirror(String origin, String destination, List<Refspec> refspec, boolean prune,
    boolean partialFetch)
    throws RepoException, ValidationException {
  GitRepository repo = gitOptions.cachedBareRepoForUrl(origin);
  List<String> fetchRefspecs = refspec.stream()
      .map(r -> r.originToOrigin().toString())
      .collect(Collectors.toList());

  generalOptions.console().progressFmt("Fetching from %s", origin);

  Profiler profiler = generalOptions.profiler();
  try (ProfilerTask ignore = profiler.start("fetch")) {
    repo.fetch(origin, /*prune=*/true, /*force=*/true, fetchRefspecs, partialFetch);
  }

  if (generalOptions.dryRunMode) {
    generalOptions.console().progressFmt("Skipping push to %s. You can check the"
        + " commits to push in: %s", destination, repo.getGitDir());
    return;
  }

  generalOptions.console().progressFmt("Pushing to %s", destination);
  List<Refspec> pushRefspecs = forcePush
      ? refspec.stream().map(Refspec::withAllowNoFastForward).collect(Collectors.toList())
      : refspec;

  try (ProfilerTask ignore = profiler.start("push")) {
    repo.push().prune(prune).withRefspecs(destination, pushRefspecs).run();
  }
}
```

**Figure 5.29:** Example 8 in the user interviews.

distance and 8,646 for code2vec + cosine distance.

P2 and P4 noted that the method definition is missing. Thus, P2 did not think that A was a good recommendation. P4 did not think that B was a bad recommendation but did not think that it was a perfect match for this code smell. P4 also pointed out that in the process some complexity has been removed. P3 stated that in A, the change probably is made to change how the objects are connected to each other rather than refactor to get fewer arguments. Therefore, P3 did not think that B was helpful. P5 did not think that B was an example of removing the code smell *excess number of function arguments*.

> "*Not really maybe. Trying to scroll up a bit to see the function signature up there, it has removed some stuff. Ah, but it was a decent example anyway. Have removed some complexity in the process. Spontaneously, for this smell, maybe it wasn't spot on. Not a bad example, a little big though.*" - P4 about A in the eighth example

> "*Here, I have to think a little more, because here it doesn't feel like you've just refactored, but that you've also changed something, in how they interact. So you may not have done this refactoring in the first place to get fewer arguments or that you have simplified something, but that you have actually changed how the objects are connected here. I think this particular one is not very helpful then. It feels more like it has slightly changed the behaviour of how these different objects are connected.*" - P3 about A in the eighth example

P3 thought that B is cleaner than A since the options are encapsulated. P4 expressed that B is more relevant and useful and pointed out that almost all arguments have been taken care of.

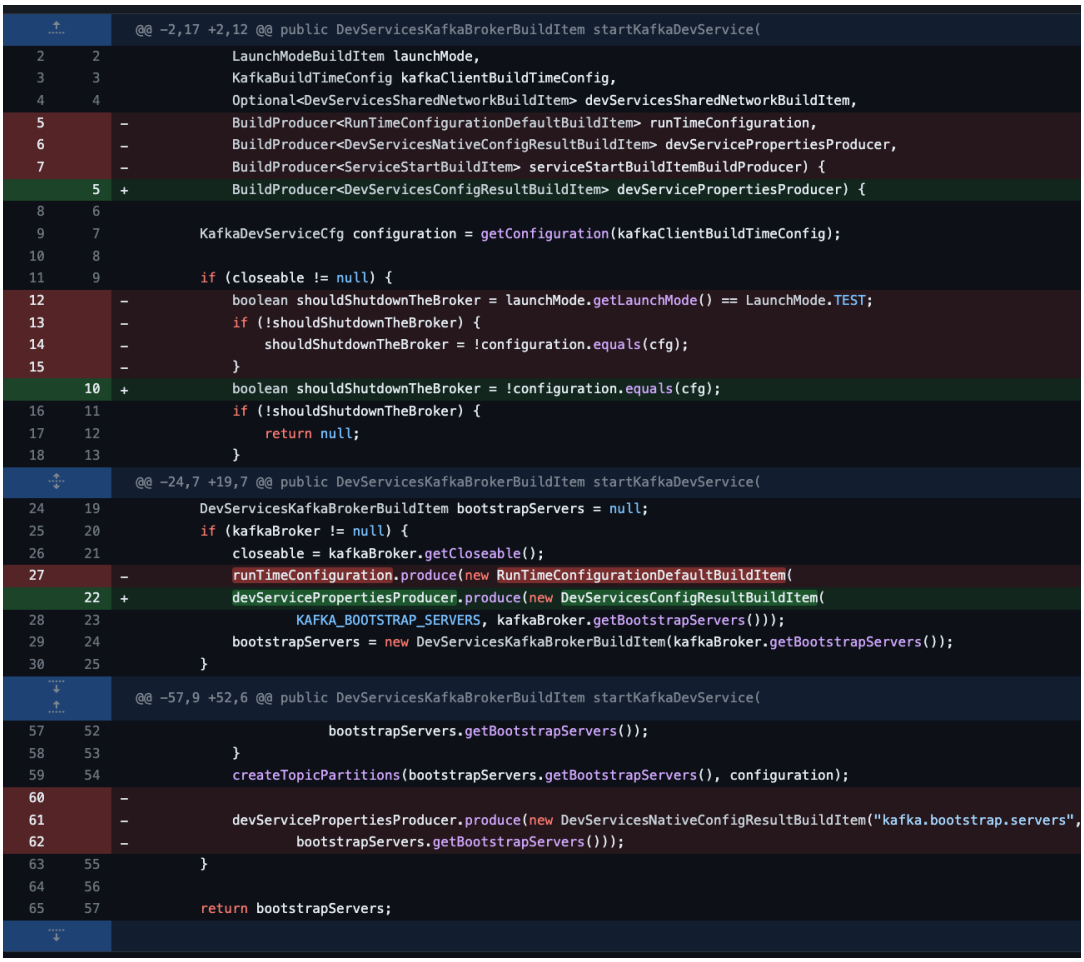



**Figure 5.30:** Option A for the eighth example. This was selected as a close recommendation based on code2vec. The code2vec + cosine distance is 0.591 and the Levenshtein distance is 2,647.

```
@@ -1,21 +1,20 @@
-    public Iterable<Object> loadObjects(Class<?> entityClass,
-                                        Optional<FilterExpression> filterExpression,
-                                        Optional<Sorting> sorting,
-                                        Optional<Pagination> pagination,
+    public Iterable<Object> loadObjects(EntityProjection projection,
                                         RequestScope requestScope) {
-        if (!filterExpression.isPresent()) {
-            return super.loadObjects(entityClass, filterExpression, sorting, pagination, requestScope);
+        if (projection.getFilterExpression() == null) {
+            return super.loadObjects(projection, requestScope);
         }

-        boolean canSearch = (canSearch(entityClass, filterExpression.get()) != NONE);
+        boolean canSearch = (canSearch(projection.getType(), projection.getFilterExpression()) != NONE);

-        if (mustSort(sorting, entityClass)) {
-            canSearch = canSearch && canSort(sorting.get(), entityClass);
+        if (mustSort(Optional.ofNullable(projection.getSorting()))) {
+            canSearch = canSearch && canSort(projection.getSorting(), projection.getType());
         }

         if (canSearch) {
-            return search(entityClass, filterExpression.get(), sorting, pagination);
+            return search(projection.getType(), projection.getFilterExpression(),
+                    Optional.ofNullable(projection.getSorting()),
+                    Optional.ofNullable(projection.getPagination()));
         }

-        return super.loadObjects(entityClass, filterExpression, sorting, pagination, requestScope);
+        return super.loadObjects(projection, requestScope);
     }
```

**Figure 5.31:** Option B for the eighth example. This was selected as a close recommendation based on Levenshtein distance. The Levenshtein distance is 921 and the code2vec + cosine distance is 0.870.

P1 preferred A whereas P2, P3, P4 and P5 preferred B.

In this example, all but one interviewee preferred B. Two interviewees did not think that A fit as a recommendation for the code smell. The interviewees also raised that the recommendation removed complexity and changed how the objects are related. It seems like more things are happening in the recommendation than only removing the code smell.

### 5.3.9 Example 9

The ninth example contains the code smell *large method* and can be seen in Figure 5.32.

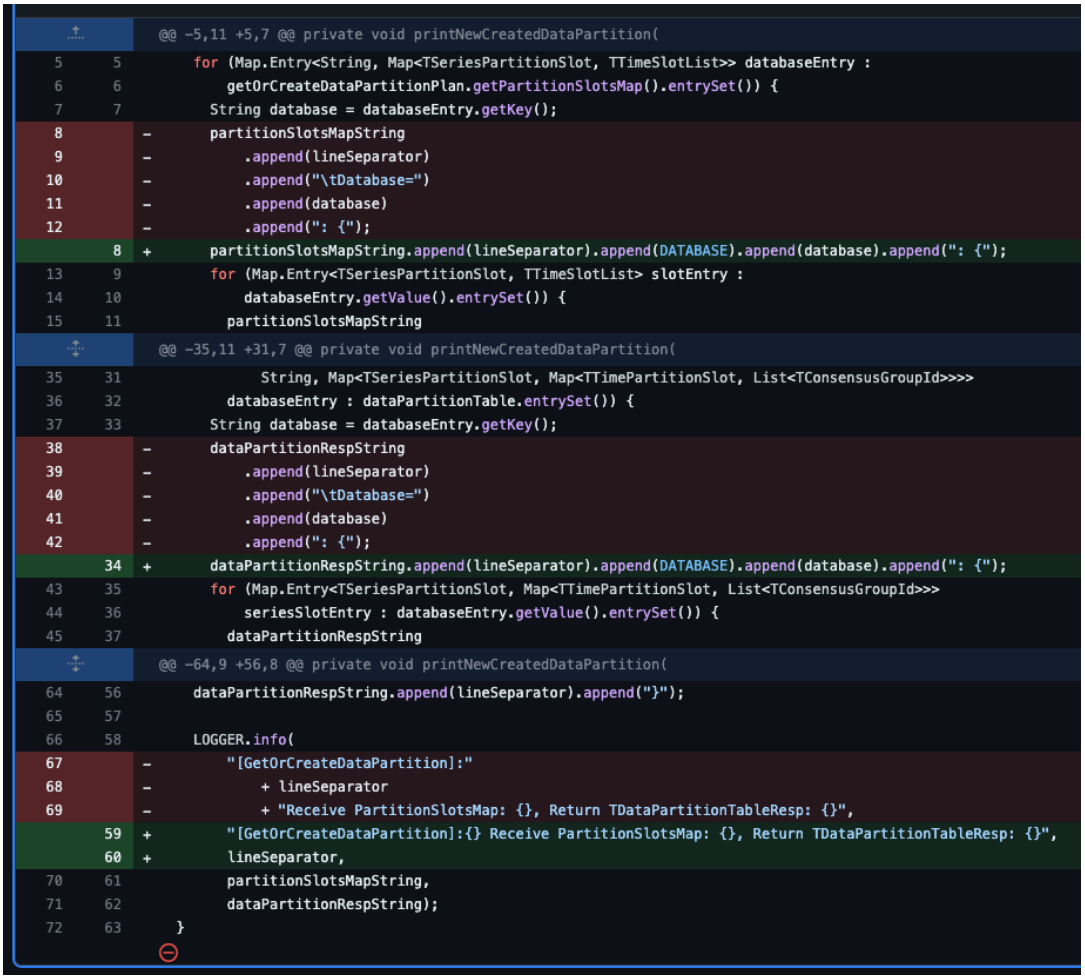
Option A is a close recommendation based on Levenshtein distance and can be seen in Figure 5.33. The Levenshtein distance between this ExRR's pre-method and the example is 2,207 and the code2vec + cosine distance is 0.839. It is ranked as number 10 for Levenshtein distance and 182 for code2vec + cosine distance. The total rank is 895 which represents the number of pairwise combinations between methods within the code smell *large method* that are not originating from the same project.

Option B can be seen in Figure 5.34. It is a close recommendation according to code2vec + cosine distance and has the code2vec + cosine distance 0.698. The Levenshtein distance is 4,405. It is ranked as number 45 for code2vec + cosine distance and 684 for Levenshtein distance.

P1 and P3 expressed that A was a bad recommendation. P1 pointed out that logically

```
public MavenExecutionResult execute( MavenExecutionRequest request )
{
    request.setStartTime( new Date() );

    MavenExecutionResult result = new DefaultMavenExecutionResult();

    Map projectSessions = new HashMap();

    ReactorManager reactorManager = createReactorManager(
        request,
        result,
        projectSessions );

    if ( result.hasExceptions() )
    {
        return result;
    }

    EventDispatcher dispatcher = new DefaultEventDispatcher( request.getEventMonitors() );

    String event = MavenEvents.REACTOR_EXECUTION;

    dispatcher.dispatchStart(
        event,
        request.getBaseDirectory() );

    MavenSession session = createSession(
        request,
        reactorManager,
        dispatcher,
        projectSessions );

    try
    {
        for ( Iterator i = request.getGoals().iterator(); i.hasNext(); )
        {
            String goal = (String) i.next();

            TaskValidationResult tvr = lifecycleExecutor.isTaskValid( goal, session, reactorManager.getTopLevelProject() );

            if ( !tvr.isTaskValid() )
            {
                result.addBuildFailureException( new InvalidTaskException( tvr ) );

                return result;
            }
        }

        getLogger().info( "Scanning for projects..." );

        if ( reactorManager.hasMultipleProjects() )
        {
            getLogger().info( "Reactor build order: " );

            for ( Iterator i = reactorManager.getSortedProjects().iterator(); i.hasNext(); )
            {
                MavenProject project = (MavenProject) i.next();

                getLogger().info( "   " + project.getName() );
            }
        }

        initializeBuildContext( request );

        try
        {
            lifecycleExecutor.execute(
                session,
                reactorManager,
                dispatcher );
        }
        catch ( LifecycleExecutionException e )
        {
            result.addLifecycleExecutionException( e );
            return result;
        }
        catch ( BuildFailureException e )
        {
            result.addBuildFailureException( e );
            return result;
        }

        result.setTopologicallySortedProjects( reactorManager.getSortedProjects() );

        result.setProject( reactorManager.getTopLevelProject() );

        return result;
    }
    finally
    {
        session.dispose();
    }
}
```

**Figure 5.32:** Example 9 in the user interviews.



```
@@ -5,11 +5,7 @@ private void printNewCreatedDataPartition(
    for (Map.Entry<String, Map<TSeriesPartitionSlot, TTimeSlotList>> databaseEntry :
        getOrCreateDataPartitionPlan.getPartitionSlotsMap().entrySet()) {
      String database = databaseEntry.getKey();
-     partitionSlotsMapString
-         .append(lineSeparator)
-         .append("\tDatabase=")
-         .append(database)
-         .append(": {");
+     partitionSlotsMapString.append(lineSeparator).append(DATABASE).append(database).append(": {");
      for (Map.Entry<TSeriesPartitionSlot, TTimeSlotList> slotEntry :
          databaseEntry.getValue().entrySet()) {
        partitionSlotsMapString
@@ -35,11 +31,7 @@ private void printNewCreatedDataPartition(
        databaseEntry : dataPartitionTable.entrySet()) {
      String database = databaseEntry.getKey();
-     dataPartitionRespString
-         .append(lineSeparator)
-         .append("\tDatabase=")
-         .append(database)
-         .append(": {");
+     dataPartitionRespString.append(lineSeparator).append(DATABASE).append(database).append(": {");
      for (Map.Entry<TSeriesPartitionSlot, Map<TTimePartitionSlot, List<TConsensusGroupId>>>
          seriesSlotEntry : databaseEntry.getValue().entrySet()) {
        dataPartitionRespString
@@ -64,9 +56,8 @@ private void printNewCreatedDataPartition(
    dataPartitionRespString.append(lineSeparator).append("}");

    LOGGER.info(
-       "[GetOrCreateDataPartition]:"
-           + lineSeparator
-           + "Receive PartitionSlotsMap: {}, Return TDataPartitionTableResp: {}",
+       "[GetOrCreateDataPartition]:{} Receive PartitionSlotsMap: {}, Return TDataPartitionTableResp: {}",
+       lineSeparator,
        partitionSlotsMapString,
        dataPartitionRespString);
  }
```

**Figure 5.33:** Option A for the ninth example. This was selected as a close recommendation based on Levenshtein distance. The Levenshtein distance is 2,207 and the code2vec + cosine distance is 0.839.

there is no difference in the change. P4 did not think that A says much and thought that A is rather formatting. P2 and P5 were not sure whether A is a refactoring. P5 thought that the change in A leads to almost the same results but liked that the code is in one row instead since it still fits the screen.

> "*I'm not even sure this is refactoring. Just shuffle some things around almost, they made one string into a constant, that is good. Other than that it is pretty minimal.*" - P2 about A in the ninth example

> "*/…I'm not sure if that can be a refactoring example. I'm not sure how is the large method calculated. Is it the same if those are in one line or not? According to this, it's different…/*" - P5 about A in the ninth example

P1, P4 and P5 pointed out that in B several lines were removed. P1 did not think that B was inspirational for removing the code smell except removing code. P2 thought that a lot of context was missing in B and thought that it did not give the developer much information but showed the developer where the refactoring should end up. P3 and P5 were not sure what B does.

```
@@ -6,51 +6,10 @@ private void handleAvro(BuildProducer<ReflectiveClassBuildItem> reflectiveClass,
        // Avro - for both Confluent and Apicurio

        // --- Confluent ---
-       if (QuarkusClassLoader.isClassPresentAtRuntime("io.confluent.kafka.serializers.KafkaAvroDeserializer")) {
-           reflectiveClass
-                   .produce(new ReflectiveClassBuildItem(true, false,
-                           "io.confluent.kafka.serializers.KafkaAvroDeserializer",
-                           "io.confluent.kafka.serializers.KafkaAvroSerializer"));
-
-           reflectiveClass
-                   .produce(new ReflectiveClassBuildItem(true, false, false,
-                           "io.confluent.kafka.serializers.context.NullContextNameStrategy"));
-
-           reflectiveClass
-                   .produce(new ReflectiveClassBuildItem(true, true, false,
-                           "io.confluent.kafka.serializers.subject.TopicNameStrategy",
-                           "io.confluent.kafka.serializers.subject.TopicRecordNameStrategy",
-                           "io.confluent.kafka.serializers.subject.RecordNameStrategy"));
-
-           reflectiveClass
-                   .produce(new ReflectiveClassBuildItem(true, true, false,
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.ErrorMessage",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.Schema",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.Config",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.SchemaReference",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.SchemaString",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.SchemaTypeConverter",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.ServerClusterId",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.SujectVersion"));
-
-           reflectiveClass
-                   .produce(new ReflectiveClassBuildItem(true, true, false,
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.requests.CompatibilityCheckResponse",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.requests.ConfigUpdateRequest",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.requests.ModeGetResponse",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.requests.ModeUpdateRequest",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.requests.RegisterSchemaRequest",
-                           "io.confluent.kafka.schemaregistry.client.rest.entities.requests.RegisterSchemaResponse"));
-       }
-
-       if (QuarkusClassLoader.isClassPresentAtRuntime(
-               "io.confluent.kafka.schemaregistry.client.security.basicauth.BasicAuthCredentialProvider")) {
-           serviceProviders
-                   .produce(new ServiceProviderBuildItem(
-                           "io.confluent.kafka.schemaregistry.client.security.basicauth.BasicAuthCredentialProvider",
-                           "io.confluent.kafka.schemaregistry.client.security.basicauth.SaslBasicAuthCredentialProvider",
-                           "io.confluent.kafka.schemaregistry.client.security.basicauth.UrlBasicAuthCredentialProvider",
-                           "io.confluent.kafka.schemaregistry.client.security.basicauth.UserInfoCredentialProvider"));
+       if (QuarkusClassLoader.isClassPresentAtRuntime("io.confluent.kafka.serializers.KafkaAvroDeserializer")
+               && !capabilities.isPresent(Capability.CONFLUENT_REGISTRY_AVRO)) {
+           throw new RuntimeException(
+                   "Confluent Avro classes detected, please use the quarkus-confluent-registry-avro extension");
        }

        // --- Apicurio Registry 1.x ---
```

**Figure 5.34:** Option B for the ninth example. This was selected as a close recommendation based on code2vec. The code2vec + cosine distance is 0.698 and the Levenshtein distance is 4,405.

> "*There is so much context missing here, like why and how they are able to do this. It doesn't give you much information but it does point you in the right general idea, what the refactoring ultimately should look like. But obviously there is so much stuff going on, code they got cut out, something they must be doing somewhere else, but we don't know why or how. Maybe this would mean something to someone.*" - P2 about B in the ninth example

> "*Ah, it removes a lot of code anyway. I'd like to see some example where it suggests removing a lot of code and replace with a function call. I don't think I saw that in any of these. I wasn't too keen on any of these suggestions actually. If someone would be the least bad, it would probably be B, mostly because it rips out quite a lot of code in any case and maybe collects it somewhere else.*" - P4 about B in the ninth example

P5 preferred A and P1, P2, P3 and P4 preferred B. P3 and P4 pointed out that they did not like neither A nor B. P3 would like to have better examples and P4 would like to have an example where a substantial chunk of code was extracted and replaced with a function call.

In this example, four interviewees preferred B. Those four interviewees thought that A was a bad recommendation. We interpreted that the interviewees also thought that B was bad.

### 5.3.10 Example 10

The tenth example also contains the code smell *large method* and can be seen in Figure 5.35.

Option A can be seen in Figure 5.36. It is a close recommendation according to code2vec + cosine distance and has the code2vec + cosine distance 0.771. The Levenshtein distance is 3,100. It is ranked as number 82 for code2vec + cosine distance and 215 for Levenshtein distance.

Option B is a close recommendation based on Levenshtein distance and can be seen in Figure 5.37. The Levenshtein distance between this ExRR's pre-method and the example is 2,913 and the code2vec + cosine distance is 0.956. It is ranked as number 163 for Levenshtein distance and 488 for code2vec + cosine distance.

P1 did not think that A was helpful and pointed out that it looks like the code only has been removed. P2 thought that A is quite detailed and a good recommendation but that it does not match the example. P3 liked the fact that code had been moved elsewhere which resulted in a shorter method and thought that A maybe could be used as inspiration. P4 liked the thought of A and thought it is positive that code has been removed. However, P4 pointed out that it looks like functionality also has been removed which you do not want when you refactor. P5 did not understand why A is an example of how to solve the code smell *large method*.

> "*Here, it looks like you just removed something completely, doesn't help much.*" - P1 about A in the tenth example

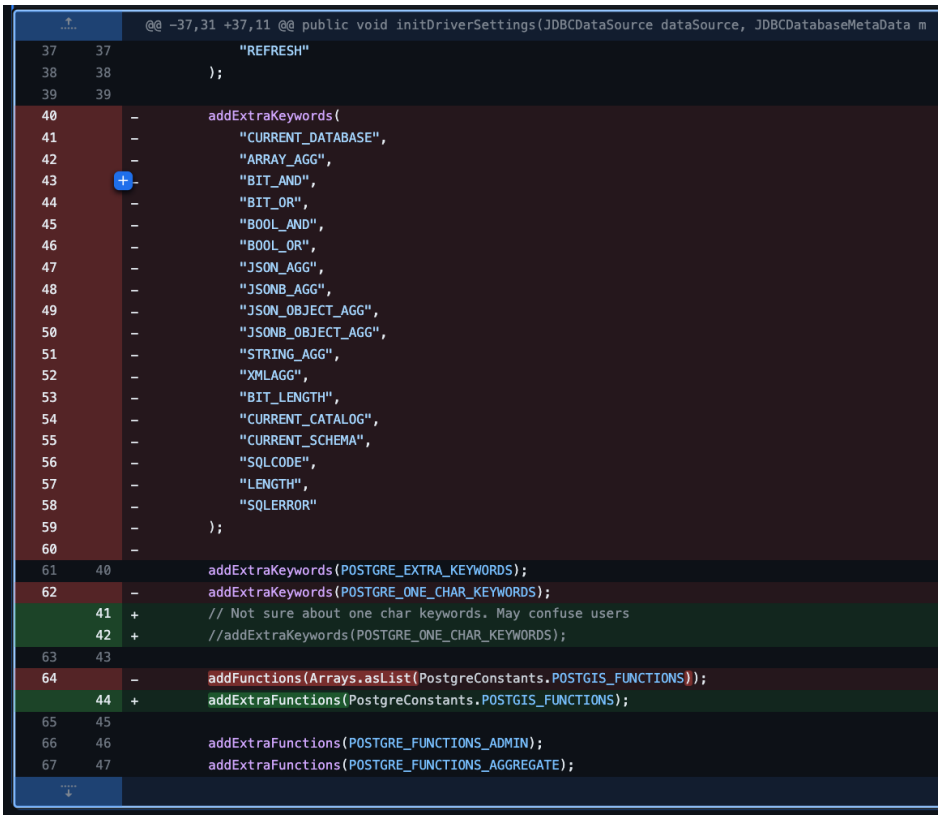
> "*I like the idea of this. A bit hard to follow, what is addExtraKeywords? I don't know, maybe I'm trying to see too much of the details in it. But it feels like in the example, functionality has also been removed. Maybe it was okay just when you did this but, not the way you want to think when you sit and refactor yourself. Then I think that then you have to keep exactly the same functionality. But still, it removes quite a lot of code in any case so it has it going for it in any case.*" - P4 about A in the tenth example

P1 thought that B is better since you could get inspired to extract a function. P2 also stated that B looks like some kind of extract method and that B is quite a good recommendation for the example. P3 and P4 also pointed out that code is replaced with a function call and explained that B is more applicable to the example.

> "*This one made more sense. Yes, cut out a lot of code and replace with function calls. I like that. Spontaneously, stuff like I'd like to do with that method on the left (the example). In this case I would definitely say B.*" - P4 about B in the tenth example

```
public void invokesSchedulerCoreForDeploymentMethod() throws Exception {
  EnvironmentId environmentId =
      EnvironmentId.builder()
          .accountId(ACCOUNT_ID)
          .cluster(CLUSTER_NAME)
          .environmentName(ENVIRONMENT_NAME)
          .build();

  DescribeEnvironmentRequest describeEnvironmentRequest =
      DescribeEnvironmentRequest.builder().environmentId(environmentId).build();
  DescribeEnvironmentRevisionRequest describeEnvironmentRevisionRequest =
      DescribeEnvironmentRevisionRequest.builder()
          .environmentId(environmentId)
          .environmentRevisionId(ACTIVE_ENVIRONMENT_REVISION_ID)
          .build();

  when(dataService.describeEnvironment(describeEnvironmentRequest))
      .thenReturn(
          DescribeEnvironmentResponse.builder()
              .environment(
                  Environment.builder()
                      .environmentId(environmentId)
                      .role("")
                      .environmentType(EnvironmentType.SingleTask)
                      .createdTime(Instant.now())
                      .lastUpdatedTime(Instant.now())
                      .environmentHealth(EnvironmentHealth.HEALTHY)
                      .environmentStatus(EnvironmentStatus.ACTIVE)
                      .deploymentMethod(DEPLOYMENT_METHOD)
                      .deploymentConfiguration(DeploymentConfiguration.builder().build())
                      .activeEnvironmentRevisionId(ACTIVE_ENVIRONMENT_REVISION_ID)
                      .build())
              .build());
  when(dataService.describeEnvironmentRevision(describeEnvironmentRevisionRequest))
      .thenReturn(
          DescribeEnvironmentRevisionResponse.builder()
              .environmentRevision(
                  EnvironmentRevision.builder()
                      .environmentId(environmentId)
                      .environmentRevisionId(ACTIVE_ENVIRONMENT_REVISION_ID)
                      .taskDefinition(TASK_DEFINITION)
                      .createdTime(Instant.now())
                      .build())
              .build());

  ClusterSnapshot snapshot =
      new ClusterSnapshot(CLUSTER_NAME, Collections.emptyList(), Collections.emptyList());

  SchedulingAction successfulAction = e -> CompletableFuture.completedFuture(true);
  SchedulingAction failedAction = e -> CompletableFuture.completedFuture(false);

  Scheduler mockScheduler = mock(Scheduler.class);
  when(mockScheduler.schedule(any(), any()))
      .thenReturn(Arrays.asList(successfulAction, failedAction));

  when(schedulerFactory.schedulerFor(any())).thenReturn(mockScheduler);

  SchedulerHandler handler = new SchedulerHandler(dataService, ecs, schedulerFactory);

  SchedulerOutput output =
      handler.handleRequest(new SchedulerInput(snapshot, environmentId), null);

  verify(dataService).describeEnvironment(describeEnvironmentRequest);
  verify(dataService).describeEnvironmentRevision(describeEnvironmentRevisionRequest);
  ArgumentCaptor<EnvironmentDescription> environmentDescription =
      ArgumentCaptor.forClass(EnvironmentDescription.class);
  verify(mockScheduler).schedule(eq(snapshot), environmentDescription.capture());
  assertThat(environmentDescription.getValue())
      .isEqualTo(
          EnvironmentDescription.builder()
              .clusterName(CLUSTER_NAME)
              .environmentName(ENVIRONMENT_NAME)
              .activeEnvironmentRevisionId(ACTIVE_ENVIRONMENT_REVISION_ID)
              .environmentType(EnvironmentDescription.EnvironmentType.SingleTask)
              .taskDefinitionArn(TASK_DEFINITION)
              .deploymentMethod(DEPLOYMENT_METHOD)
              .build());

  assertThat(output.getClusterName()).isEqualTo(CLUSTER_NAME);
  assertThat(output.getEnvironmentId()).isEqualTo(environmentId);
  assertThat(output.getSuccessfulActions()).isEqualTo(1L);
  assertThat(output.getFailedActions()).isEqualTo(1L);
}
```

**Figure 5.35:** Example 10 in the user interviews.



```
@@ -37,31 +37,11 @@ public void initDriverSettings(JDBCDataSource dataSource, JDBCDatabaseMetaData m
            "REFRESH"
        );

-       addExtraKeywords(
-           "CURRENT_DATABASE",
-           "ARRAY_AGG",
-           "BIT_AND",
-           "BIT_OR",
-           "BOOL_AND",
-           "BOOL_OR",
-           "JSON_AGG",
-           "JSONB_AGG",
-           "JSON_OBJECT_AGG",
-           "JSONB_OBJECT_AGG",
-           "STRING_AGG",
-           "XMLAGG",
-           "BIT_LENGTH",
-           "CURRENT_CATALOG",
-           "CURRENT_SCHEMA",
-           "SQLCODE",
-           "LENGTH",
-           "SQLERROR"
-       );
-
        addExtraKeywords(POSTGRE_EXTRA_KEYWORDS);
-       addExtraKeywords(POSTGRE_ONE_CHAR_KEYWORDS);
+       // Not sure about one char keywords. May confuse users
+       //addExtraKeywords(POSTGRE_ONE_CHAR_KEYWORDS);

-       addFunctions(Arrays.asList(PostgreConstants.POSTGIS_FUNCTIONS));
+       addExtraFunctions(PostgreConstants.POSTGIS_FUNCTIONS);

        addExtraFunctions(POSTGRE_FUNCTIONS_ADMIN);
        addExtraFunctions(POSTGRE_FUNCTIONS_AGGREGATE);
```

**Figure 5.36:** Option A for the tenth example. This was selected as a close recommendation based on code2vec. The code2vec + cosine distance is 0.771 and the Levenshtein distance is 3,100.

```
@@ -24,25 +24,8 @@ public void loadSnapshot(File latestSnapshotRootDir) {
24  24                storageGroupFullPath,
25  25                schemaRegionId.getId(),
26  26                regionStatistics,
27    -               measurementMNode -> {
28    -                 regionStatistics.addTimeseries(1L);
29    -                 if (measurementMNode.getOffset() == -1) {
30    -                   return;
31    -                 }
32    -                 try {
33    -                   tagManager.recoverIndex(measurementMNode.getOffset(), measurementMNode);
34    -                 } catch (IOException e) {
35    -                   logger.error(
36    -                       "Failed to recover tagIndex for {} in schemaRegion {}.",
37    -                       storageGroupFullPath + PATH_SEPARATOR + measurementMNode.getFullPath(),
38    -                       schemaRegionId);
39    -                 }
40    -               },
41    -               deviceMNode -> {
42    -                 if (deviceMNode.getSchemaTemplateIdWithState() >= 0) {
43    -                   regionStatistics.activateTemplate(deviceMNode.getSchemaTemplateId());
44    -                 }
45    -               },
      27  +           measurementInitProcess(),
      28  +           deviceInitProcess(),
46  29                tagManager::readTags,
47  30                this::flushCallback);
48  31        logger.info(
```

**Figure 5.37:** Option B for the tenth example. This was selected as a close recommendation based on Levenshtein distance. The Levenshtein distance is 2,913 and the code2vec + cosine distance is 0.956.

> *"/...I prefer this example in that case even though it is a bit different. It's like that with almost all of these, yes, it must be a bit of inspiration, then you have to take some more steps on how to use it."* - P3 about B in the tenth example

Everyone preferred B.

In this example, everyone preferred B. In A, three interviewees talked about that code had been removed, one of them that it was moved elsewhere, but it differed if they thought the recommendation was helpful or not. When it comes to B, everyone talked about that code was extracted which they were positive about.

### 5.3.11 Summary of the Examples

In this section, we will go through a summary of the results from the examples. An overview can be seen in Table 5.4. The first column shows which example the information relates to and the second what code smell the example contains. The columns *P1-P5* contains the interviewees answers for every example. The columns *Sum of Lev* and *Sum of C2v+cos* contains the total number of interviewees that thought the close recommendation according to Levenshtein distance respectively code2vec + cosine distance was most useful. In the bottom of the table, this is also summarised for all examples. *Lev rank (dist)* and *c2v+cos rank (dist)* consists of the two ExRRs for the example, referred to *A)* respectively *B)* according to the designation in Section 5.3. The first number is the rank and the number in parenthesis is the distance. The

ExRR that is marked in bold is the one that is selected as a close recommendation according to the current measure.

For four of the ten examples, highlighted in bold font, all interviewees agreed on which of the two recommendations that was the most useful. For three examples, everyone apart from one interviewee agreed on which example was the most useful. Therefore, the interviewees agreed or almost agreed in seven of ten examples. Two interviewees (P3 and P4) agreed on which recommendation was the most useful for each example.

There is no indication that one of the measures is better than the other. The Levenshtein distance recommendations received 27 votes and the code2vec + cosine distance recommendations received 23 votes. For both of the examples with the code smell *complex method* (examples 3 and 4), everyone chose the recommendations that were closest according to code2vec + cosine distance. This could mean that this distance measure is suitable for the code smell *complex method*, but further investigations are needed to confirm this. There were two more examples that everyone agreed on, one with the code smell *deep, nested complexity* (example 6) and one with the code smell *large method* (example 10). For those, the recommendations were closest according to Levenshtein distance.

There is no indication that a recommendation is better simply because it is close to the example according to both measures, neither in general nor for specific code smells. If we are looking in general, ExRR B for example 2 and ExRR A for example 3 could be considered as close to the example according to both code2vec + cosine distance and Levenshtein distance. For example 3, everyone chose A but for example 2, only one interviewee chose B.

If we are looking per code smell, we can look at the examples with *complex method* again (example 3 and 4). Here everyone agreed on A in the third example and B in the fourth example. As stated before, ExRR A for example 3 is close according to both measures, but that is not the case for ExRR B in example 4.

When looking at the ranking, the code2vec + cosine ranking is in general lower than the ranking for Levenshtein distance. This is because there are more methods that are considered as close for Levenshtein distance than for code2vec + cosine distance as can be seen in Figure 5.5 and Figure 5.7.

### 5.3.12 What Is a Good Refactoring Recommendation?

In the end of the interviews, we asked the interviewees what they thought characterised a good refactoring recommendation.

P2 shared that the challenge when making refactoring recommendations is to present enough context. P2 pointed out that there are things happening outside of the snippet that you cannot see and gave the example that when being presented an *extract method* refactoring, you do not know if the method is doing what it is supposed to do. Further, P2 thought that some blanks could be filled in if you are familiar with the context or the code base. P2 raised a fear of breaking things that already worked and thought that refactoring recommendations could help with that fear in the sense that they show that someone else has done a similar refactoring in the past. P2 also stated that seeing the recommendations as inspirational could be more important than we tend to think.

**Table 5.4:** An overview of the result from from the examples.

| Ex | Code Smell | P1 | P2 | P3 | P4 | P5 | Sum Lev | Sum C2v+cos | Lev Rank (Dist.) | C2v+cos Rank (Dist.) |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Bumpy | A | B | A | A | B | 3 | 2 | **A) 14,967 (1,084)** B) 35,687 (1,497) | A) 25,083 (0.885) **B) 200 (0.594)** |
| 2 | Bumpy | A | B | A | A | A | 4 | 1 | **A) 16,181 (1,113)** B) 18,945 (1,178) | A) 62,033 (0.992) **B) 26 (0.424)** |
| **3** | Complex | A | A | A | A | A | 0 | 5 | A) 22,347 (988) **B) 22,507 (990)** | **A) 481 (0.524)** B) 26,665 (0.852) |
| **4** | Complex | B | B | B | B | B | 0 | 5 | **A) 19,371 (947)** B) 58,342 (1,405) | A) 66,045 (0.930) **B) 420 (0.474)** |
| 5 | Deep | A | A | B | B | B | 2 | 3 | **A) 5,645 (1,021)** B) 9,887 (1,222) | A) 39,557 (0.983) **B) 5 (0.441)** |
| **6** | Deep | B | B | B | B | B | 5 | 0 | A) 9,819 (1,219) **B) 6,719 (1,076)** | **A) 49 (0.560)** B) 27,668 (0.933) |
| 7 | Excess | A | B | A | A | B | 3 | 2 | **A) 7,591 (986)** B) 18,849 (4,283) | A) 1,153 (0.727) **B) 54 (0.595)** |
| 8 | Excess | A | B | B | B | B | 4 | 1 | A) 17,134 (2,647) **B) 6,831 (921)** | **A) 44 (0.591)** B) 8,646 (0.870) |
| 9 | Large | B | B | B | B | A | 1 | 4 | **A) 10 (2,207)** B) 684 (4,405) | A) 182 (0.839) **B) 45 (0.698)** |
| **10** | Large | B | B | B | B | B | 5 | 0 | 215 (3,100) **B) 163 (2,913)** | **A) 82 (0.771)** B) 488 (0.956) |
| All | | | | | | | 27 | 23 | | |

> "*I think the challenge with refactoring recommendation is that it's hard to provide enough context.../*" - P2

P3 expressed that a good refactoring recommendation changes and simplifies the code without changing the behaviour of the code. Further, P3 expressed that the recommendation should be solving one small problem that is not dependent on too much refactoring outside of the snippet that is shown.

> "*Find examples that restructure, simplify the code without changing its behaviour. Ideally, it should solve a specific problem..../*" - P3

P4 thought that a good refactoring recommendation should be short and concise to help the developers get an overview of what they are recommended to do.

> "*Short and concise example. Because if you get too much, it becomes so difficult to get an overview of what you are recommended to do. So that these ones, which were a little shorter, I clearly liked better, when they were relevant of course.*" - P4

P5 also thought that a good recommendation should be short and concise and that it should be easy to follow what has been done to solve the code smell so the developer does not have to analyse it too much.

> "*/...So short and concise and readable.*" - P5

When we asked if there was anything that the interviewees wanted to add, P3 raised a wish to get a recommendation that fits the context. P3 thought that in test code, the desired refactorings could differ. Further, P3 thought that it would be good to not only know that it is a *bumpy road ahead* but also what was done to create that smell. As an example, P3 came up with the example of having `try...catch` blocks and for loops mixed in a method and thought that the recommendation could be to separate for loops from `try...catch` blocks. However, P3 pointed out that this would be harder to accomplish.

During the interview P3 also expressed that the task of choosing which refactoring recommendation was most helpful was hard since P3 did not feel the need for recommendations. Further, P3 said that as soon as the example was shown, P3 began to think about approximately how P3 hoped the recommendations would look like.

# Chapter 6
# Discussion

In this chapter, we will discuss the findings from the result in Chapter 5. This will be done in relation to the research questions.

## 6.1 RQ1 – How Well Do Similarity Measures Work in the Context of CodeScene's Refactoring Recommendations?

In this section, we will discuss how the similarity measures performed together and individually.

There were indications that similarity-based recommendations could be useful. Among the similarity-based recommendations, there were recommendations that the interviewees thought were good. Since the interviewees had to choose between two recommendations, it is not always clear if they actually thought that the recommendation was useful or if it only was considered as better than the other. However, there were examples where the interviewees expressed that they thought that the recommendation was good. Every recommendation was close to the examples according to at least one of the measures (except for maybe the examples with *large method* where the recommendations were not that close since the sample data only contained 44 methods with this code smell). Thus, we conclude that there were useful input to developers among recommendations that were close to an example according to the similarity measures.

The question is whether there also would have been recommendations that the interviewees thought were good if all the recommendations had been far from the examples. By looking at methods that are similar according to both similarity measures, there is no indication that the similar ExRRs are better. A few times, the recommendation was mentioned as similar to the example. Every time an interviewee expressed that a recommendation was similar to the example, the interviewee also chose that recommendation. If it was classified as

dissimilar, the recommendation was not chosen. This further indicates that similarity-based recommendations could be useful.

When looking at the similarity measures individually, generally Levenshtein distance and code2vec + cosine distance interpret different methods as similar but there is no indication that one of them is better than the other. In the pilot experiment and the similarity calculations, Levenshtein distance and code2vec + cosine distance had a very weak correlation (Table 5.2) respectively no correlation (Table 5.2). The distributions from the similarity calculations are positively respectively negatively skewed. Therefore, one might wonder whether the measures interpret similarity differently. If one of the measures would be better than the others is hard to say. The Levenshtein distance and code2vec + cosine distance almost got the same votes by the interviewees in the user study.

In our small user study, code2vec is not as successful for refactoring the code smells *bumpy road ahead*, *complex method*, *deep, nested complexity*, *excess number of function arguments* and *large method* as for move method refactoring [23]. However, further research has to be conducted to verify this finding.

## 6.2 RQ2 – How Do Senior Developers Perceive Similarity-Based Refactoring Recommendations?

In this section, we will go through differences in how the developers interpreted the examples and present our high-level themes.

From a top-down perspective, the interviewees mostly agreed, but their reasoning differed considerably. When analysing their answers on a word-by-word level, it did not seem like they agreed *that* much. The interviewees interpreted several recommendations differently. For example, in some cases when code was removed, some people interpreted that the code was only removed and some that it was moved somewhere else. Sometimes, the interviewees focus on different things that happened in the code. For example, for ExRR B in example 2, one interviewee focused on the similarity of the loop structures while one focused on the functionality change. The interviewees also saw the recommendations from different perspectives. For example, one interviewee pointed out several times that the refactoring recommendation probably was not enough for a novice developer, while some interviewees already before the recommendations expressed what kind of recommendation they hoped would come.

Based on the coding, we identified four high-level themes that can be seen in Table 6.1. Those were 1) **sentiment/usefulness**, 2) **motivations why bad**, 3) **case-specific presentation** and 4) **"textbook" refactorings**. The sentiment/usefulness includes the codes a) good recommendation, b) inspirational/some value and c) trivial/no value/irrelevant. It provides a scale for how good a recommendation is, from good to useless where inspirational recommendations end up somewhere in between.

The interviewees often reasoned about the completeness of the recommendation and how much information they thought it gave the developer. The interviewees identified recommendations where the recommendation gave the developer a lot of information. In those recommendations, they thought the developer was guided through the whole process. The

interviewees also identified recommendations with little information. Those could be giving information in a way that helped the developer getting started with the refactoring or those could be an example of the end goal of the refactoring. A recommendation could also be giving so little information that the interviewees thought it was useless. Sometimes, interviewees also said that they did not understand the recommendation. Lack of information or a good amount of information could explain where a recommendation ends up on the scale that this theme provides, but there were other explanations too.

**Table 6.1:** The four high-level themes and their corresponding low-level codes.

| High-level themes | Codes |
|---|---|
| **1. Sentiment/usefulness** | a) Good recommendation<br>b) Inspirational/some value<br>c) Trivial/no value/irrelevant |
| **2. Motivations why bad** | a) Missing context/unclear consequences or actions<br>b) Functionality change<br>c) Not applicable here |
| **3. Case-specific presentation** | a) Size of recommendation/presentation<br>b) Concise and easy to comprehend<br>d) Similarity |
| **4. "Textbook" refactorings** | a) Function extraction<br>b) Structural reasoning/control flow<br>c) Parameter object |

The theme **motivations why bad** includes the codes a) missing context/unclear consequences or actions, b)functionality change and c) not applicable here. It provides explanations for why a recommendation is not useful. The case could for example be that unclear things were going on in the code or that the code or change missed context. As for too little information, this may also be a source of not understanding the recommendation. Other things mentioned was that the functionality was changed and therefore interviewees thought that the recommendation did not fit as a refactoring recommendation. The interviewees also raised that a recommendation did not fit the current example as a reason for why it was not useful.

**Case-specific presentation** includes a) size of recommendation/presentation, b) concise and easy to comprehend and c) similarity. The size of the recommendation or the presentation could also be a source of not understanding or thinking that a recommendation is useful, leading us to the theme case-specific presentation. Two interviewees thought that the diff view in ExRR B for example 6 made the recommendation hard to understand. Further, the interviewees did not like large recommendations, while smaller recommendations were praised for being small. Clear recommendations that did not contain more changes than fixing the smell were also considered good.

**"Textbook" refactorings** includes the codes a) function extraction, b) structural reasoning/control flow and c) parameter object. For this theme, there were also things that were highlighted as good. A recurring type of refactoring that was frequently mentioned was function extraction. This was mentioned for every code smell, except for *the excess number of function arguemnts*, and it was mentioned as a positive thing. Either the recommendation was

praised for extracting code or a wish was expressed that the recommendation should have contained such an operation. Parameter object was another type of refactoring, linked to the "textbook" refactorings. This was frequently mentioned among the two examples with the code smell *excess number of function arguments* but not for other code smells.

What was also recurring was an expression of what kind of recommendation the interviewees expected or what kind of recommendation they would have preferred instead. It seemed like some of the interviewees did not need recommendations since when they looked at the examples, they started to talk about how they would refactor the example and what recommendations they hoped would come. This was more common for some interviewees. It is possible that the other interviewees thought that the recommendations were more useful or it is linked to that they had another perspective, as discussed above.

## 6.3 Limitations

A limitation of the work is the size of the data sample. With a bigger sample, there would be greater opportunities for investigating the similarity measures. The probability would be higher to get methods that are more similar to each other according to code2vec + cosine distance and Levenshtein distance. Especially for *large methods*, we had to choose ExRRs with greater distances to the examples than the other code smells due to a small data sample. The probability would also be higher to get different combinations that one can consciously explore. One combination could for example be an ExRR that is very close to an example according to both measures. Another combination could be that the ExRR is close to the example according to one measure but not the other. A third combination could be that the ExRR is far away according to both measures. You can also ask yourself if our methods in the evaluation are close enough to be able to determine if the measures are helpful?

Another limitation is that we check the distance between the example and the pre-method for the ExRR, but then we evaluate the refactoring. The similarity measure is thus linked to how the method looked before it was refactored. The refactorings can be more or less well implemented, which affects the result.

The thesis only investigates a limited number of similarity measures. Therefore, other measures could have given another result. All methods are in Java. The result could be different for other languages. The experience of the developers is somehow spread but they are still considered as a group of senior developers. We do not know if novice developers would interpret the recommendations the same way. The developers also work on the same company and could therefore have common ideas about refactoring recommendations. Furthermore, the developer work on a tool for improving code comprehension and therefore they could be more aware of refectorings than senior developers from other companies.

# Chapter 7
# Conclusions

The purpose of the thesis was to evaluate how existing code similarity measures could be used to identify similar methods from open-source software projects for refactoring recommendations. Also, to explore a potential way forward for CodeScene's refactoring recommendations. To do this, different similarity measures were identified in the literature and evaluated on manually created example methods and methods from top-starred Java projects on GitHub. For the manually created methods, the measures were evaluated with our intuitive understanding (Figure 4.2) and for methods from Java methods, the measures were evaluated qualitatively with senior developers from CodeScene.

The qualitative evaluation also examined how the developers interpret refactoring recommendations. This showed that the developers preferred recommendations that are short, clear and easy to comprehend. The interviewees also wished for recommendations that only solve the current code smell and do not contain other changes. As few changes as possible should occur outside of the recommendation shown – the changes should ideally be self-contained. The similarity measures encode different aspects of similarity but there is no indication that one of them is better than the other for refactoring recommendations. On the other hand, there were indications that similarity-based recommendations could be useful. To get a better picture, the area needs to be investigated further.

## 7.1 Future Work

As a first step, we suggest to interview more developers to see if new things arise. It would also be interesting to examine how novice developers interpret refactoring recommendations since they are probably the group most in need of this type of support.

For better understanding if the similarity measures are helpful for refactoring recommendations, a quantitative analysis could be conducted to examine whether a pattern emerges for recommendations considered as similar versus recommendations considered as dissimilar. This study should include more developers, automated data collection using a dedicated

tool, a bigger sample of methods and evaluate more examples.

The rapid development of artificial intelligence could lead to AI taking over a substantial part of refactoring. Early results are promising [30] [42]. At first, AI-based refactoring will be for small local code smells which means that the human will be needed for more bigger complex refactorings. However, this focus is in line with what our interviewees prefer – recommendations that are small and local. From this point of view, our findings encourage more work on AI-based refactorings.

# References


[1] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ó Cinnéide, Ali Ouni, and Yuanfang Cai. An Interactive and Dynamic Search-Based Approach to Software Refactoring Recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961, September 2020. Conference Name: IEEE Transactions on Software Engineering.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40:1–40:29, January 2019.

[3] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, November 1998. ISSN: 1063-6773.

[4] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, September 2007. Conference Name: IEEE Transactions on Software Engineering.

[5] c2nes. c2nes/javalang: Pure python java parser and tools. Available at `https://github.com/c2nes/javalang` (2023/10/09).

[6] CodeScene. Code health – how easy is your code to maintain and evolve? Available at `https://codescene.io/docs/guides/technical/code-health.html` (2023/06/15).

[7] CodeScene. Codescene documentation - getting started. Available at `https://codescene.io/docs/getting-started/index.html` (2023/04/20).

[8] CodeScene. Codescene documentation - refactoring recommendations. Available at `https://codescene.io/docs/guides/technical/refactoring-recommendations.html` (2023/04/20).

[9] CodeScene. Codescene's code health metric. Available at `https://codescene.com/code-health` (2023/05/03).

[10] CodeScene. From code to delivery: The 4 factors model. Available at `https://codescene.io/docs/4f-dashboard/4-factors-dashboard.html` (2023/05/19).

[11] CodeScene. Virtual code review. Available at `https://codescene.io/projects/1690/jobs/231817/results/code/hotspots/biomarkers?name=Mvc%2Fsrc%2FMicrosoft.AspNetCore.Mvc.ViewFeatures%2FViewFeatures%2FHtmlHelper.cs` (2023/10/09).

[12] Codescene-research. Similarity-refactorings. Available at `https://github.com/codescene-research/similarity-refactorings` (2023/11/01).

[13] Daniela S. Cruzes and Tore Dyba. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 275–284, September 2011. ISSN: 1949-3789.

[14] Fowler M. Refactoring. Addison-Wesley Professional, 1999.

[15] Jyotirmoy Gope and Sanjay Kumar Jain. A survey on solving cold start problem in recommender systems. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 133–138, May 2017.

[16] Halstead, Maurice H. Elements of software science (operating and programming systems series). Elsevier Science Inc, 1977.

[17] jplag. jplag/jplag: Token-based software plagiarism detection. Available at `https://github.com/jplag/JPlag` (2023/06/28).

[18] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002. Conference Name: IEEE Transactions on Software Engineering.

[19] Cory Kapser and Michael W Godfrey. Toward a Taxonomy of Clones in Source Code: A Case Study. *Evolution of large scale industrial software architectures*, 16:107–113, 2003.

[20] Sašo Karakatič, Aleksej Miloševič, and Tjaša Heričko. Software system comparison with semantic source code embeddings. *Empirical Software Engineering*, 27(3):70, March 2022.

[21] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In Patrick Cousot, editor, *Static Analysis*, Lecture Notes in Computer Science, pages 40–56, Berlin, Heidelberg, 2001. Springer.

[22] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309, October 2001. ISSN: 1095-1350.

[23] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. Recommendation of Move Method Refactoring Using Path-Based Representation of Code. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, pages 315–322, New York, NY, USA, September 2020. Association for Computing Machinery.

[24] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976. Conference Name: IEEE Transactions on Software Engineering.

[25] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004. Conference Name: IEEE Transactions on Software Engineering.

[26] Hou Min and Zhang Li Ping. Survey on Software Clone Detection Research. In *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*, ICMSS 2019, pages 9–16, New York, NY, USA, January 2019. Association for Computing Machinery.

[27] K. J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin*, 8(4):30–41, December 1976.

[28] Ali Ouni, Marouane Kessentini, Mel Ó Cinnéide, Houari Sahraoui, Kalyanmoy Deb, and Katsuro Inoue. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process*, 29(5):e1843, 2017. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1843.

[29] P. Jaccard. Distribution de la flore alpine dans le bassin des dranses et dans quelques r´egions voisines. In *Bulletin de la Soci´et´e vaudoise des sciences naturelles*, page 37:241–272, 1901.

[30] Russell A. Poldrack, Thomas Lu, and Gašper Beguš. AI-assisted coding: Experiments with GPT-4, April 2023. arXiv:2304.13187 [cs].

[31] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding Plagiarisms among a Set of Programs with JPlag. 2002.

[32] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. A comparison of code similarity analysers. *Empirical Software Engineering*, 23(4):2464–2519, August 2018.

[33] Ghulam Rasool and Zeeshan Arshad. A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895, 2015. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1737.

[34] Robertfeldt. Robertfeldt/multidistances.jl: Collects many string and object distances (incl compression and set distances). Available at `https://github.com/robertfeldt/MultiDistances.jl/tree/master` (2023/06/28).

[35] Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation Systems for Software Engineering. *IEEE Software*, 27(4):80–86, July 2010. Conference Name: IEEE Software.

[36] Chanchal K. Roy and James R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, June 2008. ISSN: 1092-8138.

[37] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1157–1168, New York, NY, USA, May 2016. Association for Computing Machinery.

[38] Tech-Srl. Tech-srl/code2vec: Tensorflow code for the neural network presented in the paper: "code2vec: Learning distributed representations of code". Available at `https://github.com/tech-srl/code2vec` (2023/06/12).

[39] tsantalis. tsantalis/jdeodorant: Jdeodorant. Available at `https://github.com/tsantalis/JDeodorant` (2023/10/19).

[40] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, May 2009. Conference Name: IEEE Transactions on Software Engineering.

[41] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics-doklady*, volume 10, page 707–710, 1966.

[42] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design, March 2023. arXiv:2303.07839 [cs].

[43] Wikipedia. Jaccard index. Available at `https://en.wikipedia.org/wiki/Jaccard_index#cite_note-3` (2023/06/29).

[44] Michael Wise. String Similarity via Greedy String Tiling and Running KarpRabin Matching. *Unpublished Basser Department of Computer Science Report*, January 1993.

[45] Michael D. Wolcott and Nikki G. Lobczowski. Using cognitive interviews and think-aloud protocols to understand thought processes. *Currents in Pharmacy Teaching and Learning*, 13(2):181–188, February 2021.

[46] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study. *ACM Computing Surveys*, October 2023. Just Accepted.

[47] Feng Zhang, Lulu Li, Cong Liu, and Qingtian Zeng. Flow Chart Generation-Based Source Code Similarity Detection Using Process Mining. *Scientific Programming*, 2020:e8865413, July 2020. Publisher: Hindawi.

# Appendices

# Appendix A

# Manually Created Example Methods for Pilot Experiment

**Listing A.1:** M1: addition

```
private int addition(int x, int y){
    return x + y;
}
```

**Listing A.2:** M2: addElseIfStatement

```
private int addition(int x, int y) {
  if (x < 1) {
    System.out.println("Hello");
  } else {
    System.out.println("Hi");
  }
  return x + y;
}
```

**Listing A.3:** M3: addFiveLines

```
private int addition(int x, int y) {
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    return x + y;
}
```

**Listing A.4:** M4: addForLoop

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
      }
    return x + y;
}
```

**Listing A.5:** M5: addForLoopV2

```
private int addition(int x, int y){
    for (int i = 0; x < y; i++) {
        System.out.println(i);
      }
    return x + y;
}
```

**Listing A.6:** M6: addIfStatement

```
private int addition(int x, int y) {
  if (x < 1) {
    System.out.println("Hello");
  }
  return x + y;
}
```

**Listing A.7:** M7: addIfStatementV2

```
private int addition(int x, int y) {
  if (x != y) {
    System.out.println("Hello");
  }
  return x + y;
}
```

**Listing A.8:** M8: addParameter

```
private int addition(int x, int y, int z){
    return x + y + z;
}
```

**Listing A.9:** M9: addSwitch

```
private int addition(int x, int y) {
  switch (x) {
    case 1:
      System.out.println("Hello");
      break;
    case 2:
      System.out.println("Hi");
```

```
      break;
  }

  return x + y;
}
```

**Listing A.10:** M10: addTenLines

```
private int addition(int x, int y) {
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    System.out.println("Hello");
    return x + y;
}
```

**Listing A.11:** M11: addTryCatch

```
private int addition(int x, int y){
    try {
        x = x/y;
    } catch(Exception e) {
        System.out.println("Error");
    }

    return x + y;
}
```

**Listing A.12:** M12: addWhileLoop

```
private int addition(int x, int y){
    int i = 0;
    while (i < x) {
      System.out.println(i);
      i++;
    }
    return x + y;
}
```

**Listing A.13:** M13: deleteParameter

```
private int addition(int x){
    return x;
}
```

**Listing A.14:** M14: nestedForLoop

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            System.out.println(i);
        }
    }
    return x + y;
}
```

**Listing A.15:** M15: renameMethod

```
private int method(int x, int y) {
  return x + y;
}
```

**Listing A.16:** M16: renameParameter

```
private int addition(int nbr, int y) {
  return nbr + y;
}
```

**Listing A.17:** M17: addForLoop+IfStatement

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
      }

    if (x < 1) {
        System.out.println("Hello");
    }
    return x + y;
}
```

**Listing A.18:** M18: addForLoop+IfStatementX2

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
      }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x < 1) {
        System.out.println("Hello");
    }
```

```
    return x + y;
}
```

**Listing A.19:** M19: addForLoop+IfStatementX3

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
      }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    return x + y;
}
```

**Listing A.20:** M20: addForLoopMixX3+IfStatementMixX3

```
private int addition(int x, int y){
    for (int i = 0; x < y; i++) {
        System.out.println(i);
      }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    for (int i = 0; i < x; i++) {
        y++;
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x != y) {
        System.out.println("Hello");
    }
```

```
    if (x < 1) {
        x = x + y;
    }

    return x + y;
}
```

**Listing A.21:** M21: addForLoopMixX3+IfStatementMixX3+changedOrder

```
private int addition(int x, int y){
    for (int i = 0; x < y; i++) {
        for (int i = 0; i < x; i++) {
            System.out.println(i);
        }

      }

    System.out.println(i);

    for (int i = 0; i < x; i++) {
        y++;
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x != y) {
        System.out.println("Hello");
    }

    if (x < 1) {
        x = x + y;
    }

    return x + y;
}
```

**Listing A.22:** M22: addForLoopMixX3+IfStatementMixX3+changedOrder2

```
private int addition(int x, int y){
    for (int i = 0; x < y; i++) {
        for (int i = 0; i < x; i++) {
            System.out.println(i);
        }
```

```
      }

    System.out.println(i);

    for (int i = 0; i < x; i++) {
        y++;

        if (x < 1) {
            System.out.println("Hello");
        }
    }

    if (x != y) {
        System.out.println("Hello");
    }

    if (x < 1) {
        x = x + y;
    }

    return x + y;
}
```

**Listing A.23:** M23: addForLoopMixX3+IfStatementMixX3+changedOrder3

```
private int addition(int x, int y){

    if (x != y) {
    for (int i = 0; x < y; i++) {
        for (int i = 0; i < x; i++) {
            System.out.println(i);
        }

      }
    }

    System.out.println(i);
    System.out.println("Hello");

    for (int i = 0; i < x; i++) {
        y++;

        if (x < 1) {
            System.out.println("Hello");
        }
    }
```

```
    if (x < 1) {
        x = x + y;
    }

    return x + y;
}
```

**Listing A.24:** M24: addForLoopV2X3+IfStatementV2X3

```
private int addition(int x, int y){
    for (int i = 0; x < y; i++) {
        System.out.println(i);
      }

    for (int i = 0; x < y; i++) {
        System.out.println(i);
    }

    for (int i = 0; x < y; i++) {
        System.out.println(i);
    }

    if (x != y) {
        System.out.println("Hello");
    }

    if (x != y) {
        System.out.println("Hello");
    }

    if (x != y) {
        System.out.println("Hello");
    }

    return x + y;
}
```

**Listing A.25:** M25: addForLoopV2X3+IfStatementX3

```
private int addition(int x, int y){
    for (int i = 0; x < y; i++) {
        System.out.println(i);
      }

    for (int i = 0; x < y; i++) {
        System.out.println(i);
    }
```

```
    for (int i = 0; x < y; i++) {
        System.out.println(i);
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    return x + y;
}
```

**Listing A.26:** M26: addForLoopV3X3+IfStatementX3

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        y++;
      }

    for (int i = 0; i < x; i++) {
        y++;
    }

    for (int i = 0; i < x; i++) {
        y++;
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x < 1) {
        System.out.println("Hello");
    }
```

```
    return x + y;
}
```

**Listing A.27:** M27: addForLoopX2+IfStatement

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
      }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    if (x < 1) {
        System.out.println("Hello");
    }
    return x + y;
}
```

**Listing A.28:** M28: addForLoopX3

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    return x + y;
}
```

**Listing A.29:** M29: addForLoopX3+IfStatement

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
      }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }
```

```
    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    if (x < 1) {
        System.out.println("Hello");
    }
    return x + y;
}
```

**Listing A.30:** M30: addForLoopX3+IfStatementV2X3

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
      }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    if (x != y) {
        System.out.println("Hello");
    }

    if (x != y) {
        System.out.println("Hello");
    }

    if (x != y) {
        System.out.println("Hello");
    }

    return x + y;
}
```

**Listing A.31:** M31: addForLoopX3+IfStatementV3X3

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
      }

    for (int i = 0; i < x; i++) {
```

```
        System.out.println(i);
    }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    if (x < 1) {
        x = x + y;
    }

    if (x < 1) {
        x = x + y;
    }

    if (x < 1) {
        x = x + y;
    }

    return x + y;
}
```

**Listing A.32:** M32: addForLoopX3+IfStatementX3

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
      }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x < 1) {
        System.out.println("Hello");
    }

    if (x < 1) {
        System.out.println("Hello");
```

```
    }

    return x + y;
}
```

**Listing A.33:** M33: addForLoopX3+IfStatementX3+changedOrderAndContent

```
private int addition(int x, int y, int z){
    for (int i = 0; i < x; i++) {
        y = y + i;
      }

    if (y < x) {
        y = y + 100;
    }

    for (int i = 0; i < x; i++) {
        System.out.println("Running");
        System.out.println("Hola");
    }

    x = z;

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    if (y > 50) {
        System.out.println("Bonjour");
        if (y < 100) {
            System.out.println("Okay");
            y = 2*z;

        }
    }

    return y;
}
```

**Listing A.34:** M34: addForLoopX3+IfStatementX3+changedOrderAndContent2

```
private int addition(int x){
    int y = 0;

    for (int i = 0; i < x; i++) {
        y = y + i;
```

```
    }

    if (y < x) {
        y = y + 20;
    }

    if (y < 100) {
        for (int i = 0; i < x; i++) {
            System.out.println("Running");
            System.out.println("Hola");
        }
    }

    for (int i = 0; i < x; i++) {
        y++;
    }

    if (y > 50) {
        System.out.println("Bonjour");
    }

    return y;
}
```

**Listing A.35:** M35: addForLoopX5

```
private int addition(int x, int y){
    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }

    for (int i = 0; i < x; i++) {
        System.out.println(i);
    }
```

```
    return x + y;
}
```

**Listing A.36:** M36: addIfStatementX3

```
private int addition(int x, int y) {
  if (x < 1) {
    System.out.println("Hello");
  }

  if (x < 1) {
    System.out.println("Hello");
  }

  if (x < 1) {
    System.out.println("Hello");
  }

  return x + y;
}
```

**Listing A.37:** M37: addIfStatementX5

```
private int addition(int x, int y) {
  if (x < 1) {
    System.out.println("Hello");
  }

  if (x < 1) {
    System.out.println("Hello");
  }

  if (x < 1) {
    System.out.println("Hello");
  }

  if (x < 1) {
    System.out.println("Hello");
  }

  if (x < 1) {
    System.out.println("Hello");
  }

  return x + y;
}
```

**Listing A.38:** M38: addWhileLoopX3

```
private int addition(int x, int y){
    int i = 0;
    while (i < x) {
      System.out.println(i);
      i++;
    }

    while (i < x) {
      System.out.println(i);
      i++;
    }

    while (i < x) {
      System.out.println(i);
      i++;
    }
    return x + y;
}
```

**Listing A.39:** M39: addWhileLoopX5

```
private int addition(int x, int y){
    int i = 0;
    while (i < x) {
      System.out.println(i);
      i++;
    }

    while (i < x) {
      System.out.println(i);
      i++;
    }

    while (i < x) {
      System.out.println(i);
      i++;
    }

    while (i < x) {
        System.out.println(i);
        i++;
    }

    while (i < x) {
        System.out.println(i);
        i++;
    }
```

```
    return x + y;
}
```

# Appendix B
# User Interview Manuscript

## User Interview Manuscript

### Background

Today we are doing a qualitative interview for my master's thesis. The master's thesis is about finding similar methods - primarily from open-source software repositories - to be used as inspirational examples in the context of refactoring recommendations. We have used CodeScene to collect refactoring examples that improved the Code Health of files in popular open-source projects.

### Consent

- Everything you say will be anonymous. As a first step after this interview, we will anonymize the information.
- You do this interview voluntarily. You can skip answering any question, and you are free to stop the interview at any time.
- The information collected during this interview, such as notes, will be securely stored.
- Before we start we want to ask for your consent to also record the interview. The recording will only be used for the thesis and will be deleted when the thesis is completed. Ok?
- We also want to ask for your consent to use your answers for the thesis and for the thesis only. Your answers will be kept anonymous and confidential. Ok?
- You have the right to withdraw your consent if you change your mind.

### Basic information

So first some basic information about you…

- Please describe your experience/career as a developer.
    - How long have you worked as a developer?
    - Describe your current role?
- You'll get to see some Java code. How much Java experience do you have?
- We're going to discuss refactoring examples. What thoughts come to your mind when I say refactoring? Please elaborate. Posivite/Negative?

### Perspective

Before we start with the main task, we want you to imagine that it is Friday and it is clean-up day at the office. You will get the whole day to clean up some code that you haven't had time to clean up before.

### Instructions

We will show you a Java method containing one code smell detected by CodeScene. Suppose your task is to remove that code smell. To inspire you, you will get to see two previous refactoring examples.

We will show you two different examples of how some unknown developer commited a refactoring change that - again according to CodeScene - removed the same code smell from another Java method in some other project.

We want you to explain which of the two inspirational examples that you perceive as the most helpful in your refactoring task.

You will get ten methods in total. The methods are classified according to a code smell that it contains. Take the code smell into consideration when looking at the recommendations.

We are doing this according to the think aloud protocol which basically means that we want you to verbalize your thoughts while you are looking at the example methods and the refactoring recommendations.

**Examples**
Here is the first example. Feel free to zoom in and out. *Read the code smells*

…

Now that you've looked at this, what would you say is a good refactoring recommendation?
Is there anything that you would like to add that came up during the interview?

**EXAMENSARBETE** Evaluating Similarity-Based Refactoring Recommendations
**STUDENT** Emma Ericsson
**HANDLEDARE** Markus Borg (LTH), Emil Aasa (CodeScene)
**EXAMINATOR** Emma Söderberg (LTH)

# Hur får man en programmerare att städa i sin kod?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Emma Ericsson**

Med en växande mängd källkod finns det ett ständigt behov av att "städa" den för att undvika onödigt komplexa lösningar. Som hjälp vill programmerare få rekommendationer som är korta och enkla att förstå.

Eftersom vår värld ständigt går mot ökad digitalisering växer mängden mjukvara lavinartat. Inom företag så finns det mer och mer kod att hålla reda på och det är många personer som skriver på den. Det är som att hundratals personer håller på att skriva en och samma enorma bok tillsammans. Inom programmering kallas all kod som tillhör en mjukvaruapplikation en kodbas.

För att alla som arbetar med koden ska kunna förstå den, måste den vara enkel att läsa och arbeta med. För att koden ska vara enkel och lätthanterlig så behöver man med jämna mellanrum städa i den. Precis som att man omarbetar ett stycke eller en mening som är svår att förstå. Med verktyget CodeScene kan man få rekommendationer som hjälp för att städa i koden. Om en kollega som arbetar på samma kodbas har arbetat med liknande kod som dig själv och gjort en ändring som gör att koden blir enklare så rekommenderar CodeScene dig att själv göra en liknande ändring. Om ingen har gjort en sådan ändring tidigare så får du bara ett skolboksexempel på vad som skulle kunna göras. För att få mer hjälp än vad ett standardiserat skolboksexempel kan erbjuda, ville vi utforska sätt att hitta rekommendationer utanför den egna kodbasen.

Låt oss återgå till bokliknelsen. Trots att det finns hundratals författare, vill vi ändå att boken ska vara enhetlig och skriven på samma sätt från början till slut. Det här vill vi ha i en kodbas också. Därför ville vi hitta rekommendationer som passade in på det sättet som kodbasen var skriven på. För att koden ska hållas enhetlig har vi därför undersökt hur man kan hitta rekommendationer som liknar de man skulle ha fått om en kollega hade gjort liknande ändringar i kodbasen tidigare. Detta gjordes genom att hitta liknande kod från andra kodbaser som är allmänt tillgängliga.

Det finns inget entydigt sätt att jämföra kod på så vi undersökte olika sätt och testade till slut två av dessa i försök med erfarna programmerare. Det ena sättet jämförde bokstavligen själva texten som koden bestod av medan det andra snarare jämförde innebörden av koden. Det var inget av sätten vi jämförde koden på som visade sig vara bättre eller sämre än det andra men det visade sig att rekommendationer som är framtagna genom att hitta liknande kod skulle kunna vara hjälpsamma för programmerare. Deltagarna i studien förespråkade starkt rekommendationer som var korta, tydliga och lätta att förstå.