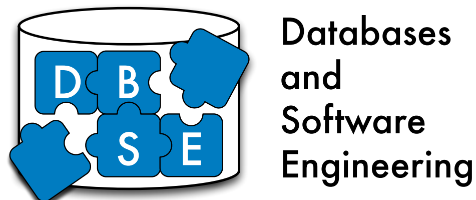
University of Magdeburg

School of Computer Science



Master's Thesis

Why, How, and When Refactorings are (NOT) Applied: A Systematic Literature Review

Author:

Vladyslav Buriakovskyi

November 26, 2018

Advisors:

Dr.-Ing. Sandro Schulze
Prof. Dr. rer. nat. habil. Gunter Saake

Databases and Software Engineering Research Group

Buriakovskyi, Vladyslav:
Why, How, and When Refactorings are (NOT) Applied: A Systematic Literature Review
Master's Thesis, University of Magdeburg, 2018.

Abstract

Refactoring is an area in software engineering that every person who is programming is confronted with. It has many direction of research and a large number of written articles. Each programmer is engaged in refactoring, sometimes without even noticing and not understanding that. But how exactly is refactoring used? Why refactoring is used? When is it performed, and when is it neglected? Where to do refactoring and what is used for refactor? In this thesis we will try to answer these questions from a practical point of view. To do this, we conducted a systematic literature review on articles that explored the practical use of refactoring, and show which techniques are used most often and by which methods. We will try to answer the questions why refactoring is used or vice versa, not used, how it is used and where it is used. We will show the main reasons and motivators for each of the questions asked, describe them in detail and try to explain them. Also, we will show other findings in the field of refactoring from practical point of view, which were found during our analysis. In the end, we will discuss the conflicting data and name the possible reasons for this and describe the limitations of our study.

Contents

List of Figures ix

List of Tables xi

1 Introduction 1
1.1 Motivation . . . 1
1.2 Goal of this Thesis . . . 2
1.3 Outline . . . 2

2 Background 3
2.1 What is Refactoring? . . . 3
2.1.1 Refactoring Types . . . 5
2.1.2 Refactoring Possibilities . . . 6
2.2 Conducting a Systematic Literature Review . . . 6

3 Methodology 9
3.1 Planning the Review . . . 9
3.1.1 Research Questions . . . 9
3.1.2 Systematic Search Strategy . . . 10
3.2 Conducting the Review . . . 11
3.2.1 Selection Criteria . . . 11
3.2.2 Quality Assessment . . . 12
3.2.3 Data Extraction . . . 12
3.3 Reporting the Results . . . 13
3.3.1 Data Collection . . . 13
3.3.2 Snowballing . . . 14
3.3.3 Primary Studies . . . 15
3.3.4 Data Synthesis . . . 18

4 Results and Findings of the SLR 21
4.1 Meta Data of Studies . . . 21
4.2 Quantitative Data . . . 23
4.2.1 High-Level Refactorings . . . 24
4.2.2 Low-Level Refactorings . . . 27
4.3 Qualitative Data . . . 32
4.3.1 RQ1: Why Refactoring is Not Used? . . . 33
4.3.2 RQ1: Why Refactoring is Used? . . . 36
4.3.3 RQ2: When Refactoring is Used? . . . 39

4.3.4 RQ3: How Refactoring is Used? 40
4.3.5 Other Findings 41
4.4 Discussion 44
4.5 Study Limitations 46

5 Related Work **47**

6 Conclusion **49**

7 Future Work **51**

A Appendix **53**

Bibliography **61**

List of Figures

2.1 Class Person representing God Class smell . . . 4
2.2 Class Person after refactoring . . . 4

4.1 Types of studies . . . 22
4.2 Expirience of participants . . . 22
4.3 Distribution of program languages . . . 22
4.4 Amount of investigated code . . . 22
4.5 Most considered projects . . . 22
4.6 Relation between automated and manual High-Level refactorings . . . 26
4.7 Distribution of High-Level refactorings . . . 26
4.8 Distribution of automated High-Level refactorings . . . 27
4.9 Distribution of manual High-Level refactorings . . . 27
4.10 Distribution of High-Level refactorings without data about usage . . . 27
4.11 Detailed relation between manual and automated High-Level refactoring techniques . . . 28
4.12 Relation between manual and automated Low-Level refactorings . . . 28
4.13 Distribution of Low-Level refactorings . . . 30
4.14 Distribution of automated Low-Level refactorings . . . 30
4.15 Distribution of manual Low-Level refactorings . . . 31
4.16 Detailed relation between manual and automated Low-Level refactoring techniques . . . 31
4.17 Distribution of Low-Level refactorings without data about usage . . . 32

List of Tables

3.1 PICOC Criterias 10

3.2 Quality Assessment 12

3.3 Data Extraction Form - Quantitative Data 13

3.4 Data Extraction Form - Qualitative Data 14

3.5 Initial Search 14

3.6 Backward Snowballing 15

3.7 Forward Snowballing 15

3.8 Summary of Snowballing 15

3.9 Selected Studies 18

4.1 Refactoring Techniques generalized as ”Other” for High-Lever refactorings 25

4.2 Refactoring Techniques generalized as ”Other” for Low-Lever refactorings 29

4.3 Open Coding - Why refactoring is NOT used? 34

4.4 Open Coding - Why refactoring is used? 37

4.5 Open Coding - When refactoring is used? 39

4.6 Open Coding - How refactoring is used? 41

4.7 Open Coding - Other findings 42

A.1 All Excluded Studies 55

A.2 Quality Assessment - Included Studies 57

A.3 Quality Assessment - Excluded Studies 58

A.4 Quality Assessment for Papers excluded by Data Extraction 59

1. Introduction

In this chapter we present the motivations of this thesis, briefly show research questions, used methods and describe the thesis structure.

1.1 Motivation

The process of creating software includes many tasks and is not limited to just implementing the required functionality of the program. One of the tasks is not only write the working code, but write a code that is easily readable, extensible, has no "code-smells" and uses rationally the capabilities of a programming language. However, writing a working and simultaneously good-looking program from the first time is an almost impossible task. That's why programmers may improve the structure of the code during the programming or on different stages. This process is called *refactoring* and its main task is to improve the design of existing code without changing its behaviour.

The first mention of refactoring appeared already in the early 90s of the 20th century. For almost thirty-year history many articles have been written about refactoring, its need, methods, how to conduct it, when and why. The development of programming languages also contributed to the development of refactoring, its techniques and the emergence of new opportunities for automatic refactoring using certain tools. Despite many different programming languages, some techniques of refactoring can be applied in a language-independent fashion.

Refactoring is often performed not only as a separate stage of software development. Very often it is carried out simultaneously with the addition of new functions, the improvement of program performance or the search and removal of bugs. It can be said, that refactoring is an integral part of writing a program. Refactoring in an explicit or implicit form is applied to almost all software that is being created.

Moving from theory to practice, the questions on the use of refactoring arises. Methods, objectives, opportunities for refactoring are clear. But it is not clear, when the use of refactoring are rational, how it could be trouble-free carried and how to not

create new issues during this process. Even more, it is unknown what type of refactoring are better in different cases and is the tool support for refactoring sufficient. From that arises the question: how exactly do developers use refactoring in practice, in which cases and for what?

The purpose of this master thesis is to try to answer these questions. The main aims are collecting information about practical use of refactoring and investigation this usage. Such a study would help to understand practical point of view on refactorings, motivations, drivers behind refactorings and how developers applying (or not) the refactoring during their work.

1.2 Goal of this Thesis

The method of solving the problem is a systematic literature review, which will include articles, books, researches and reviews about refactoring. In this review, we will try to cover the available information about the use of refactoring by developers and answer questions like when, why and how refactoring is used. This review will be conducted according to a protocol, in which will be described a PICOC method, research questions, literature search methods, sources, definition of primary studies, data extraction and data synthesis.

To identify specific research problems three main research questions are formulated:

RQ1: *Why is refactoring used?*

RQ2: *When is refactoring used?*

RQ3: *How is the refactoring used?*

The answers on this questions will give us results, which will be analysed and represented. These results reflect the actual state of the use of refactoring and will help determine the direction of further development of refactoring.

1.3 Outline

This chapter provides a motivation of this work, understanding the need and research goal. Chapter 2 describes the term Refactoring, main refactoring types, possibilities to do refactoring and principles of conducting systematic literature review. Chapter 3 represents the protocol to our review, including research questions, search strategy, conducting the review, data collection and data synthesis. Chapter 4 presents analysis of selected studies, quantitative and qualitative data found, contradictions in the study and study limitations. Chapter 5 describes related work in field of refactoring. Chapter 6 defines the summary of this thesis, and in the end we show directions of future research.

2. Background

To make it clear what will be exactly studied, we deeply explain what is refactoring, code smells, main terms and approaches of it and shortly show the history of refactoring. Also, this section brings up overview on methodology of conducting systematic literature reviews.

2.1 What is Refactoring?

Refactoring – process of changing a software system, that improves quality and design of existing source code without changing its functionality [Fow02].

From a formal point of view, any change in code can be accepted as refactoring. But its main goal is to make the code more clear, flexible and easier to understand. Refactoring should be distinguished from optimization, which also does not change the behaviour of the program, but only increases speed of program execution. However, in contrast, optimization often makes it difficult to understand the code, which is the opposite of refactoring. Refactoring improves the design of software and makes the cleaner, simpler and elegant. Refactoring did not appear as something separate. Formally, this is an integral part of programming, which was identified as a separate stage of code improvement.

Refactoring has been initially subject to research in the early 80s of the last century. One of the first serious scientific works on refactoring was written already in 1992 [Gri92]. The main incentives for the development of refactoring were the development of programming languages, the need to understand the code by several developers and the emergence of *code smells*.

Code smells is a term, which was introduced by Fowler et al., and describes defects in source code induced by poor design or evolution of a software system [Fow02]. In particular, a code smell represents a concrete template that point outs some deeper problems in the source code or design [FSMS15]. The code smells are often one of the main motivators for refactoring [VGSMD03], because refactoring removes them and improves the overall structure of the program.

As an explaining of refactoring and code smell, we will show example, which was given by Cedrim et al. [CSGG16]. On Figure 2.1 we show the class Person, which has at least three attributes representing two concepts: person and telephone number. This class can be assumed as a *God Class*. *God Class* smell is a class with several responsibilities, which makes the class hard to read, modify and develop [Fow02]. With the aim to remove this smell, the developer can extract part of the class structure into another class TelephoneNumber, which we show on Figure 2.2. Such refactoring technique calls *Extract Class*, the use of which implies creation of a new class and moving the relevant fields and methods from the old class into the new one [Fow02]. After such refactoring the code has no longer *God Class* smell but still has the same functionality.

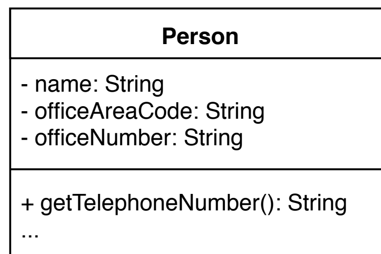


Figure 2.1: Class Person representing God Class smell

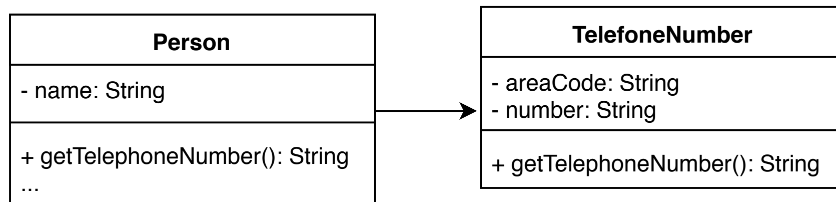


Figure 2.2: Class Person after refactoring

This small example was only one of more than 70 refactoring techniques, which were described in book of Fowler [Fow02]. A large number of refactoring techniques are common [MT04], therefore their principles are successfully used for different languages.

A workflow of refactoring could be given by the guidelines of Test Driven Development (TDD). In terms of TDD were proposed the approach of Red-Green-Refactor [Sho05]:

- Think: Figure out what kind of test will best move for code towards completion.
- Red: Write a small amount of test code. Run the tests and look how it fails: the test bar should turn red.
- Green: Write a small amount of production code. Do not pay attention about design or conceptual elegance. Run the tests and watch them pass: the test bar will turn green.

- Refactor: Now the changes to the code could be carried out without worrying about breaking functionality. Take a look at the written code and ask how it can be improved. Look for duplication and other code smells. After each little refactoring, run the tests and make sure they still pass.
- Repeat: Do it again.

This approach represents the *floss* refactoring, by which the implementation of new functionality is combined with refactoring tasks. There exists another type of refactoring - *root-canal*, where developer has to carry out refactoring without adding new functionality. In more detail we describe this types later.

Refactoring is quite a difficult task, which also has certain risks. One of the main risks in the process of refactoring is the probability of creating new bugs [TGA17, KZN14, Fow02]. Another risk is that although refactoring is intended to improve the appearance of the code, there is a chance that it will still worsen. Programmers try not to touch the working code at all, so as not to disturb its functionality. By performing refactoring, the changes could affect functionality and not for the better. Moreover, such changes can be seen externally. Also, important fact is that refactoring is a time-consuming process and if it is not successful it will be a waste of time and effort [LML+15].

2.1.1 Refactoring Types

Implementation of refactoring could be done it two different ways. One way is separate stage, e.g. maintenance task. Another is during the ordinary development process. Developers to some extent (obviously or not explicitly) often can be engaged in refactoring without even noticing this. The fact is that this two types of refactoring are different and have specific names: *floss refactoring* and *root canal refactoring*. These terms were firstly defined by Murphy-Hill and Black [MHB08b].

Floss refactoring is a refactoring that takes place simultaneously with other programming tasks. Such refactoring often remains unnoticed by the developer himself, because it is implicit. For example: the developer writes the code and adds new functionality to the program. He uses existing functions, but in the process of development it changes their external and makes them more convenient and clearer. The result is a new functional and the refactoring of the already existing code. However, if the refactoring was conducted explicit, but ran together with other programming tasks, such refactoring also applies to floss refactoring.

A completely different type is *root canal* refactoring which is a separately isolated stage in the development of software, in which the developer is only concerned with refactoring.

The main differences between these types of refactoring were given by Hui Liu et al. [LLXG14]. So, root canal refactoring has a larger spectrum of review, requires the use of applications to find the code smells and must be performed on a schedule algorithm. Because of larger spectrum, especially in large applications, it could be difficult to find or identify refactoring opportunities. In this case it could be helpful to use automated detection algorithms to identify code smells and possible

refactorings. On the another hand, floss refactoring seems to be more effective and currently more widely used [MHB07]. Moreover, frequent floss refactoring is helpful in avoiding the need for root canal refactoring later [CXW+16].

2.1.2 Refactoring Possibilities

As already said, one of the main reasons of refactoring are the code smells. Fowler in his work describes the 22 smells of code that precede refactoring and the corresponding techniques of dealing with them [Fow02]. Because manual search is not effective activity, a large number of tools have been created that can automatically detect bad smells [VCN+12]. These tools can very effectively cope with the task and help developers in refactoring pointing to the place in the program where refactoring is needed [STV16].

However, knowing about the need for refactoring at some place does not save the developer from doing it. In most cases refactoring have to be carried out manually. This is partially confirmed by floss refactoring. But in the case of root canal refactoring, by working with large applications, conducting manual refactoring can be a very long and routine exercise.

To simplify the process of refactoring it would be nice to have a tool that could run it automatically. Since the semantics of refactoring techniques are clear, the tools for refactoring can help the developer in this routine work [MH09]. One of the first who started creating such a tool were D. Roberts et al. [RBJ97], which was successfully used for Smalltalk refactorings. Later, other tools were developed which work with different programming languages. A large number of tools were created for languages such as Java, COBOL or PHP. But, some exotic languages like APL or Fortran still do not have such tools [SD16].

Basically, these tools can work with the most common techniques of refactoring. Some sophisticated techniques of refactoring must still be done manually. Also, such tools do not always apply refactoring correctly. The question of using such tools is open, and many scientists are engaged in its research [MH09, Ste17, GDMH12, VCN+12].

2.2 Conducting a Systematic Literature Review

This thesis follows to the guidelines for performing Systematic Literature Reviews in Software Engineering area written by Barbara Kitchenham and Stuart Charters [KC07]. Most of the information used in this section taken from this guidelines. According this guidelines, the *Systematic Literature Review* (SLR) is a form of secondary study that uses a well-defined methodology to identify, analyse and interpret all available evidence related to a specific research question in some research area in a way that is unbiased and repeatable to some degree.

The SLR has own features, which are differentiate it from a conventional expert literature review. SLR starts from defining a review protocol, which specifies the research questions and the methods that will be used to carry out the review. For SLR will be created search strategy, with aim to find so many relevant literature as possible. The search strategy will be documented, so readers and other researcher

can reiterate whole process. The primary studies for review are selected by explicit inclusion and exclusion criteria and, in additional, each primary study evaluated by quality assessment. The information to be obtained from each selected study will be specified.

The SLR involves several discrete activities. The stages of SLR summarized in three main phases: Planning the Review, Conducting the Review and Reporting the Review.

Planning the review confirm the need for a review. The one of the most important pre-review activities are defining the research questions(s), which address the systematic review and producing a review protocol, which defines the basic review procedures. The review protocol is a subject to an independent evaluation process.

The stages associated with planning the review are:

- Identification of the need for a review - arises from the demand to collect all existing information about some problem in a thorough and unbiased manner.
- Commissioning a review - create commissioning document, which specifies the work.
- Specifying the research question(s) - the most important part of any systematic review, because the questions drive the all methodology of review.
- Developing a review protocol - specify the methods that will be used to undertake a review.
- Evaluating the review protocol.

In this thesis, the stage of commissioning a review is incorporated into the review protocol, which contains and describes deeply all stages according to planning. The protocol is described in Chapter 3.

As the protocol has been agreed, the researcher can start the review. The stages associated with conducting the review are:

- Identification of research - generate a strategy and documenting of search and explore the potential relevant studies.
- Selection of primary studies - identify primary studies that provide evidence about the research questions.
- Study quality assessment - consider critical to assess the “quality” of primary studies.
- Data extraction and monitoring - design data extraction forms to record the information, which is obtained from the primary studies.
- Data synthesis - collate and summarise the results of the included primary studies.

The stages of identification, selection of studies, quality assessment and data synthesis are described in Chapter 3. Data extraction and results of SLR could be found in Chapter 4.

The stages associated with reporting the review are:

- Specifying dissemination mechanisms - how the results of review will be published and where.
- Formatting the main report - how the review will look like.
- Evaluating the report.

All this stages listed above may be sequential, but many of them involve iteration. In particular, activities are initiated during the development stage and refined later during the review.

SLR has some advantages and disadvantages. To the advantages can be attributed:

- Well defined methodology, which makes the results of literature less biased, although does not protect from bias in primary studies.
- Providing information about research area across a wide range of requirements and empirical methods.
- Possibility to merge data using metaanalytic techniques, which increases the change of detecting real effects that small studies are unable to detect.

The main disadvantage of SLR is fact, that it requires considerably more effort than traditional review. Another disadvantage could be increased power for meta-analysis, which makes possible to detect small biases and true effects.

3. Methodology

In this chapter we present the protocol of our SLR that we conduct in this thesis. In particular, we explain research questions, aspects of our SLR and literature search strategy. Moreover, we describe how the primary studies were selected, inclusion and exclusion criteria, backward and forward snowballing, process of data collection and data synthesis.

3.1 Planning the Review

Despite the long history of refactoring and large number of research and articles, it is not clear, how many attention developers pay to it, how they use it, which techniques and possibilities they use and how common refactoring is. As an extreme example, some programmers even try to avoid refactoring for some reasons [TGA17]. Those, who still use refactoring, do it in some own style based on previous experience [NCV$^+$13]. Given this great diversity in (not) applying refactoring, uncertainty how the refactorings are applied and why, we formulate main our goal of this research: we want to investigate practical usage of refactoring by developers, reasons why they use or try to avoid refactoring and reasons that may lead them to their respective decision. Besides, we want to clarify, which refactoring techniques used more frequently and in which manner, how the developers use them and when the refactorings occur. One of the reason for this study is also the fact, that this direction of refactoring research have not yet been considered in a comprehensive systematic work.

In summary, the objective of this research is to review the current state of using refactoring, purposes of that use or disuse and the way how it is used by developers.

3.1.1 Research Questions

To better define aspects of the literature review, we used the PICOC method proposed by Petticrew and Roberts [PR06]. PICOC is an abbreviation of the population, intervention, comparison, outcomes and context. This method allows to describe in detail the target group for the research (Population), to determine the

aspects of the study (Intervention), to present to what the results are compared (Comparison), to describe the results of the study (Outcomes) and to set the research area (Context). In Table 3.1 we show the PICOC method for this thesis.

PICOC	Description
Population	Literature about code refactoring with empirical analysis of its use
Intervention	Situations, analysis, experience, which give insights why, where, when and how was (or wasn't) refactored i.e. applicability, error proneness, usability, awareness, time for refactoring, integration in the development process, etc.
Comparison	Data about refactoring usage
Outcomes	Identifying gaps/challenges but also opportunities that prevent/allow for refactoring
Context	Software Maintenance/Evolution and Software Development process

Table 3.1: PICOC Criterias

To accomplish our task and determine the scope of the study, we formulated three research questions that would clearly and entirely describe the above mentioned goal of this thesis:

RQ1: *Why is refactoring used or not used?* By which conditions and for what purpose besides improving the code programmers use the refactoring? What are the main motivations for using refactoring and what makes to avoid them? Which factors also affect the decision of programmers to do or not to do refactoring?

RQ2: *When is refactoring used?* How often developers use refactoring? On which stage of programming, while writing the source code or independent, with separately allotted time? Is refactoring part of the development plan, comes spontaneously or there are other conditions?

RQ3: *How is the refactoring used?* What kind of refactoring techniques are mainly used? What type of refactoring is more used and why? Is the manual refactoring more preferred to automated or vice versa?

3.1.2 Systematic Search Strategy

The search process is a manual look for papers, journal articles and other works using the widespread electronic libraries. For finding most relevant studies for our theme, we will determine search string. After conducting a search and collecting the literature, we check if the papers are relevant for our study. The selection will be made according to inclusion and exclusion criteria, which are described below. The resulting step will be complementary search by included literature using *snowballing* techniques and also selection founded papers based on inclusion and exclusion criteria. The last step is to choose most relevant literature through verification by Quality Assessment.

3.2 Conducting the Review

First, we have to define the sources in which we search for papers for our review. This sources are digital libraries, i.e. IEEExplore, ACM Digital library, Google scholar, Science Direct, CiteSeerX, Web of Science.

For more detailed search in electronic libraries we define a search string, according to our scope, as follows:

("refactor" OR "refactoring") AND/OR ("empirical") AND ("study" OR "survey" OR "research" OR "analysis") AND/OR ("methods" OR "types" OR "use" OR "frequency" OR "causes" OR "issues") AND ("software" OR "code")

3.2.1 Selection Criteria

For our work, we created selection criteria for including and excluding articles. These criteria are used to guide the initial selection of articles. This criteria are formulated based on the research questions. Articles will be included if they pass through them.

The Inclusion Criteria are:

- IC01 Empirical studies - studies about refactoring, which bring overview to actual state of art in refactoring, experimental data, data about use or disuse of refactoring, techniques, methods and other quantitative data;
- IC02 Experiments or analyses - papers with controlled experiments or analysis of refactoring with detailed data, why, when and how refactoring is used, by which conditions, which also can contain personal comments, observations or other findings regarding to refactoring;
- IC03 Comparison studies - articles which compare different types and possibilities of refactoring and give a sign, what types and methods are most frequently used, why and is there some dependencies;
- IC04 Articles published between 1th January 2000 and 31th May 2018. The book of Fowler was firstly written in 1999 [Fow02] and from that moment the active research into the practical use of refactoring began.

The Exclusion Criteria are:

- EC01 Abstract - literature, that are not focused on an application or the empirical research of refactoring, technical literature;
- EC02 Overview - papers about refactoring process, methods, tutorials;
- EC03 Tools papers - articles about tools, that support automated refactorings as well papers about tools, which allow to find refactorings;
- EC04 Papers not written in English.

If the study meets at least one exclusion criteria, it will be excluded.

3.2.2 Quality Assessment

After all searches, for selecting primary studies we will evaluate them by quality assessment criteria. These criteria will measure the quality of the each article and allow us to choose only the most suitable for our research. For each criteria the article will receive a certain number of points (note). The scoring procedure is Yes=1, No=0 and Partially=0,5. The criteria are questions, which we described in Table 3.2.

The articles, which passed inclusion criteria and will get note more than 4 by Quality Assessment will be chosen for data extraction. The grade 4 was chosen because only articles with this grade and above, according to the quality assessment questions, will contain enough information for our research.

Number	Question	Answer
Q01	Was the article referred?	Yes/No
Q02	Was the study conducted?	Yes/No/Partially
Q03	Was the target(s) clearly formulated?	Yes/No/Partially
Q04	Were the study participants or observational units clearly described?	Yes/No/Partially
Q05	Were the data collections executed well?	Yes/No/Partially
Q06	Were the chosen analyses methods compared and argued?	Yes/No/Partially
Q07	Was the possible inaccuracy taken into account?	Yes/No/Partially
Q08	Were the conclusions reliable?	Yes/No/Partially

Table 3.2: Quality Assessment

3.2.3 Data Extraction

To collect the relevant information from each article in a structured way, we created a Data Extraction Form. Each article we selected was analysed and data from it was collected. This form is divided into three parts. In first part we recorded meta data of articles, such as ID, authors, title, article type and aspect of study.

The second part is for quantitative data. We contributed data, such as the number of participants in the study, their experience in programming, the projects reviewed and the amount of the examined code. In Table 3.3 on the facing page we show fields, according to quantitative data. The field "Experience of participants in programming" is divided into three subfields, in which we highlight three groups of people according to their experience.

We also collect the information about which refactoring techniques were used and corresponding amount of them. This could be data from experiments, where developers did refactorings and researches recorded this data, or other data, for example that was achieved with help of tools for searching for refactorings. Later, based on

this data, we will show the distribution and the most commonly used refactoring techniques.

Aspects	Data Fields
Total number of participants in study	
Experience of participants in programming	<5 years: 5-10 years: >10 years:
How many and which projects are considered in study?	
Which program language is considered?	
How many commits are investigated?	
How many versions are investigated?	
How many lines of code are considered?	
Which refactoring techniques are used and corresponding amount of them?	

Table 3.3: Data Extraction Form - Quantitative Data

The third part of Data Extraction Form is qualitative data according to our research questions. This part is represented in Table 3.4 on the next page. Fields such as ”Reasons for using refactoring”, ”Targets of using refactoring” and ”When refactoring is used and when not?” will be filled with text data. In the field ”How is refactoring used?” we entered information about what type of refactoring was considered in the article (manual or made with the help of refactoring tools). The fields ”Comments from personal experience of programmers” and ”Other findings” are also text fields containing real comments from developers and additional information that were found by the authors of the articles during the research.

3.3 Reporting the Results

As mentioned above, we examined six on-line databases, which contain Software Engineering papers, using defined search string. The search took place in June 2018 and was two week long. During the search we paid attention on title of article, abstract and in some cases on conclusion. Decision to include or exclude paper was took according to inclusion and exclusion criteria.

3.3.1 Data Collection

Generally, we found 372 articles related to theme Refactoring. Out of them 187 articles were excluded, because they are not focused on empirical research or application of refactorings, 34 were overview papers, 83 were papers, which describe the refactoring-tools and their usage and one paper were written in Spanish. 65 papers passed all inclusion criteria, but four of them were duplicated (already previously found). Thus, from initial search were selected 61 papers. Overview about numbers of papers per database and how many of them were included or excluded we show in Table 3.5 on the following page.

Aspects	Data Fields
Reasons for using refactoring	
Targets of using refactoring	
When refactoring is used and when not?	
What kind of distribution is between root-canal and floss refactoring?	
How is refactoring used?	
What kind of distribution is between manual and automated refactoring?	
Comments from personal experience of programmers	
Other Findings	
To which research question refers the paper?	

Table 3.4: Data Extraction Form - Qualitative Data

Data source	All	IC01-04	EC01	EC02	EC03	EC04	Dupl.	Sel.
ACM	129	26	41	12	49	1	0	26
CiteSeerX	99	12	57	16	14	0	17	10
IEEExplore	33	7	23	1	2	0	8	6
Scholar	56	12	35	4	5	0	14	11
Science Direct	39	5	20	1	11	0	2	5
Web of Science	16	3	11	0	2	0	7	3
Total	372	65	187	34	83	1	48	61

Table 3.5: Initial Search

3.3.2 Snowballing

After initial search we started *Snowballing* procedure to find more related papers, which could be not found by manual search. *Snowballing* is a technique, that allows to find related literature from the list of references (*Backward snowballing*) or by articles, which cite the founded literature (*Forward snowballing*). The Snowballing were divided into two iterations: on first iteration we took papers from initial search, on second - papers, which were found on first iteration of snowballing. In each iteration we did backward and forward snowballing. As mentioned above, by backward snowballing we searched articles, which were in list of references of initial founded papers. By forward snowballing we searched articles, which cite papers from initial search. In Table 3.6 on the next page we show the results of backward snowballing both iterations. Here, column ”References” are number of all cited articles through all papers from initial search. Other columns shows how many papers were excluded

according to each exclusion criteria, how many papers were duplicated by citing between papers and last column shows how many new papers were found among all references.

References	EC01	EC02	EC03	EC04	Duplicated	Selected
2415	1790	201	155	2	262	3

Table 3.6: Backward Snowballing

In Table 3.7 we show the forward snowballing both iterations. The difference between previous table is in first column: here we mention instead number of references number of articles, which cite papers from initial search.

Cited	EC01	EC02	EC03	EC04	Duplicated	Selected
1769	1178	89	104	14	139	9

Table 3.7: Forward Snowballing

In Table 3.8 we show the summary of both snowballing techniques. Thus, during snowballing process we obtained 4184 papers and 12 of them were selected for our study.

Reference and citing	EC01	EC02	EC03	EC04	Dupl.	Selected
4184	2968	290	259	16	401	12

Table 3.8: Summary of Snowballing

3.3.3 Primary Studies

After carrying out all searches, we found 73 articles that are suitable for our research. The next step were to determine the quality of the founded articles and to select the most suitable ones from them. For this purpose, each article were verified by the quality assessment, as described in Section 3.2.2. For each question of quality assessment we use ◇ for score 0, ⬙ for score 0,5 and ◆ for score 1. In Table A.2 on page 57 we show scores by quality assessment for each selected study and in Table A.3 on page 58 we represent quality assessment and scores for excluded studies.

In the end of quality assessment 51 studies were selected and 22 rejected. Moreover, out of the 51 that passed the quality assessment, 14 studies were rejected on the stage of data extraction, because of lack of data for our research. In Table A.4 on page 59 we show scores for studies, which were excluded during data extraction. In Table 3.9 on page 18 we show all studies, which remained and used in this thesis. In Table A.1 on page 55 we represent all studies, which were excluded during data extraction.

Study ID	Name	Year	Source
P01	A case study on refactoring in Haskell programs [Lee11]	2011	ACM
P02	A Multidimensional Empirical Study on Refactoring Activity [TGSH13]	2013	ACM
P03	An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution programs [KCK11]	2011	ACM
P04	An Exploratory Study on the Relationship between Changes and Refactoring [PZODL17]	2017	ACM
P05	Does refactoring improve software structural quality? A longitudinal study of 25 projects [CSGG16]	2016	ACM
P06	Drivers for Software Refactoring Decisions [ML06]	2006	ACM
P07	How does refactoring affect internal quality attributes? [CFF+17]	2017	ACM
P08	How We Refactor, and How We Know It [MHPB09]	2009	ACM
P09	Issues Arising From Refactoring Studies: An Experience Report [CS12]	2012	ACM
P10	Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study [OKS+16]	2016	ACM
P11	Reconciling Manual and Automatic Refactoring [GDMH12]	2012	ACM
P12	Reflections on Teaching Refactoring: A Tale of Two Projects [AABA15]	2015	ACM
P13	Use, Disuse, and Misuse of Automated Refactorings [VCN+12]	2012	ACM
P14	Why We Refactor? Confessions of GitHub Contributors [STV16]	2016	ACM
P15	An Empirical Evaluation of Refactoring [WKK07]	2007	CiteSeerX
P16	Package Evolvability and its Relationship with Refactoring [MCHH07]	2007	CiteSeerX

Study ID	Name	Year	Source
P17	Refactoring Practice: How it is and How it Should be Supported – An Eclipse Case Study [XS06]	2006	CiteSeerX
P18	Why Don't People Use Refactoring Tools? [MHB07]	2007	CiteSeerX
P19	An Empirical Study of Refactoring Challenges and Benefits at Microsoft [KZN14]	2014	IEEExplore
P20	What Motivate Software Engineers to Refactor Source Code? Evidences from Professional Developers [Wan09]	2009	IEEExplore
P21	A Case Study of Refactoring Large-Scale Industrial Systems to Efficiently Improve Source Code Quality [SNFG14]	2014	Google Scholar
P22	An Empirical Study on Refactoring Activity [HRP+14]	2014	Google Scholar
P23	Barriers to Refactoring [TGA17]	2017	Google Scholar
P24	A Comparative Study of Manual and Automated Refactorings [NCV+13]	2013	Google Scholar
P25	Improving Refactoring with Alternate Program Views [MH06]	2006	Google Scholar
P26	Refactoring—a Shot in the Dark? [LML+15]	2015	Google Scholar
P27	Empirical study on refactoring large-scale industrial systems and its effects on maintainability [SAN+17]	2017	Science Direct
P28	Evaluating refactorings for spreadsheet models [CFM+16]	2016	Science Direct
P29	Perspectives on refactoring planning and practice: an empirical study [CXW+16]	2016	Web Of Science
P30	Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects [CGM+17]	2017	Web Of Science
P31	A Case Study in Refactoring Functional Programs [TR03]	2003	ACM
P32	How Are Java Software Developers Using the Eclipse IDE? [MKF06]	2006	ACM
P33	Major motivations for extract method refactorings: analysis based on interviews and change histories [LL16b]	2016	ACM

Study ID	Name	Year	Source
P34	What Kinds of Refactorings are Co-occurred? An Analysis of Eclipse Usage Datasets [SCY+14]	2014	ACM
P35	Empirical Analysis of Software Refactoring Motivation and Effects [Gil15]	2015	IEEExplore
P36	Case study on software refactoring tactics [LLXG14]	2014	CiteSeerX
P37	Programmer-Friendly Refactoring Tools [MH09]	2009	CiteSeerX

Table 3.9: Selected Studies

3.3.4 Data Synthesis

All articles will be read and analysed. From each of them we will collect qualitative and quantitative data, which will be separately retained. Afterwards we will look for the dependencies between this data and try to discuss and explain them. All data will be collected according Data Extraction Form. After collecting, we will summarize and write descriptive and quantitative data in two corresponding tables.

First table will collect qualitative data and based on this table it will be possible to identify whether results from studies are consistent with each another (i.e. homogeneous) or inconsistent (e.g. heterogeneous). Afterwards we will explain all collected in this table data and based on them will answer our research questions. Besides, in this table we will also collect personal comments from developers and other notes or findings from research, which could be useful or important for our research area.

In another table we will collect quantitative data, such as refactoring techniques used and corresponding amount of them. This data will be used to find most used techniques. Besides, we will show, which techniques are more often used manual and which with help of refactoring tools, the percentage distribution of them and will discuss this results.

Because of big amount and possible heterogeneous of qualitative data, it could be complicated to find most important conclusions and statements. To achieve better results by qualitative analyses we will use *Open Coding* technique, which were purposed by Strauss and Corbin [CS90]. *Open Coding* is the interpretive process by which data are broken down analytically.

The principle of the Open Coding technique is to isolate the main idea from text fragments, understand its main message and “code” it in one or several words. The researcher reads textual data and makes notes in the margins of words or phrases that summarize the meaning of text [BGS+08]. Further, from this codes are created certain categories, which include one or more codes. Subsequently, the categories themselves are assigned to a specific research question.

In the process of analysing all written out fragments of the text, these fragments will be coded and categorized either to already created "codes" or they created a new previously non-existing "code". In the end, for each research question we receive a list of relevant categories, each of which included a set of codes. Each of the codes has a specific meaning and our task is to explain what a particular code means and how all collected information answers on research question.

4. Results and Findings of the SLR

In this chapter we present the results and findings of our review. We show the collected meta data of primary studies and quantitative data we extracted within our SLR, such as kind of studies, the considered programming languages and projects and total amount of developers. Then, we present the quantitative data according to refactoring techniques, the qualitative data according to our research questions and other findings of review. At the end of chapter we discuss the conflicting data and present limitations of our study.

4.1 Meta Data of Studies

Our data reveal that most of the papers (about 80%) conducted empirical studies to investigate different aspects of refactoring. Some of them also contained Interviews (4) or Surveys (5) and one paper was Qualitative Evaluation. Under Empirical Evaluation we summarized different types of studies, such as Empirical Studies (10), Case Studies (7), Longitudinal Studies (2), Quantitative Investigations (1), A multi-project Studies (1), Experience Reports (1), Industrial Case Studies (1), Formative Studies (1), Experiments (1), Field studies (1) and Large-scale Studies (1). We summarized these different types under one term, because all of them constitute a Quantitative Evaluation, with focus on numbers/numeric values, hat allow to reason about who, what, where and how did. Also, such type of studies allows measure of variables, uses statistical data analysis and could be generalized to large samples of data. For example, Case Study can be described as an intensive, systematic investigation of some unit of interest in which the researcher examines in-depth data relating to several variables. Longitudinal study describes and analyses repeated observations of the same variables or events. Formative studies use qualitative and quantitative methods to help to identify and to understand characteristics of research area that influence decisions and actions. A Field Study is a general method for collecting data about users, user needs, and product requirements that involves

observation and interviewing. One paper from our study refers to Qualitative Evaluation, which data is non-numerical. On Figure 4.1 we show the distribution of types of studies.

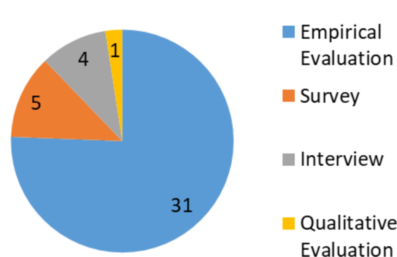


Figure 4.1: Types of studies

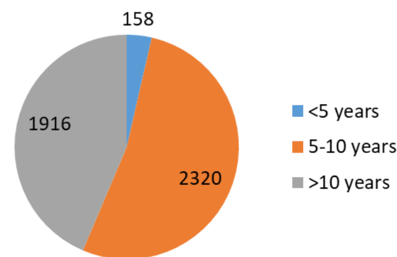


Figure 4.2: Expirience of participants

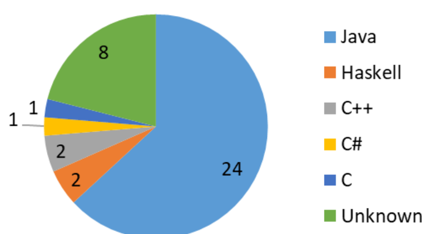


Figure 4.3: Distribution of program languages

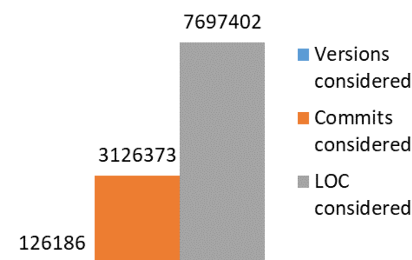


Figure 4.4: Amount of investigated code

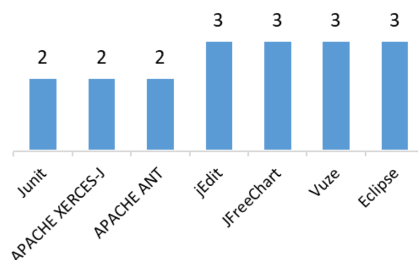


Figure 4.5: Most considered projects

Among all articles the total amount of participants was 1 072 556. Unfortunately, in most of the articles the experience of the participants was not indicated or mentioned in any way. But, from those, were it was indicated, we summarized data about experience and showed it on Figure 4.2. As shown, the researchers rarely studied refactoring of developers, which experience lower than 5 years. Most papers show data about refactorings use related to experienced programmers.

We also summarized the information, about which program language was considered in articles. This data for refactoring is showed on Figure 4.3. Our data reveal that most considered language were Java. Only rarely, other languages such as C/C++ or Python have been considered in the primary studies. Some papers even considered more than one language. In some articles the language was not specified, which is indicated as *Unknown*. The larger amount of articles in Java could be explained with large popularity of Java language for projects and good refactoring support by tools and develop environments for Java.

The total amount of considered projects in our primary studies was 224. Here as project we define a software system, that was researched as part of the study. Many papers didn't point out the names of considered projects. In papers, where the name of project was defined, we collect the number of times the project has been considered across all studies. Most of projects according to distribution of program languages are written in Java. Moreover, between this projects are well-known projects as jEdit, Ant and Eclipse. On Figure 4.5 on the facing page we show the most considered projects and how often they were considered.

Among all projects we summarized the information, about how many versions, commits and lines of code (LOC) were investigated. On Figure 4.4 on the preceding page we show the data of considered code. By versions we mean, that researchers investigated differences between two releases of software. Amount of commits represent among of all considered commits across all projects respectively. The last column on Figure 4.4 on the facing page constitutes the total amount of LOC, that was investigated across all projects.

4.2 Quantitative Data

At the data extraction stage, we found that many articles considered data about using refactoring techniques. Out of our 37 primary studies, 13 articles provide detailed information about the refactoring techniques they investigated, which we consider for the following analyses.

Generally, researchers manually or automatically analysed code changes and identified which techniques were used in one or another place. In some articles, these were controlled experiments [AABA15, GDMH12], in others were repositories mined [Lee11, TGSH13, CFF+17]. In most cases, researchers used techniques described by Fowler [Fow02]. This allows us to bring together all the data among all the articles about which techniques are most frequently used.

The articles dealt with both types of refactoring - manual and automatic. In two of them was clearly stated that the researchers consider only refactorings made with the help of tools [MHPB09, VCN+12] and two more - that they consider a combination of manual and automatic refactorings [STV16, NCV+13]. Unfortunately, many of them (9 articles) did not say what type of refactoring was considered. Based on the obtained data and the analysis of the papers, we can assume that in most cases of articles where the type of refactorings was not indicated, the automatic refactorings were considered. This is indicated by the distribution of the number of articles with an explicitly indicated type, in which dominates the reviews of automatic refactorings, as well as the number of used techniques considered.

Most articles used automatic methods for detecting refactorings using special tools. Such a method makes it easier to reduce the time to search for refactorings. However, the main disadvantages of this method are that only a small number of techniques can be identified (10-15 techniques) and researchers point out that the detection accuracy is relatively low [MH09, NCV+13].

Totally, among all the articles, we managed to allocate 374 633 applied refactoring operations. As mentioned above, there are obviously manual, explicitly automatic

and implicit refactorings among them. Because in some papers the amount of used refactoring techniques were generalized (for example, was given amount of "*Extract*" technique) and since the Fowler refactoring list contains more than 70 separate techniques, analysis using all of them would be inconspicuous and difficult. We decided to divide the techniques into two levels - a *High-Level* and a *Low-Level* refactorings. This was done for the same reasons that some specific refactoring techniques are not automatically supported and can be performed only manually, while others are rarely used, as well as for the possibility of a more detailed analysis in cases of clearly specified techniques.

For High-Level refactorings we conducted a generalization of refactoring techniques into categories. We created seven categories, they are: *Extract*, *Inline*, *Move*, *Pull Up*, *Push Down*, *Rename* and *Other*. For example, the category *Extract* contains summarized data about using techniques such as *Extract Class*, *Extract Constant*, *Extract Interface*, *Extract Local Variable*, *Extract Method* and *Extract Superclass*. Accordingly, the category *Inline* was summed up by the *Inline Constant*, *Inline Function*, *Inline Local Variable* and *Inline Method* techniques. Respectively, the *Move* category contains techniques, which deal with moving elements of code into another part of program. Category *Pull Up* explains all techniques, which describe moving elements to a superclass. Category *Push Down* is also inheritance-related and contains techniques of moving elements to subclass. Category *Rename* deals with techniques according to renaming elements. Other techniques were also subject to generalization and were recorded in the category *Other*. The list of techniques caught in the category *Other* and their number we present in Table 4.1 on the next page. The High-Level contains all 374 633 refactoring techniques.

In the Low-Level refactorings we consider non-generalised data from the High-Level refactorings and without their *Other* list. It means, that in this case in contrast to the High-Level refactorings, we look at the concrete techniques and their frequency of use. This is done in order to be able to analyse separately each technique from the generalized categories, where such data was provided in papers. Low-Level refactoring list contains 101 620 from 374 633 refactorings. In the Low-Level refactorings we also have the category *Other*. This category includes refactorings, the total amount of which use is less than 0.5% of the total number of Low-Level refactorings. The list of techniques caught in the category *Other* for Low-Level refactorings and their number we show in Table 4.2 on page 29.

4.2.1 High-Level Refactorings

After dividing the refactorings into two levels, we can proceed to the analysis of the data found. As mentioned above, not all articles indicated which refactoring techniques were performed automatically, and which manually. From the data where it was indicated, we found that only 1% of all the considered techniques were performed manually. The ratio of this is shown on the Figure 4.6 on page 26.

The reason for this may be that only two articles considered manual refactoring. In most cases, researchers resort to the use of tools that automatically detect the performed refactorings, and it is unknown, how thus refactorings occurred.

On the next stage we analysed the distribution of refactoring categories, in order to highlight the most used. This distribution is shown on the Figure 4.7 on page 26.

Refactoring Technique	**Amount among all papers**
Add Parameter	663
Change Method Signature	5119
Class inheritance change	304
Consolidate conditional	273
Convert Anonymous to Nested	407
Convert Local To Field	2128
Data (return) Type Change	1524
Encapsulate Field	2225
Entity Addition	38565
Entity Removal	7910
Generalize Declared Type	180
Hide Method	9
Infer Generic Type Arguments	744
Interface Implementation Change	855
Introduce Assertion	77
Introduce explaining variable	166
Introduce Factory	122
Introduce Indirection	147
Introduce Null Object	15
Introduce Parameter	545
Introduce Parameter Object	208
Remove Assignment to Parameters	61
Remove Control Flag	117
Remove Parameter	443
Repl. magic number w. symbolic	321
Replace Method with Method Object	86
Use Supertype Where Possible	152
Visibility Change	1842
Total	65208

Table 4.1: Refactoring Techniques generalized as ”Other” for High-Lever refactorings

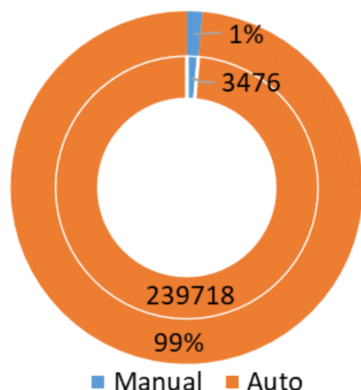


Figure 4.6: Relation between automated and manual High-Level refactorings

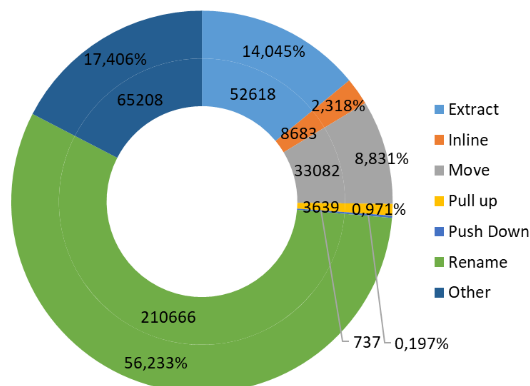


Figure 4.7: Distribution of High-Level refactorings

As we can see, the most popular among the analysed articles were the *Rename* techniques. That was to be expected, since the *Rename* is a very common operation and also that this refactoring is one of the easiest and has no hard side effects. The second most popular was the group *Other*, which included different refactorings techniques displayed in Table 4.1 on the previous page. On the third place of most popular techniques is the *Extract* category, which is used common due to simplify complex elements and our data correlates with data achieved by Silva et al. [STV16]. The *Push Down* and *Pull Up* categories were the least used due to complexity of such refactorings and the relationship of these techniques to the tasks of inheritance.

On the Figure 4.8 on the facing page we identified refactoring categories that were used only by the tools, and on the Figure 4.9 on the next page techniques, which were applied manually. The clear leader here is also the *Rename* category. Again, this is due to the fact, that the *Rename* operation is easy and is well supported in many tools for automatic refactoring and those developers, who know about this possibility, quite willingly use it. Categories *Extract* and from the group *Other* generally coincide in frequency of manual and automatic use. The biggest difference in these data is in the *Move* and *Pull Up* categories, which are manually used three times more often than automatically. Also, the *Push Down* category, despite its rare use in general, is manually applied almost 8 times more often than automatically. Such a difference can be explained by the lack of support for these techniques by the tools, lack of awareness by programmers about the possibilities of automatic refactoring, or by the preference of developers to carry out these techniques manually (because of distrust of the tools). In more detail, the use of these techniques will be discussed in the Low-Level refactoring section.

On Figure 4.11 on page 28 we show the comparative distribution of automatic and manual uses of refactoring categories. As we can see, because of the small percentage ratio between the methods under consideration, the manual methods of refactoring takes up less than 1% of each generalized technique. Only the *Push Down* category is used almost 10 times more often manually than automatically, despite the uneven distribution of manual and automatic techniques.

On Figure 4.10 on the next page we show the distribution of refactoring categories with no data about how they were used (manual or automatically). Here we can

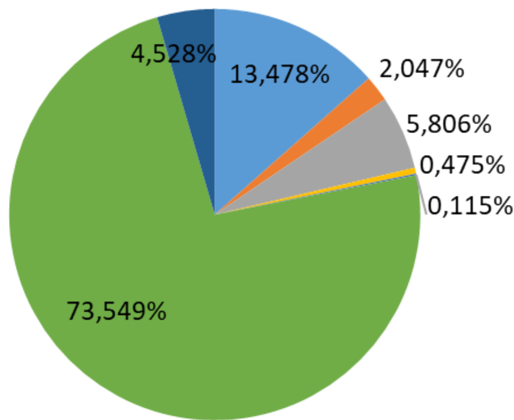


Figure 4.8: Distribution of automated High-Level refactorings

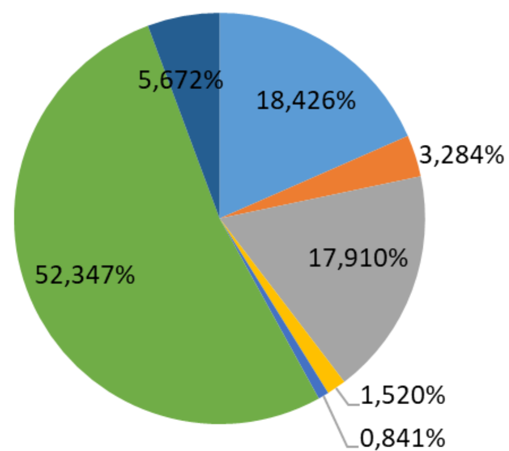


Figure 4.9: Distribution of manual High-Level refactorings

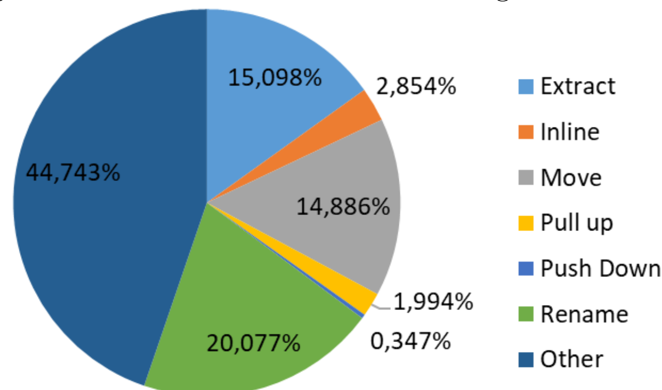


Figure 4.10: Distribution of High-Level refactorings without data about usage

see a situation in which almost half of the techniques belong to the category *Other*. This is due to the fact, that in the articles, in which it was not indicated how the refactoring is used, the more specific refactorings from the Table 4.1 on page 25 were also taken more into account. Also, since it is not clear what the distribution of types in techniques without data about use is and the refactorings from the category *Other* are less common than those, which were collected in other categories, we can assume that a sufficiently large number of them were applied manually. The reason for this could be, that techniques from category *Other* are very specific, most of them do not have a tool support and some of them are made mostly manually (for example technique *Entity Addition*). This is also indicated by the rather large percentage of *Move* category on this figure. For the rest, the data coincides with the previous distributions.

4.2.2 Low-Level Refactorings

As mentioned above, in Low-Level refactorings we attributed non-aggregated data from High-Level, the list of which is also given above. This is done because for only

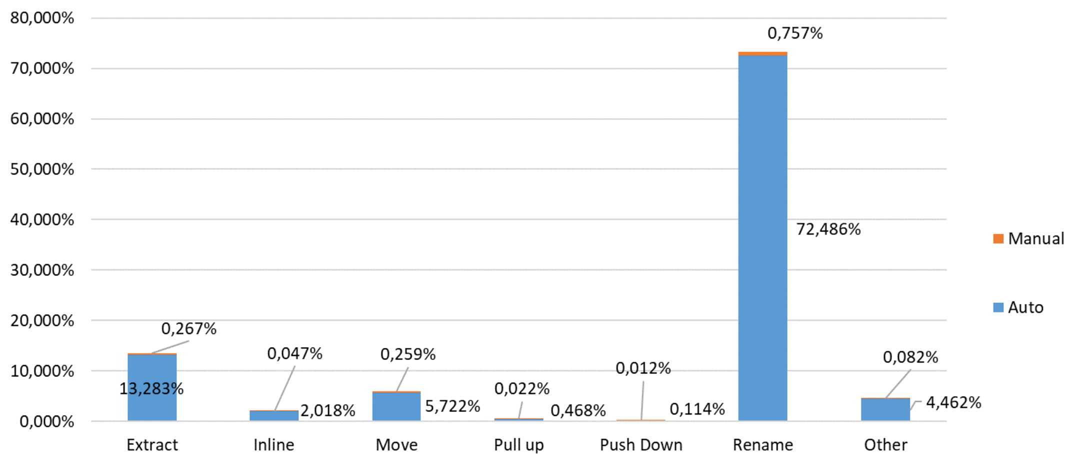


Figure 4.11: Detailed relation between manual and automated High-Level refactoring techniques

one third of all refactorings found we have data about what kind of non-generalised techniques were observed.

In the distribution of automatic and manual refactorings, we have a slightly different situation - 8% of all the techniques were performed manually. We show this on Figure 4.12. In this case, it can be explained with the fact that with detailed and not generalized consideration it is easier to find refactorings which were done manually. Also, this shows, that for searches of non-generalised techniques of refactorings were less used tools for automatic search. The smaller total number of refactoring operations considered also affects the sample. However, as mentioned above, due to the rather rare consideration of manual refactorings in articles, the distribution is not uniform, which subsequently affects the distribution of methods among all the techniques.

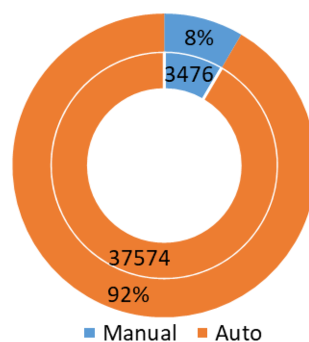


Figure 4.12: Relation between manual and automated Low-Level refactorings

On the Figure 4.13 on page 30 we show the distribution ratio of the most used Low-Level refactoring techniques. Our data reveal that the techniques *Extract Method*, *Rename Method*, and *Rename Local Variable* are used considerably more than the other techniques. This results show, that this techniques are most commonly used and main reason for this could be simplicity and need of them for frequent use during development. This results correlate with data, that was achieved by Applied

Refactoring Technique	Amount among all papers
Inline Constant	38
Move Attribute	134
Move Package	4
Move Static Member	253
Pull Up Attribute	24
Push Down Attribute	16
Push Down Field	169
Push Down Method	260
Rename Enumeration Constant	3
Rename Interface	4
Rename Package	126
Rename Type Parameter	6
Total	1037

Table 4.2: Refactoring Techniques generalized as "Other" for Low-Lever refactorings

Software Engineering Research Group at the Federal University of Minas Gerais and Silva et al. [STV16].

On the Figure 4.14 on the next page we identified refactoring techniques that were applied with the help of tools. The most common technique here is the *Extract Local Variable* and immediately following it use the *Extract Method*. The reason for this distribution may be that both of these operations are quite common by removing code smells and are well supported in many refactoring tools.

On the Figure 4.15 on page 31 we show the use of techniques that have been applied manually. Here the distribution turned out to be smoother than in the case of automatic techniques. Technique *Rename Local Variable* is the most commonly used. Moreover, techniques such as *Extract Method*, *Rename Field* and *Move Class* are also commonly used. The refactoring techniques from category *Other* are also used manually more often than automatically. We assume, that this could happen because the sampling of these techniques among the articles was small, and these data may not reflect the real situation.

More visually, the relation of automatic to manual refactoring is displayed on the Figure 4.16 on page 31. One of the main differences between automatic and manual Low-Level techniques of refactoring is, that in the reviewed articles there was not a single use of the technique *Extract Local Variable* manually, although it is automatically used in more than 35% of all cases. Despite the uneven distribution of the number of automatic and manual refactorings, the *Rename Field* and *Rename Local Variable* techniques are almost equally used manually and automatically. Based on this, we can assume that in real cases, these techniques are used more often man-

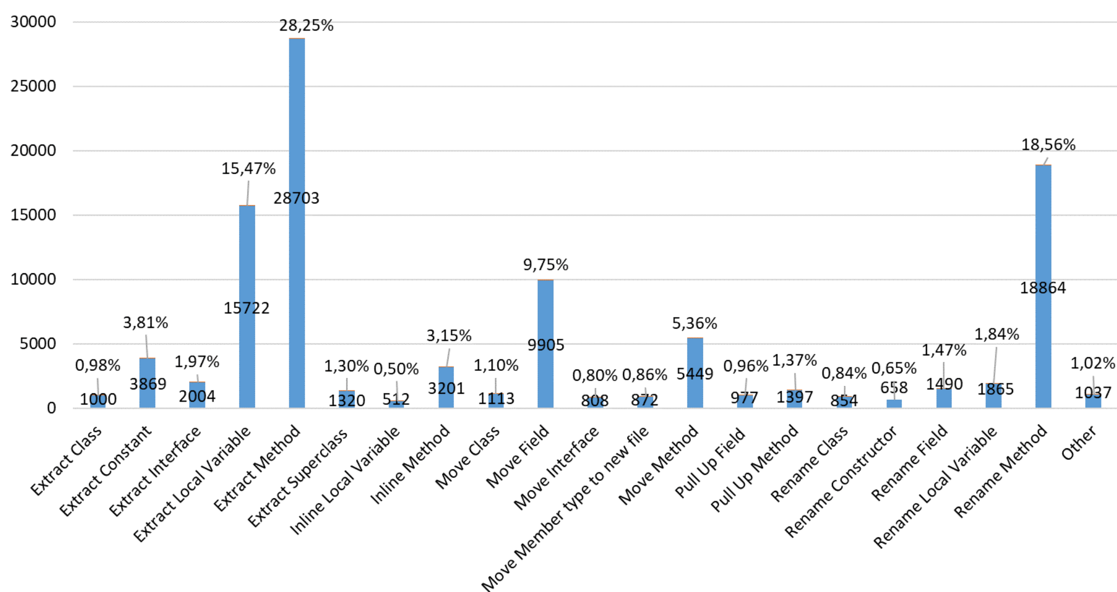


Figure 4.13: Distribution of Low-Level refactorings

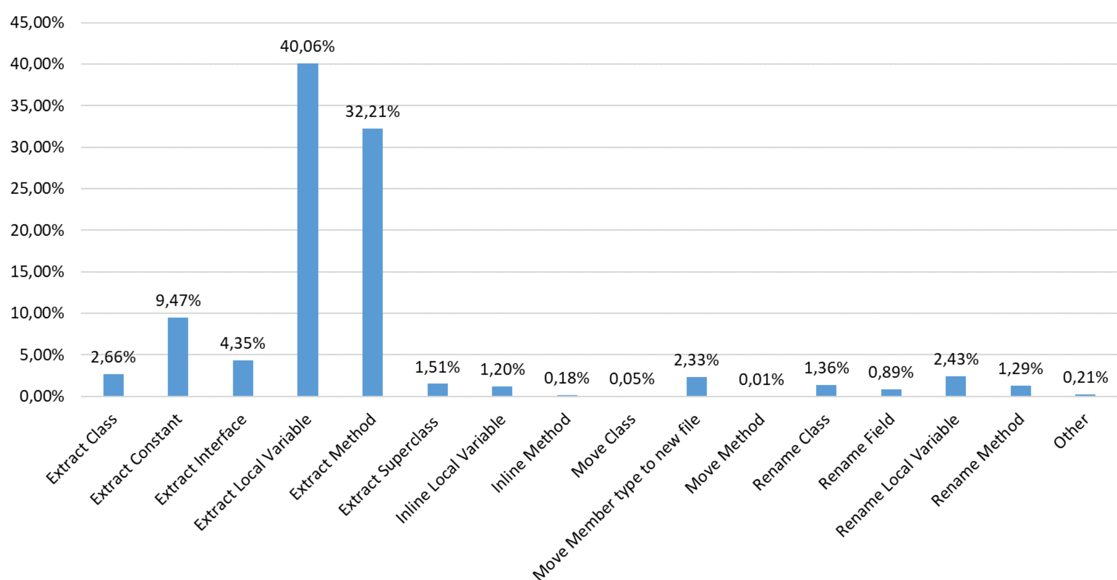


Figure 4.14: Distribution of automated Low-Level refactorings

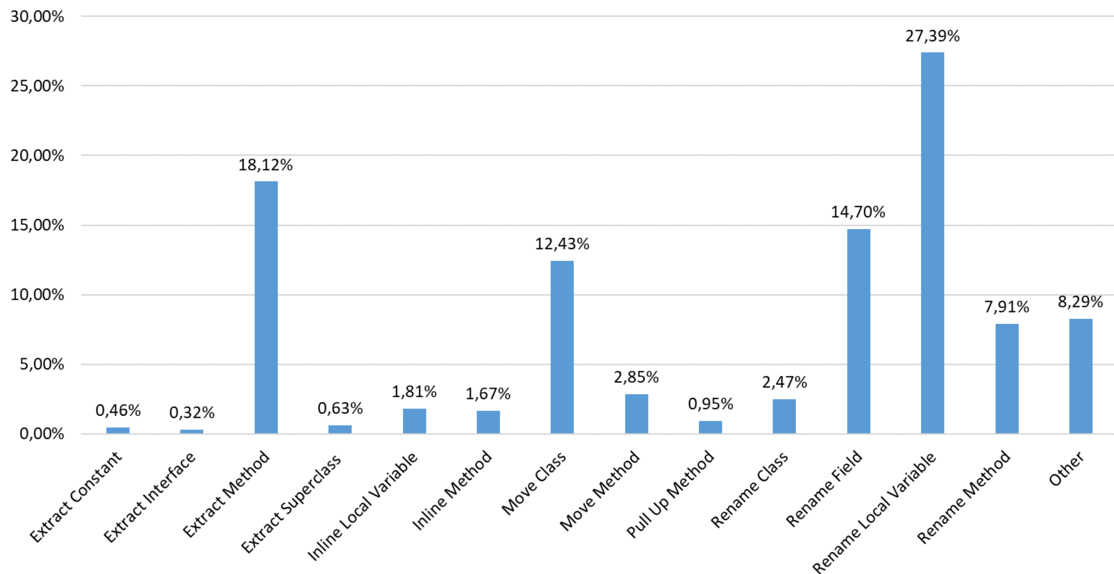


Figure 4.15: Distribution of manual Low-Level refactorings

ually than with the help of tools. We see the same in the case of the *Move Class* technique and category *Other* - in this case, manual use dominates. Techniques such as *Extract Class*, *Extract Constant*, *Extract Interface*, and *Move Member Type to a new File* are almost never used manually. This is explained by the support of techniques by refactoring tools, since this techniques have dependencies and there is a high probability of making a mistake by refactoring them manually.

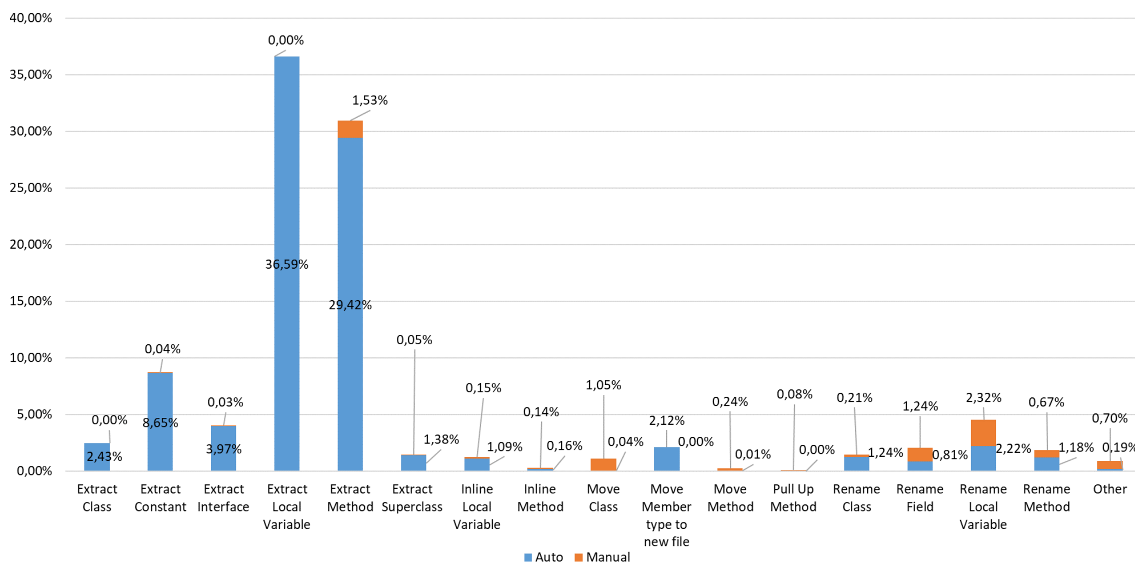


Figure 4.16: Detailed relation between manual and automated Low-Level refactoring techniques

On the Figure 4.17 on the following page we show those techniques for which it has not been explicated whether they are used manually or automatically. Comparing this diagram with others, we can notice a correlation in methods that were more often used manually. From this we can assume that it is possible, that in articles where the

type of refactorings was not indicated, manual refactorings had a large share. The distinctive point is the *Move Field* technique, which is third in use on this diagram, but was not noticed at all among the diagrams of manual and automatic refactorings techniques. This could be explained by the reason, that some researchers have not encountered such refactorings in their research, and thus, who encountered, used for refactoring detection automated tools. But, tools for refactoring detection could not consider *Move Field* technique, because strategy of moving fields refactoring lead to contradicting refactoring detections according to the strategy of moving methods [TC09].

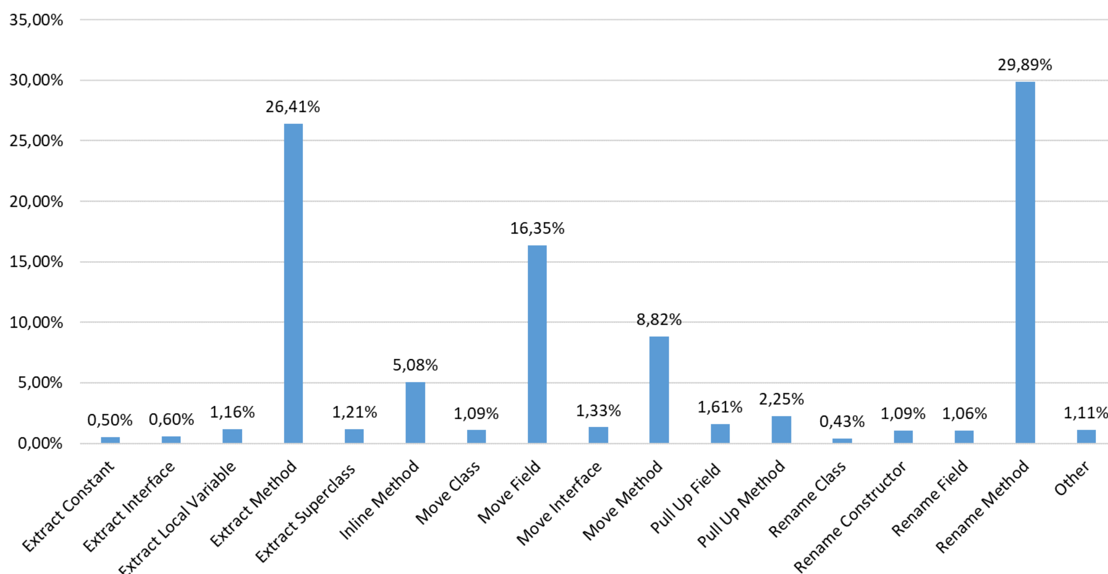


Figure 4.17: Distribution of Low-Level refactorings without data about usage

4.3 Qualitative Data

In each of the 37 articles we found information and observations from researchers about how, why, and when refactorings are (not) used. We analyse this information to find answers to our research questions and discuss them. In some articles there was a lot of such information, in others we found few. In general, in each article we looked for any notes that would allow us to answer research questions.

During analyse of articles we wrote down observations and conclusions as text fragments. We received a large number of such text fragments and the data from these fragments needed to be interpreted for our study. As mentioned in Chapter 3, for the analysis of this data we used the Open Coding technique [CS90].

This technique allowed us to highlight the most important information from the articles according to each research question. In addition, we wrote out other interesting observations from the articles of researchers who could shed light on the topic of refactoring and what role refactoring has in the development process and in software engineering in general.

As mentioned above, the first step was to read all the articles and highlight the information we are interested in regarding research questions. All this information

was entered into the data extraction form. After reading and filling out all data extraction forms we put together all the qualitative data in one table and started the analysis using open coding. During process of open coding we first analysed what was written and extracted the “code” from text information. After creating the code or assigning information to the existing code, we compiled the categories to which the “codes” created by us belonged. In the process of analysing the text sometimes we characterized a limited piece of text to other research questions than to the originally recorded on the date extraction stage. Also, we met fragments that were very extensive and informative, and which were subsequently assigned to simultaneously two codes, or even two codes of different questions.

4.3.1 RQ1: Why Refactoring is Not Used?

In the Table 4.3 on the next page we show results of open coding as codes and categories for the question ”Why refactoring is not used?”. We found eight main categories, which explain disuse of refactoring by developers, which we explain in the following.

Resources

Based on [TGA17, CXW+16]

Programmers often complain about the lack of time for refactoring. The workload and the need to create new functionality leave no time for refactoring. They also mention that the tight schedule and deadlines allow no refactoring. This is one of the simplest reasons, but studies show, that this reason mentioned by more that 40% of programmers. From our point of view it could be interesting to investigate, whether programmers would actually become involved in refactoring if they had more time, or, behind the time there are other reasons, as personal unwillingness.

Risk

Based on [TGA17, CSGG16, HRP+14, KZN14]

Programmers are afraid of refactoring because in the process of refactoring there is a high probability of creating new bugs. These bugs are likely to be missed and will not be noticed before the testing phase, and as a consequence the result of refactoring will be negative. This reason is very serious and reasonably repels programmers. Another fact is, that refactoring does not always improve source code quality. Although one of the tasks of refactoring is to remove the smells of code and improve the code itself, often after refactoring the smell-code metrics on the contrary get worse. At the same time, the deterioration and degradation of the code occurs, which breaks the concept of refactoring. Some researchers have noticed that negative refactorings happen as often as positive ones.

Complexity

Based on [TGA17, CXW+16]

Programmers often mention that they are forced to avoid refactoring because of the complexity of some refactoring techniques. Perhaps this is due to the lack of

Category	**Code**
Resources	Time
	Deadlines
Risk	Introducing new faults
	Causing externally visible changes
	Breaking something
	Degradation of code quality
Complexity	Complexity of some refactorings
	Simplicity of refactorings
Return of investments	No benefits
	No costs for refactoring
	Wasting time
Quality issues	Impact of changes
	Does not improve performance
	Lack of support
	Familiarity with code
	It still works
Management issues	No planning
	No budged
Tools issues	Lack of tools
	No tool support
	Complexity of tools
Personal reasons	Unwillingness

Table 4.3: Open Coding - Why refactoring is NOT used?

experience of the programmers themselves, in particular in refactoring. Also, we can assume, that lack of tools support for some techniques can play a side role. Some techniques are too complicated to be done manually and the tools for refactoring could not support this technique or could be to complex for use. But in general, there is a tendency that the more complex the refactoring operation is, the less often it is tried to be used. For example, we found some comment of developer, who said, that inheritance is tricky to refactor correctly [TGA17]. On the other hand, some researchers mention that some refactoring techniques, on the contrary, are too simple. Because “this is too simple” and such refactorings do not take much time, programmers postpone or do not consider it necessary to conduct such refactoring.

Return of investments

Based on [TGA17, Wan09]

Developers note that despite the obvious benefits of refactoring, there are cases in which the benefits of refactoring are less than the cost of its implementation. Programmers weigh the costs before making a decision on refactoring. If these costs are high, or the benefits are too low, the developers decide not to refactor at all.

Quality issues

Based on [TGA17, CXW+16]

Although refactoring should not improve product performance, programmers expect that refactoring would improve performance, and because of this they call it as one of the reasons for not refactoring. In this case, the role played by the awareness of the developers what is refactoring and what are its goals, since from the point of view of refactoring improving performance is not a goal. Another reason is that refactoring is not a necessity and programmers can skip the refactoring process and continue developing. Also, here it is possible to attribute the fact that very often the developers try not to touch the working code at all. Other technical reasons are the lack of support for the refactoring process and the degree of familiarity with the code. If the developer receives the task of supporting code that he did not write, for him the task of refactoring is complex. In this case, even automatic tools could not help, however using such tools is also reason for not refactor.

Management issues

Based on [TGA17, CXW+16]

Programmers note that from the point of view of management there are cases where refactoring is never planned. Also, since they do not always manage their time and are forced to act strictly according to plan, even when they want to refactor clients often do not pay for refactoring or their boss does not see the need for them. Separately, we want to mention, that although some developers say, that they do not refactor separately (the root-canal refactoring), they try to combine refactoring with other tasks (floss-refactoring).

Tools issues

Based on [TGA17]

Mentioning a problem with using tools by developers is rare among articles. But we cannot fail to note that some developers pointed out that the available tools are difficult to master, there is only small amount of the available tools and often it is very problematic to refactor automatically together with version control systems.

Personal reasons

Based on [CXW+16, LML+15]

One of the simplest reasons for not doing refactoring is reluctance from the developers side. Research participants and surveys said, that either no one wants to refactor, or that the refactoring process is not related to the work of programmers. In this case, we can again say that developers are not sufficiently aware of what refactoring is.

4.3.2 RQ1: Why Refactoring is Used?

In Table 4.4 on the facing page we show result of open coding as codes and categories for the question "Why refactoring is used?". We found five main categories, which explain main motivation for use of refactoring by developers.

Improving understanding of code

Based on [Lee11, TGSH13, PZODL17, ML06, OKS+16, AABA15, STV16, MCHH07, XS06, MH06, LML+15, CFM+16, TR03]

One of the reasons for using refactoring is the same as idea of refactoring: it improves readability and gives a chance to understand existing code. Refactoring is an exploratory process. When a programmer do refactoring, he learns more about the program itself. Programmers say, that when they refactor they need to get to the core of the written code. Refactoring allows to go deeper into the details of the program, notice the implicit things and improve the understanding of the written code. Besides, the process of refactoring is much more efficient in studying a program than the simple reading of written code. This allows to create a more accurate model of the system after refactoring. In the process of refactoring, developers make clearer the names of variables and methods, decompose and simplify complex and long methods and reduce the level of code abstraction. Also, refactoring facilitates subsequent documentation, developer support for the code, reusability, flexibility, and also the effectiveness of the refactored program. Ultimately, the internal attributes of the quality system will either be improved or remain untouched.

Extension of system

Based on [TGSH13, KZN14, LML+15, STV16, MCHH07, MH06, TR03, PZODL17]

In addition to improving the understanding of code, programmers point out that refactoring facilitates further development. After refactoring, it is much easier to add a new feature, new functionality and further expansion of the system is becomes easier. In some cases, it is possible only after refactoring to use certain functions, such as recursion or overriding. The refactoring process is used to simplify the program and make easier future development.

Category	**Code**
Improving understanding of code	Understanding
	Readability
	Following naming sheme
	Documentation
Extension of system	Easier future development
	Improving extensibility
	Adding new features
	Future extension
	Add flexibility
	Easier future change
Software quality	Backward compatibility
	Improving design
	Improving quality attributes
	Improving performance
	Improving structure
	Reuse
	Removing code smells
	Removing bugs
	Reducing duplicates
	Tactic
Improving maintenance	Improving testability
	Improving comprehensibility
	Allowing parallel work
Personal reasons	Boosting morale and motivation
	Personal needs
	Assigned Task
	Responsibility
	Often use
	Rewarding for better code
	Personal knowledge
Time	Reducing time to market
	Reducing time to release

Table 4.4: Open Coding - Why refactoring is used?

Software quality

Based on [XS06, MH06, DAF12, PZODL17, HRP+14, MHPB09, CXW+16, MH09, KCK11, GDMH12, LLXG14, MHB07, LML+15]

One of the largest categories that programmers refer to. From a technical point of view, refactoring reorganizes and redistributes information among parts of the system. Programmers notice, that refactoring facilitates the task of backward compatibility, improves system performance, enable quicker enhancement, improves the structure of the code, removes duplicates, removes code smells and design defects. Developers often use refactoring as a technique for removing bugs or to enable bug fixes. Moreover, in programs where refactoring is carried out, fewer bugs in release are noted. The researchers note that most refactoring operations are applied to code elements that contain critical attributes. In industrial companies it is noted, that refactoring is one of the main motivators for optimizing program execution time, increase reliability and robustness. Also, current reuse is often referred to as the need for refactoring.

Improving Maintenance

Based on [KZN14, PZODL17, STV16]

As well as improving the readability of the code and reducing the number of bugs, refactoring improves testability. Some researches have pointed out that refactoring improves comprehensibility and maintainability of code, which by himself precedes a bug fixing. Also, the researchers say, that testing code without prior refactoring becomes much more difficult.

Personal reasons

Based on [Wan09, LML+15]

Unlike reluctance to refactor, the researchers note that high self esteem motivates to refactor. Developer' belief, that he is capable of performing in a certain manner to attain certain goals is one of the important facts why refactoring is performed. In addition, the responsibility of the developer, his inner desire to create high-quality code and the authorship of the code play a role in whether to refactor. Researchers point out that if in a team of developers, organizations and communities, refactoring is a normal practice, then the desire of developers to follow accepted standards and not being treated as isolated ones will push them to refactor. Another fact is that developers often "punished" for their quality issues, and to avoid this, programmers spend more time on code quality. Researchers have noticed that for some programmers refactoring is a habit and having a chance to refactor increased their motivation and satisfaction. Another researchers say, that refactoring activities driven more by intuition than metrics.

Time

Based on [KZN14, SNFG14]

The important reason, which rather even makes it necessary to refactor. Immediate concrete, visible needs of changes that they must implement in a short term require

refactoring. Another factor is the relationship with the return of investment, when the development should be carried out with the least expenditure of resources, because refactoring reduce the time, which later needed for testing. Also, researchers note, that refactoring reduces the product release time.

4.3.3 RQ2: When Refactoring is Used?

According to second research question, in Table 4.5 we show the results of open coding to selected studies. We mention two main categories.

Planning

Based on [KZN14, LML+15, CXW+16, LLXG14, AABA15, NCV+13]

Developers have different opinions regarding the time for refactoring planning: some of them plan refactoring, other never plan or do it only as needed. Another interviewed say, that they do not know how to answer on the question. The experience of programmers shows that refactoring is often not a strictly planned task. The researchers note that refactoring is planned as a separate task only in half of highly efficient teams. Totally, less than 25% of teams plan refactoring. Unplanned refactorings appear to be necessary in those cases, when either the technical side becomes very high (code smells, unmaintainability of system), or when it is not particularly urgent to add new features or fix bugs. In most cases, floss refactoring is not planned at all and is part of system redesign or regular system changes. Researchers point out that the need for refactoring often appears as part of maintenance activity. Moreover, small refactorings, such as changing the name of a method or moving a code, are not perceived by developers as a refactoring activity. Thus, we can once again note the lack of awareness of programmers in the term refactoring and its principles.

Category	**Code**
Planning	Planned activities
	Not planned activities
How often	Common
	Rarely
	By request

Table 4.5: Open Coding - When refactoring is used?

How often

Based on [CXW+16, LML+15, MCHH07]

Researchers point out different data on how often refactoring is used. Interview results reveal that developers spend about one-fifth of the time on refactoring. On the other hand research of repositories and experiments show, that ratio of refactorings to all activities is 0.1%. It is impossible to say unequivocally how frequent or rare are the refactoring of activities. Once again we could admit, that this greatly

depends on the developer's experience and planning task. So, many developers conduct refactorings across all milestones and also during time to fix bugs and clean up code without the responsibility to add new features. If programmers are not in a hurry to introduce new features, refactoring is done more often and vice versa. Some refactorings are carried out only upon request from a manager or customer, which indirectly indicates a lack of refactoring planning and awareness at all.

4.3.4 RQ3: How Refactoring is Used?

In Table 4.6 on the facing page we present the results of open coding according to third research question. We mention two main categories.

Methods

Based on [KZN14, NCV+13, CGM+17, PZODL17, HRP+14, MHPB09, CXW+16, MH09, KCK11, GDMH12, LLXG14, MHB07, LML+15]

Researchers note that in general, programmers use several standard basic refactoring actions, but interleave those actions in a variety of ways. As mentioned in the second chapter, there are two methods of refactoring: manual and automatic. Studies show that despite the existence of automated refactoring tools, more than half of programmers do refactor manually. Only about 35% of programmers in general try to use automatic methods. These data are different from those that we obtained at the stage of analysing quantitative data. We can assume, that this happened due to the fact that in the articles we used for quantitative analysis automatic refactoring was considered more frequently. The reasons, why only one third of developers use refactoring tools, is that in many cases programmers do not trust automatic methods in the case of complex refactorings, refactoring is often trivial and can be done manually, for certain IDE there are no suitable tools, and if they are, programmers are not familiar with them. If an IDE has such tool support, more than half of programmers in research tend to use them in a certain manner. Although many developers point out that there is no tool that really helps in refactoring. The use of tools itself depends on the programmer's experience. Thus, programmers with experience more than 10 years used more manual refactorings, and programmers with experience from 5 to 10 years used automatic possibilities. In general, the popularity of automatic refactorings does not reflect the popularity of refactoring in general. In some cases, developers combine manual refactoring with partial use of tools, thereby performing combined refactoring. Researches show that this type used by approximately 7% of developers and again depends on the availability of automated tools.

Refactoring types used

Based on [CFF+17, STV16, MHPB09, GDMH12, MHB07, LML+15, MH09, KZN14, NCV+13]

Among all the articles we can definitely say that floss refactoring is the dominant type. According to various sources, the frequency of the root-canal refactoring is less than 25% of all refactorings activities and the frequency of floss refactoring in some projects reaches 97%. The reasons for this is that refactoring is much

Category	Code
Methods	Manual
	Automated
	Combined
Refactoring types used	Floss refactorings
	Root-canal refactorings

Table 4.6: Open Coding - How refactoring is used?

easier to do in conjunction with other tasks, such as creating new features, bug fix, maintenance or any other developers activity. Here we can also add the lack of refactoring planning in general and the fact that many programmers do refactoring unknowingly. Researchers point out that floss refactoring is more efficient, frequently used and more likely to continue to be used. Floss refactoring is also often an integral part of agile software projects. Root-canal refactoring is not so common, but in the future it is expected that it will be used more often.

4.3.5 Other Findings

During our analysis we found also another data, that is strong related to theme refactoring. In Table 4.7 on the next page we show the results of open coding to other interesting data according to refactoring.

Problems of refactoring

Based on [KCK11, PZODL17, CSGG16, OKS+16, MCHH07, KZN14, LML+15, SNFG14, TGA17, CXW+16, Gil15]

One of the first problems of refactoring is creating bugs. Incorrect refactoring, whether it be a floss refactoring or a root-canal, will certainly create a bug, which later need to be fixed. Researchers note that after applying refactoring, there is a short period of time in which the number of bug fixes increases. Also, the number of refactorings in conjunction with the bug-fixes increases immediately before major version releases. Moreover, often refactoring is a necessity right in front of bug-fixing activities.

Some researchers have noticed, that there no correlation between design errors and the amount of refactoring activities. It means, that programmers do not pay attention to the code smell metrics. Despite this, about 80% of all the refactorings are applied to elements with a smell and that probably done unknowingly. However, code smells metrics are not reliable indicators of whether refactoring will be performed. Neither the role of the programmer nor the background gives a clear answer whether this or other part of the code will be refactored. The use of refactoring creates almost the same amount of smells as removes them. About two thirds of all refactorings are “neutral” - they do not change the code smell density. At the same time, one third of all refactorings are ”stinky”, that means, they are negative refactorings and create new smells. Only about 10% of refactorings really remove

Category	**Code**
Problems of refactoring	Incorrect refactorings
	Dependency to bug-fixes and creating new bugs
	Dependency to smells and nor removing them
	Dependency to releases
	Understanding of term ”refactoring”
	Dependencies of programming language
	Meaning of metrics
Refactoring activities	Changing quality of code
	Using tools for refactoring
	Reuse and maintenance
	Difference of refactoring types
	Support of refactoring
	Smell-elements
Search for refactorings	Commit logs as indicator of refactorings
	Problems with tools for detection of refactoring
Personal	Accepting of refactorings
	Dependencies to personal experience
	Dependencies to type of a project

Table 4.7: Open Coding - Other findings

code smells. Moreover, the researchers note that the number of negative refactorings in the root-canal type is surprisingly high.

The academic definition of the term refactoring among developers is often perceived as not entirely correct - developers often use refactoring with other activities, which still change the functionality of the program. About 40% of developers did not mention preservation of behavior, semantics, or functionality in their refactoring definition at all. Also, the definition of refactoring from developers does not quite match with the definition of Fowler - for more than one-third of developers, refactoring is part of normal software development.

In our review, Java is the dominant programming language. Researchers who studied refactoring to several programming languages have noted, that the most dangerous refactorings are refactoring for basic Java functionalities.

Refactoring activities

Based on [CFF+17, MHPB09, CS12, CXW+16, KZN14, TR03, LLXG14, MH09, LL16b]

Any type of refactoring improves the internal quality indicators of attributes when considering at least one metric about attribute. Moreover, floss refactoring improves these attributes even if the programmer had no specific task to improve one or another attribute. The use of manual and automatic refactoring is different, it is noted by the programmers themselves. Different types of refactoring use different techniques. This is confirmed by our analysis of the use of different types of refactoring by High-Level refactorings on Figure 4.11 on page 28 and Low-Level refactorings on Figure 4.17 on page 32. Moreover, different refactoring techniques are applied differently in the root-canal and floss refactoring and, accordingly, have a different number of applications.

The developers of refactoring tools mainly use other refactoring techniques compared to the users of tools. This difference in views may influence development direction of refactoring tools and users of tool may have not what they actually want. Programmers also mentioned, that machine support for refactoring is highly desirable.

An important part of using refactoring is documentation. Researches point out, that it is extremely important to document in detail the procedures of applied refactoring, since this facilitates further development and support not only among programmers, but also other team members.

Despite the fact that code smell metrics are a poor indicator of whether refactoring will be carried out, programmers note that when they deciding where to start refactoring, they try to find the most smelly part of the program. Thus, code smells do play a role in refactoring.

Search for refactorings

Based on [MHPB09, CS12, NCV+13]

Using tools to search for refactorings makes easy to find refactorings for researchers. But unfortunately, the messages recorded in the commit log very often do not allow

either the tools no manual analysis to identify the presence of a refactoring in the commit. Tools for refactoring searching can find only a High-Level refactorings. Since about half of all refactorings are not High-Level, a large number of Low-Level refactorings go unnoticed. Many Low-Level refactorings in addition are "shadowed" (refer to floss refactoring) and it is also not possible to determine them correct with help of tools. Based on this, there is a need in tools that could search in more detail for refactorings and the programmers' task is to write in more detail in the commit log what has been done.

Personal

Based on [TGSH13, LML+15, CXW+16]

In some projects, the decision on refactoring is applied by one person. Moreover, if the programmer needs to inform about the need to refactor, it becomes a real problem. Since refactoring is something that does not change the functionality of the program, for the customer there are no direct benefits from refactoring and it is very difficult to explain to the customer the need for this procedure. There is also a dependence on the experience of developers. The difference in the experience of the development teams shows that if for an experienced group refactoring can be combined with other tasks (they make floss refactorings), then inexperienced groups are more focused on continuing development without paying attention to the need for refactoring. Also, the more experienced group's focus on extensibility and maintainability and trying to take account of the future changes. This makes them decide to refactor earlier. Generally, awareness of refactoring should become a skill, which can be acquired from experience. The researchers point out, that well-executed refactoring plans are signs of high-quality projects.

4.4 Discussion

In the course of the analysis, we found somewhat contradictory data according to refactoring process. In this section we discuss them and try to name their causes.

One of the main contradictions regarding refactoring is the time to use it. Some programmers do not use refactoring, because it takes a lot of time, which they do not have. Other developers mention that using refactoring can reduce testing and bug-fix time and speed up development. Regarding this arises the question: what is actually better in terms of time? We assume that the speed of refactoring activity directly depends on the experience of the programmer, the development language, planning and support of refactoring in IDE. The development language plays an important role how difficult it is to refactor and how well it is supported by automatic tools. Lack of refactoring planning allows programmers not to consider refactoring at all as part of development and not to deal with it. The developer's experience and how familiar he is with refactoring is one of the main reasons for whether refactoring will be used or not. As already mentioned, if programmers would have the opportunity to refactor, many developers will deal with it. For this reason, it is very important to plan refactoring in the development process.

Another controversial fact is the difficulty of refactoring. Some techniques are very complex, others, on the contrary, are simple. Both cases are the reason for refusing

refactoring. If the reason for refusing complex refactorings can be understood, then it is very strange that the simplicity of refactoring is also the reason for not doing refactoring. Here, again, plays a role what programming language is considered and the experience of the programmers. Programmers with less than 5 years of experience tend to refuse to refactor by naming both reasons. For them, further development and addition of new functionality is much more important. They also mention, that the tasks of refactoring are not clear and often look like a waste of time. In addition, because automatic tools are difficult to learn and in some cases do not help in conducting refactoring, the propensity of programmers to refuse refactoring increases. To eliminate this uncertainty, it is necessary that programmers understand the importance of simple and complex refactoring for the entire system. It is also very important that programmers understand how to properly refactor, so that its consequences will be not negative.

From this point of view, the technical side of refactoring also has some contradictions. Using refactoring allows you to improve the system, reduce development time, bug fixes, testing, and facilitates further support. But not all refactorings have a positive effect. Many of them, on the contrary, create new bugs, add code smells, increase time costs and generally worsen the system. This raises the question: why does refactoring have a negative effect? In most cases, refactoring is a personal initiative of a programmer, his task and the decision on his venue are often taken by intuition, and not by the code smell metrics. The programmer and his experience in development and refactoring affects whether the refactoring will be positive or negative. While refactoring some programmers expect improvements of system performance, although this is not the purpose of refactoring. From this we can call another contradiction in the field of refactoring - its definition. Despite a clear academic definition, goals and objectives, the definition from developers often does not coincide with them. For many programmers refactoring is not a separate development process, but a part of it. Developers do refactor where they see fit, and not where it is a necessity on the part of the system. Why among programmers there is no clear definition of refactoring and not everyone understands unequivocally what it is and why to conduct it? Again we come to the fact that the personal experience of the developer affects how refactoring will be carried out.

The use of tools also has some conflicting data. As mentioned above, programmers with 5-10 years of experience use automatic refactoring more often than the more experienced ones. We can assume that this is due to the fact that experienced programmers know exactly how to conduct this or that refactoring and do it manually based on previous experience. Perhaps due to the fact that the tools appeared relatively recently, many experienced programmers are not so open to new methods and capabilities. Other programmers are not so conservative, they are interested in the capabilities of the tool and they are trying to learn and use them. Automatic tools, which should facilitate the task of refactoring and help programmers to carry out it faster and more efficiently often fail to cope with their task and remain ignored. For some programming language there is a lack of such tools.

The most controversial aspect of refactoring is its use by differently experienced groups of programmers. If in experienced groups of developers refactoring is a standard practice, it is carried out more often and more successfully and they have

already developed experience with refactoring and its definition, then by programmers with medium and small experience are all rather ambiguous. As has already been said, many developers do not know a clear definition and goals of refactoring, do not pay enough attention to it and try to avoid it whenever possible. Programmers with less experience most often do negative refactoring. Some developers do not realize that in the development process they sometimes do floss refactoring. The task of refactoring some developers is motivating, for others it is completely unpleasant work. Not everyone understands that refactoring is part of development, which should not be combined with other development processes. Not everyone knows where it is best to start refactoring, how to conduct it without worsening the system. All this suggests that the experience of the programmer determines how, when and how the refactoring will be carried out. The more experienced a programmer, the more likely that he will be better understood and do refactor. Therefore, taking time to refactor even among young developers is a necessity for developing a good tone of programming and creating good software.

4.5 Study Limitations

The entire review was conducted by one person and the result was reviewed by another. There was no parallel or collaborative analysis of data or comparison of intermediate results. Despite, that we tried to be unbiased, this SLR may be somewhat subjective.

The 37 articles, which were obtained in the review, are not the ideal number. Not all found articles had open access. Perhaps, some articles were missed, which could not be found by our search strings or were in other libraries. Relatively few articles contained interviews or surveys.

Many articles dropped out at the data extraction stage, although they were suitable for quality assessment. This may mean that for this topic, questions for quality assessment should be chosen more carefully and the quality of the found articles should be thoroughly checked.

The dominant language considered in the review is Java. Despite the availability of articles about other languages, relatively few were submitted. Therefore, this review applies more to Java language.

Considered refactorings in most cases were found through the automatic search tools. Since such tools have a certain accuracy and in some cases are not able to accurately find refactorings, the data on the number of used techniques are not completely accurate and have a certain error.

Data extraction is in itself a very complex process that requires great concentration and attention to each article. Despite the availability of the data extraction form and an attempt to fully cover all aspects of interest to us, it is not excluded that some data was missing or incorrectly highlighted.

By analysing text data using open coding techniques were made generalizations to display the essence of the found information. These generalizations can also be subjective or, to some extent, not quite correctly interpreted.

5. Related Work

In this chapter we present other studies, which are related to our area of research.

Abebe and Yoo have reviewed the different angles of refactoring to understand and extract the software refactoring knowledge [AY14]. This work has a large sample of articles, the study was carried out qualitatively and brightly shows the current directions of the study of refactoring, its trends, opportunities, challenges, gaps, tools and connection with other areas of software engineering. Also, the paper presents a qualitative classification of general study directions in area of refactoring. The main difference of this work to our thesis is that this work is a more generalized summation of directions and knowledge of refactoring today, although in our case we concentrate on the practical use of refactoring.

Wang has observed the main motivators of refactoring based on an interview of 20 people [Wan09]. The data from this article are used also by us in our work. Despite the small sample of respondents, the data still correlate with other articles that were found for our work. The main difference to this thesis is that this article describes the data obtained only from the interview and no comparison with other data has been made.

Tsantalis et al. conduct an empirical study on the use of refactoring based on three open source projects [TGSH13]. The article describes in detail the difference between refactorings for a test code and for a production code, when a greater number of refactoring activity occurs and the reasons for using certain refactoring techniques. The main purpose and motivators for refactoring are described in detail. The difference to out work is that their research focuses more on refactoring activities and the individual contribution of developers on this activities.

Vakilian et al. done a good research on the use of refactoring tools [VCN+12]. The main goal was to understand why automatic tools are not used. For this purpose were used tools for actively monitoring the programmer's activity and logging his actions regarding both types of refactoring, manual and automated. This data were analysed and compared. In addition, participants were interviewed, in which specific data were obtained on certain refactoring activities which were observed. As a result

of the article, conclusions were drawn about the non-use of automatic refactorings and the reasons for this. Despite the fact that the main goal was to analyse the use of automatic tools, the article provides comprehensive and highly accurate data on the use of refactoring and motivation as a whole. The main distinction to our work is that in our work we do not concentrate us only on research of the use of tools and study the refactoring use more generally.

Silva et al. were analysed the main motivators of using certain refactoring techniques [STV16]. The article explored the Github repositories to search for refactorings and after each refactoring found, the programmer was asked why he did it. The results of the article cover the motivations for 12 types of basic refactoring techniques, comments from programmers, and highlighting the most frequently used techniques. Also, data on the use of automatic or manual refactorings are given and the main reasons for this are indicated. This article contains a large number of comprehensive data and this data is actively used in our work. The difference to out work is that they empirically investigated the actual motivation of applied refactorings and their research does not affect the issues of when and how refactoring is conducted.

6. Conclusion

In this thesis we showed the motivation for our research and its main stream. We have described what refactoring is, its techniques, how it can be used and the main possibilities for refactor. We also described how a systematic literature review is conducted and, based on it, described the protocol to our review.

In the protocol to review we described the planning of our review. We used the PICOC criteria to identify research questions, described the process of searching for articles and the snowballing technique. We have indicated the sources for the search and the search queries for finding articles. For selecting primary studies we described the criteria for including or excluding studies and created the quality assessment. To obtain necessary data we described procedure of data synthesis and, according to this, data extraction form and open coding techniques.

Following the protocol, we conducted a search for articles based on the selection criteria. In addition, snowballing was conducted on the articles found and articles not previously found were highlighted. After searching and snowballing we achieved 73 articles. All articles were checked according to the quality assessment and 22 studies left our review after this stage. Moreover, on the stage of data extraction some articles were also excluded. In the end, 37 papers remained for our review.

In the results of our work, we analysed the articles and their meta data, indicated the types of selected articles, the total number of study participants among all papers, their experience, the number of projects reviewed, the amount of code analysed and considered programming languages. We showed the quantitative data regarding refactoring techniques that were found. This data was divided into High-Level and Low-Level refactorings. Also we observed the data of manual and automatic use of refactoring separately. We identified the most used techniques, described the possible reasons for their widespread use and explained the heterogeneity in the data of the use of manual and automatic refactorings.

To answer research questions, we collected qualitative data from each article and analysed them based on the open coding technique. All collected data according to each research questions was divided into short categories. Each category represents

some aspect of research question. Every category was described in detail according to founds of researches and personal opinions and comments of developers. For each category we tried to explain the possible causes of its occurrence and how to deal with it. Also, we described other findings obtained by us during our review and which are interesting to the area of refactoring.

Summing up, refactoring is not used mainly due to lack of resources, high risks, difficulties with tools and the software itself, poor management, accounting for return on investment and personal reluctance. Refactoring used for reasons of improving the understanding of code, its quality, facilitating development and support, as well as for personal interest and responsibility of programmers. Refactoring is mainly used not according to plan and in many cases refactoring planning reflects high-quality projects. The distribution of frequency of refactoring activities depend on the project management and developer experience. Refactoring is not just a separate process. Often it is the part of adding new functionality. Also, some programmers are not limited to manual refactoring, sometimes they refactor with the tools and even combining automatic and manual refactoring.

The distribution of articles between research questions shows that there are relatively few studies which answers the question ”Why refactoring is not used?”. In most articles researches are trying to show positive aspects from refactoring. Also, the questions ”when refactoring is used” and ”how refactoring is used” are relatively not often investigated.

At the end of our thesis, we discussed the conflicting data and their causes, the limitations of our work and described the related work.

7. Future Work

In this chapter we briefly show the directions of future work according to our study.

There are several future research directions to improve the results presented in this thesis. One of the main ones could be similar to our review, but with a larger sample of articles and, in particular, aimed at a greater number of different programming languages. From this appears the need for more empirical researches of refactoring use that would more fully show motivation, reasons, the use and effect of refactoring. Also, it would be important to conduct interviews or surveys involving a large number of developers with varying degrees of programming experience who would answer questions about the use of refactoring.

Parallel to this, to improve the results of work in the field of refactoring is necessary to further develop automatic tools that could find a greater number of refactorings with greater accuracy in version control systems.

We can not overlook the need for further development of automatic refactoring tools that would be highly efficient, easy to learn, to use and would really help developers in refactoring tasks.

One of the more global directions could be a study in which it will be shown why there are ambiguous opinions of programmers about the term refactoring and its use and how this situation could be changed to be better, by popularizing refactoring among all groups of developers.

A. Appendix

Study ID	Name	Year	Source
Ec01	Detection Technology and Application of Clone Refactoring [YDL18]	2018	ACM Digital Library
Ec02	Gathering refactoring data a comparison of four methods [MHBDP08]	2008	ACM Digital Library
Ec03	Measuring refactoring benefits a survey of the evidence [OCYC16]	2016	ACM Digital Library
Ec04	Refactoring Patterns, Practices for Daily Work [LL16a]	2016	ACM Digital Library
Ec05	Representing Refactoring Opportunities [PPA+09]	2009	ACM Digital Library
Ec06	The Evolution of Knowledge in the Refactoring Research Field [OPMT15]	2015	ACM Digital Library
Ec07	A Discussion of Refactoring in Research and Practice [BGA+04]	2004	CiteSeerX
Ec08	Challenges of Refactoring C Programs [GJ02]	2002	CiteSeerX
Ec09	Towards Concurrency Refactoring for x10 [MFM09]	2009	CiteSeerX
Ec10	A Survey of Software Refactoring [MT04]	2004	IEEExplore
Ec11	Effective software refactoring process [KCOV18]	2018	IEEExplore
Ec12	An empirical assessment of refactoring impact on software quality using a hierarchical quality mode [SL11]	2011	Google Scholar

Study ID	Name	Year	Source
Ec13	Extracting Refactoring Trends from Open-source Software and a Possible Solution to the 'Related Refactoring' Conundrum [AHC06]	2006	Google Scholar
Ec14	Refactoring: Emerging Trends and Open Problems [MBVD03]	2003	Google Scholar
Ec15	Refactoring in the Presence of Aspects [Wlo03]	2003	Google Scholar
Ec16	Trends in Java Code Changes: The Key to Identification of Refactorings? [CHJ+03]	2003	Google Scholar
Ec17	An empirical study of refactoring decisions in embedded software and systems [DAF12]	2012	Science Direct
Ec18	A Study on Quality Improvements by Refactoring [Boi06]	2006	ACM Digital Library
Ec19	Refactoring Myths [HO15]	2015	ACM Digital Library
Ec20	An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks [FSP+13]	2013	CiteSeerX
Ec21	Experience-Based Refactoring for Goal-Oriented Software Quality Improvement [RR04]	2004	CiteSeerX
Ec22	"Refactoring" Refactoring [JBY17]	2017	IEEExplore
Ec23	A Field Study of Refactoring Challenges and Benefits [KZN12]	2012	ACM Digital Library
Ec24	A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability [KHFG16]	2016	ACM Digital Library
Ec25	Analyzing Students' Software Redesign Strategies [SPB16]	2016	ACM Digital Library
Ec26	Refactoring Planning and Practice in Agile Software Development: An Empirical Study [CXW+14]	2014	ACM Digital Library
Ec27	Seven Habits of a Highly Effective Smell Detector [MHB08c]	2008	ACM Digital Library
Ec28	The Need for Richer Refactoring Usage Data [VCN+11]	2011	ACM Digital Library

Study ID	Name	Year	Source
Ec29	Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method [MHB08a]	2008	CiteSeerX
Ec30	Does Refactoring Improve Reusability? [MSAS06]	2006	CiteSeerX
Ec31	Refactoring: Current Research and Future Trends [MDB+03]	2003	CiteSeerX
Ec32	An Initial Study on Refactoring Tactics [LGN12]	2012	IEEExplore
Ec33	Refactoring-Aware Code Review [GSWMH17]	2017	IEEExplore
Ec34	An Empirical Study on the Impact of Refactoring Activities on Evolving Client-Used APIs [KOGI17]	2017	Google Scholar
Ec35	Comparing Approaches to Analyze Refactoring Activity on Software Repositories [SGMHJ13]	2013	Google Scholar
Ec36	Refactoring Tools and Their Kin [Ste15]	2015	Web Of Science

Table A.1: All Excluded Studies

Study ID	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Score	Selected
P01	◆	◆	◆	⬙	⬙	◇	⬙	◇	4,5	Yes
P02	◆	◆	◆	◆	◆	◇	◆	◆	7	Yes
P03	◆	◆	◆	⬙	◆	◆	⬙	◆	6	Yes
P04	◆	◆	⬙	⬙	⬙	◇	⬙	⬙	4,5	Yes
P05	◆	◆	◆	◆	◆	⬙	⬙	⬙	6.5	Yes
P06	◆	◆	◆	◆	◆	⬙	◆	◆	7.5	Yes
P07	◆	◆	◆	◆	◆	◇	◆	⬙	6.5	Yes
P08	◆	◆	◆	◆	◆	⬙	◆	◆	7.5	Yes
P09	◆	◆	◆	◇	⬙	◇	⬙	⬙	4.5	Yes
P10	◆	◆	◆	⬙	⬙	⬙	◆	⬙	6	Yes
P11	◆	⬙	◆	◆	◆	⬙	◆	◆	7	Yes
P12	◆	⬙	◆	◆	◆	⬙	◆	◆	7	Yes
P13	◆	⬙	◆	◆	⬙	⬙	◆	◆	6.5	Yes
P14	◆	◆	◆	◆	◆	⬙	◆	◆	7.5	Yes
P15	◆	⬙	⬙	◆	⬙	⬙	⬙	⬙	5	Yes
P16	◆	⬙	◆	⬙	◆	◇	⬙	⬙	5	Yes
P17	◆	◆	◆	⬙	◆	⬙	◇	⬙	5.5	Yes
P18	◆	⬙	◆	◆	⬙	◇	◇	⬙	4.5	Yes
P19	◆	◆	◆	◆	◆	⬙	◆	◆	7.5	Yes
P20	◆	◆	◆	◆	◆	⬙	◆	⬙	7	Yes
P21	◆	◆	◆	◆	◆	⬙	⬙	◆	7	Yes
P22	◆	◆	◆	⬙	⬙	◇	◆	◆	6	Yes
P23	◆	⬙	◆	◆	⬙	◇	⬙	⬙	5	Yes
P24	◆	◆	◆	◆	⬙	◇	⬙	⬙	5.5	Yes
P25	◆	⬙	◆	◆	⬙	⬙	◆	◆	6.5	Yes
P26	◆	◆	◆	◆	◆	⬙	◇	◆	6.5	Yes
P27	◆	⬙	◆	⬙	◆	◇	◆	⬙	5.5	Yes
P28	◇	⬙	⬙	⬙	⬙	⬙	◆	◆	4.5	Yes
P29	◆	◆	◆	◆	◆	◆	◆	◆	4	Yes

Study ID	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Score	Selected
P30	◆	◆	◆	◆	◆	⬙	◆	⬙	7	Yes
P31	◆	◆	⬙	◆	◆	⬙	⬙	◆	6.5	Yes
P32	◆	⬙	⬙	◆	◆	◇	◇	◇	4	Yes
P33	◇	◆	◆	⬙	◆	⬙	◆	⬙	5.5	Yes
P34	◆	⬙	◆	◆	◆	⬙	⬙	⬙	6	Yes
P35	◆	⬙	◆	⬙	⬙	◇	⬙	⬙	4.5	Yes
P36	◆	◆	◆	◆	⬙	⬙	◆	◆	7	Yes
P37	◆	◆	◆	◆	◆	⬙	◇	◆	6.5	Yes

Table A.2: Quality Assessment - Included Studies

Study ID	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Score	Selected
Ec01	◇	◇	⬙	◆	⬙	◆	◇	⬙	3.5	No
Ec02	◆	◇	⬙	◆	◇	⬙	◇	⬙	3.5	No
Ec03	◇	◇	⬙	⬙	◇	◇	⬙	◆	2.5	No
Ec04	◇	⬙	⬙	◆	⬙	◇	◇	⬙	3	No
Ec05	◆	⬙	⬙	◇	⬙	◇	⬙	⬙	3.5	No
Ec06	◆	⬙	◆	⬙	⬙	◇	◇	◇	3.5	No
Ec07	◆	◇	⬙	⬙	⬙	◇	◇	⬙	3	No
Ec08	◆	◇	⬙	◆	◇	◇	◇	⬙	3	No
Ec09	◆	◇	◆	⬙	◇	◇	◇	◇	2.5	No
Ec10	◆	◇	◆	⬙	◇	◇	◇	⬙	3	No
Ec11	◇	⬙	◆	⬙	⬙	◇	◇	⬙	3	No
Ec12	◆	⬙	⬙	⬙	⬙	◇	⬙	◇	3.5	No
Ec13	◆	⬙	⬙	⬙	⬙	⬙	◇	◇	3.5	No
Ec14	◆	◇	◆	⬙	⬙	◇	◇	⬙	3.5	No
Ec15	◆	◇	⬙	⬙	◇	◇	◇	⬙	2.5	No
Ec16	◆	⬙	⬙	⬙	⬙	◇	◇	⬙	3.5	No
Ec17	◆	◆	◇	⬙	⬙	◇	◇	◇	3	No
Ec18	◆	◆	⬙	⬙	⬙	◇	◇	◇	3.5	No
Ec19	◆	◇	⬙	◇	⬙	◇	⬙	⬙	3	No
Ec20	◆	◇	⬙	◇	⬙	◇	◇	⬙	2.5	No
Ec21	◆	◇	⬙	◇	⬙	◇	◇	◇	2	No
Ec22	◆	◇	⬙	⬙	⬙	◇	⬙	⬙	3.5	No

Table A.3: Quality Assessment - Excluded Studies

Study ID	**Q01**	**Q02**	**Q03**	**Q04**	**Q05**	**Q06**	**Q07**	**Q08**	**Score**	**Selected**
Ec23	◆	◆	◆	◆	◆	⬙	◆	◆	7,5	Yes
Ec24	◆	⬙	⬙	⬙	◆	⬙	⬙	⬙	5	Yes
Ec25	◆	◆	◆	◆	⬙	⬙	◇	⬙	5.5	Yes
Ec26	◆	◆	◆	◆	⬙	◆	◆	◆	7.5	Yes
Ec27	◆	⬙	◆	⬙	◆	⬙	◇	⬙	5	Yes
Ec28	◆	◆	◆	⬙	⬙	⬙	◆	◇	5.5	Yes
Ec29	◆	⬙	◆	◆	◆	◇	◆	⬙	6	Yes
Ec30	◆	◆	◆	⬙	⬙	◇	⬙	⬙	5	Yes
Ec31	◆	◇	⬙	◆	⬙	◇	⬙	⬙	4	Yes
Ec32	◆	◆	◆	◆	◆	◇	◇	⬙	5.5	Yes
Ec33	◆	◆	◆	◆	◆	⬙	◆	⬙	7	Yes
Ec34	◆	◆	⬙	⬙	⬙	⬙	⬙	⬙	5	Yes
Ec35	◆	◇	◆	⬙	◆	◆	◆	◆	6.5	Yes
Ec36	◆	◆	◆	◆	◆	◇	◆	◆	7	Yes

Table A.4: Quality Assessment for Papers excluded by Data Extraction

Bibliography

[AABA15] S. Abid, H. Abdul Basit, and N. Arshad. Reflections on teaching refactoring: A tale of two projects. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pages 225–230, New York, NY, USA, 2015. ACM. (cited on Page 16, 23, 36, and 39)

[AHC06] D. Advani, Y. Hassoun, and S. Counsell. Extracting refactoring trends from open-source software and a possible solution to the 'related refactoring' conundrum. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1713–1720, New York, NY, USA, 2006. ACM. (cited on Page 54)

[AY14] M. Abebe and C.-J. Yoo. Trends, opportunities and challenges of software refactoring: A systematic literature review. *International Journal of Software Engineering & Its Applications*, 8, 2014. (cited on Page 47)

[BGA+04] B. Du Bois, P. Van Gorp, A. Amsel, N. Van Eetvelde, H. Stenten, and S. Demeyer. A discussion of refactoring in research and practice. Technical report, Universiteit Antwerpen, Belgium, 2004. Available online at http://lore.ua.ac.be. (cited on Page 53)

[BGS+08] P. Burnard, P.W. Gill, K.F. Stewart, E. Treasure, and B. Chadwick. Analysing and presenting qualitative data. *British Dental Journal*, 204:429 – 432, 2008. (cited on Page 18)

[Boi06] B. Du Bois. *A Study of Quality Improvements by Refactoring*. Phd, Universiteit Antwerpen, September 2006. Available online at http://lore.ua.ac.be. (cited on Page 54)

[CFF+17] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, pages 74–83, New York, NY, USA, 2017. ACM. (cited on Page 16, 23, 40, and 43)

[CFM+16] J. Cunha, J. P. Fernandes, J. Mendes, R. Pereira, J. A. Saraiva, and P. Martins. Evaluating refactorings for spreadsheet models. *Journal of Systems and Software*, 118:234–250, 2016. (cited on Page 17 and 36)

[CGM+17] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 465–475, New York, NY, USA, 2017. ACM. (cited on Page 17 and 40)

[CHJ+03] S. Counsell, Y. Hassoun, R. Johnson, K. Mannock, and E. Mendes. Trends in java code changes: The key to identification of refactorings? In *Proceedings of the 2Nd International Conference on Principles and Practice of Programming in Java*, PPPJ '03, pages 45–48, New York, NY, USA, 2003. Computer Science Press, Inc. (cited on Page 54)

[CS90] J. M. Corbin and A. Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, Mar 1990. (cited on Page 18 and 32)

[CS12] S. Counsell and S. Swift. Issues arising from refactoring studies: An experience report. *SIGSOFT Softw. Eng. Notes*, 37(3):1–5, May 2012. (cited on Page 16 and 43)

[CSGG16] D. Cedrim, L. Sousa, A. Garcia, and R. Gheyi. Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, SBES '16, pages 73–82, New York, NY, USA, 2016. ACM. (cited on Page 4, 16, 33, and 41)

[CXW+14] J. Chen, J. Xiao, Q. Wang, L. J. Osterweil, and M. Li. Refactoring planning and practice in agile software development: An empirical study. In *Proceedings of the 2014 International Conference on Software and System Process*, ICSSP 2014, pages 55–64, New York, NY, USA, 2014. ACM. (cited on Page 54)

[CXW+16] J. Chen, J. Xiao, Q. Wang, L. J. Osterweil, and M. Li. Perspectives on refactoring planning and practice: An empirical study. *Empirical Softw. Engg.*, 21(3):1397–1436, June 2016. (cited on Page 6, 17, 33, 35, 36, 38, 39, 40, 41, 43, and 44)

[DAF12] S. Dersten, J. Axelsson, and J. Fröberg. An empirical study of refactoring decisions in embedded software and systems. In *Conference on Systems Engineering Research (CSER)*. Elsevier, March 2012. (cited on Page 38 and 54)

[Fow02] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Pearson Education (US), 2002. (cited on Page 3, 4, 5, 6, 11, and 23)

[FSMS15] W. Fenske, S. Schulze, D. Meyer, and G. Saake. When code smells twice as much: Metric-based detection of variability-aware code smells. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 171–180, Sept 2015. (cited on Page 3)

[FSP+13] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Trans. Softw. Eng. Methodol.*, 22(2):14:1–14:41, March 2013. (cited on Page 54)

[GDMH12] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 211–221, Piscataway, NJ, USA, 2012. IEEE Press. (cited on Page 6, 16, 23, 38, and 40)

[Gil15] S.M. Gilliland. Empirical analysis of software refactoring motivation and effects. Master's thesis, Massachusetts Institute of Technology. Engineering Systems Division, 2015. (cited on Page 18 and 41)

[GJ02] A. Garrido and R. Johnson. Challenges of refactoring c programs. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE '02, pages 6–14, New York, NY, USA, 2002. ACM. (cited on Page 53)

[Gri92] W. G. Griswold. *Program Restructuring As an Aid to Software Maintenance.* PhD thesis, University of Washington, Seattle, WA, USA, 1992. UMI Order No. GAX92-03258. (cited on Page 3)

[GSWMH17] X. Ge, S. Sarkar, J. Witschey, and E. Murphy-Hill. Refactoring-aware code review. In *Visual Languages and Human-Centric Computing*, 2017. (cited on Page 55)

[HO15] M. Hafiz and J. Overbey. Refactoring myths. *IEEE Software*, 32(6):39–43, Nov.-Dec. 2015. (cited on Page 54)

[HRP+14] M. I. Hoque, V. N. Ranga, A. R. Pedditi, R. Srinath, Md A. A. Rana, Md E. Islam, and A. Somani. An empirical study on refactoring activity. *CoRR*, abs/1412.6359, 2014. (cited on Page 17, 33, 38, and 40)

[JBY17] L. J. Waguespack Jr., J. Babb, and D. J. Yates. "refactoring" refactoring. In *HICSS*. AIS Electronic Library (AISeL), 2017. (cited on Page 54)

[KC07] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007. (cited on Page 6)

[KCK11] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, New York, NY, USA, 2011. ACM. (cited on Page 16, 38, 40, and 41)

[KCOV18] M. Kaya, S. Conley, Z. S. Othman, and A. Varol. Effective software refactoring process. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–6, March 2018. (cited on Page 53)

[KHFG16] I. Kádár, P. Hegedűs, R. Ferenc, and T. Gyimóthy. A manually validated code refactoring dataset and its assessment regarding software maintainability. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2016, pages 10:1–10:4, New York, NY, USA, 2016. ACM. (cited on Page 54)

[KOGI17] R. G. Kula, A. Ouni, D. M. Germán, and K. Inoue. An empirical study on the impact of refactoring activities on evolving client-used apis. *CoRR*, abs/1709.09474, 2017. (cited on Page 55)

[KZN12] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 50:1–50:11, New York, NY, USA, 2012. ACM. (cited on Page 54)

[KZN14] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Trans. Softw. Eng.*, 40(7):633–649, July 2014. (cited on Page 5, 17, 33, 36, 38, 39, 40, 41, and 43)

[Lee11] D. Y. Lee. A case study on refactoring in haskell programs. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1164–1166, New York, NY, USA, 2011. ACM. (cited on Page 16, 23, and 36)

[LGN12] H. Liu, Y. Gao, and Y. Niu. An initial study on refactoring tactics. In *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference*, COMPSAC '12, pages 213–218, Washington, DC, USA, 2012. IEEE Computer Society. (cited on Page 55)

[LL16a] S. Lahtinen and M. Leppänen. Refactoring patterns, practices for daily work. In *Proceedings of the 10th Travelling Conference on Pattern Languages of Programs*, VikingPLoP '16, pages 6:1–6:8, New York, NY, USA, 2016. ACM. (cited on Page 53)

[LL16b] W. Liu and H. Liu. Major motivations for extract method refactorings: analysis based on interviews and change histories. *Frontiers of Computer Science*, 10(4):644–656, Aug 2016. (cited on Page 17 and 43)

[LLXG14] H. Liu, Y. Liu, G. Xue, and Y. Gao. Case study on software refactoring tactics. *IET Software*, 8(1):1–11, February 2014. (cited on Page 5, 18, 38, 39, 40, and 43)

[LML+15] M. Leppänen, S. Mäkinen, S. Lahtinen, O. Sievi-Korte, A. Tuovinen, and T. Männistö. Refactoring - a shot in the dark? *IEEE Software*, 32(6):62–70, Nov 2015. (cited on Page 5, 17, 36, 38, 39, 40, 41, and 44)

[MBVD03] B. Mons Belgique and A. Van Deursen. Refactoring: Emerging trends and open problems. *Proceedings 1st International Workshop on REFactoring: Achievements, Challenges, and Effects*, 12 2003. (cited on Page 54)

[MCHH07] A. Mubarak, S. Counsell, R. M. Hierons, and Y. Hassoun. Package evolvability and its relationship with refactoring. *ECEASST*, 8, 2007. (cited on Page 16, 36, 39, and 41)

[MDB+03] T. Mens, S. Demeyer, B. Du Bois, H. Stenten, and P. Van Gorp. Refactoring: Current research and future trends. *Electronic Notes in Theoretical Computer Science*, 82(3):483 – 499, 2003. LDTA'2003 - Language descriptions, Tools and Applications. (cited on Page 55)

[MFM09] S. A. Markstrum, R. M. Fuhrer, and T. D. Millstein. Towards concurrency refactoring for x10. *SIGPLAN Not.*, 44(4):303–304, February 2009. (cited on Page 53)

[MH06] E. Murphy-Hill. Improving Refactoring with Alternate Program Views. 8. Other, May 2006. (cited on Page 17, 36, and 38)

[MH09] E. Murphy-Hill. *Programmer-Friendly Refactoring Tools*. PhD thesis, Portland State University, February 2009. (cited on Page 6, 18, 23, 38, 40, and 43)

[MHB07] E. Murphy-Hill and A. P. Black. Why don't people use refactoring tools. In *In Proceedings of the 1 st Workshop on Refactoring Tools. ECOOP '07. TU Berlin, ISSN 14369915*, 2007. (cited on Page 6, 17, 38, and 40)

[MHB08a] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 421–430, New York, NY, USA, 2008. ACM. (cited on Page 55)

[MHB08b] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, Sept 2008. (cited on Page 5)

[MHB08c] E. Murphy-Hill and A. P. Black. Seven habits of a highly effective smell detector. In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, RSSE '08, pages 36–40, New York, NY, USA, 2008. ACM. (cited on Page 54)

[MHBDP08] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin. Gathering refactoring data: A comparison of four methods. In *Proceedings of the 2Nd Workshop on Refactoring Tools*, WRT '08, pages 7:1–7:5, New York, NY, USA, 2008. ACM. (cited on Page 53)

[MHPB09] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society. (cited on Page 16, 23, 38, 40, and 43)

[MKF06] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23(4):76–83, July 2006. (cited on Page 17)

[ML06] M. V. Mäntylä and C. Lassenius. Drivers for software refactoring decisions. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 297–306, New York, NY, USA, 2006. ACM. (cited on Page 16 and 36)

[MSAS06] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. Does refactoring improve reusability? In *Reuse of Off-the-Shelf Components*, pages 287–297, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (cited on Page 55)

[MT04] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004. (cited on Page 4 and 53)

[NCV+13] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 552–576, Berlin, Heidelberg, 2013. Springer-Verlag. (cited on Page 9, 17, 23, 39, 40, and 43)

[OCYC16] M. Ó Cinnéide, A. Yamashita, and S. Counsell. Measuring refactoring benefits: A survey of the evidence. In *Proceedings of the 1st International Workshop on Software Refactoring*, IWoR 2016, pages 9–12, New York, NY, USA, 2016. ACM. (cited on Page 53)

[OKS+16] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Trans. Softw. Eng. Methodol.*, 25(3):23:1–23:53, June 2016. (cited on Page 16, 36, and 41)

[OPMT15] M. Orrú, S. Porru, M. Marchesi, and R. Tonelli. The evolution of knowledge in the refactoring research field. In *Scientific Workshop Proceedings of the XP2015*, XP '15 workshops, pages 10:1–10:10, New York, NY, USA, 2015. ACM. (cited on Page 53)

[PPA+09] E. Piveta, M. Pimenta, J. Araújo, A. Moreira, P. Guerreiro, and R. T. Price. Representing refactoring opportunities. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1867–1872, New York, NY, USA, 2009. ACM. (cited on Page 53)

[PR06] M. Petticrew and H. Roberts. *Systematic Reviews in the Social Sciences: A Practical Guide*, volume 11. Blackwell Publishing, 01 2006. (cited on Page 9)

[PZODL17] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia. An exploratory study on the relationship between changes and refactoring. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 176–185, Piscataway, NJ, USA, 2017. IEEE Press. (cited on Page 16, 36, 38, 40, and 41)

[RBJ97] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, October 1997. (cited on Page 6)

[RR04] J. Rech and E. Ras. Experience-based refactoring for goal-oriented software quality improvement. In *The First International Workshop on Software Quality (SOQUA)*, 2004. (cited on Page 54)

[SAN+17] G. Szőke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. *Journal of Systems and Software*, 129(C):107–126, July 2017. (cited on Page 17)

[SCY+14] T. Saika, E. Choi, N. Yoshida, A. Goto, S. Haruna, and K. Inoue. What kinds of refactorings are co-occurred? an analysis of eclipse usage datasets. In *IWESEP*, pages 31–36. IEEE Computer Society, 2014. (cited on Page 18)

[SD16] Inc. Semantic Designs. Refactoring Tools. Website, 2016. Available online at http://www.semdesigns.com; visited on July 27th, 2018. (cited on Page 6)

[SGMHJ13] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *J. Syst. Softw.*, 86(4):1006–1022, April 2013. (cited on Page 55)

[Sho05] J. Shore. Red-Green-Refactor. Website, 2005. Available online at https://www.jamesshore.com; visited on August 28th, 2018. (cited on Page 4)

[SL11] R. Shatnawi and W Li. An empirical assessment of refactoring impact on software quality using a hierarchical quality model. *International Journal of Software Engineering and its Applications*, 5:127–150, 01 2011. (cited on Page 53)

[SNFG14] G. Szőke, C. Nagy, R. Ferenc, and T. Gyimóthy. A case study of refactoring large-scale industrial systems to efficiently improve source code quality. In *Computational Science and Its Applications – ICCSA 2014*, pages 524–540, Cham, 2014. Springer International Publishing. (cited on Page 17, 38, and 41)

[SPB16] S. Stuurman, H. Passier, and E. Barendsen. Analyzing students' software redesign strategies. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, pages 110–119, New York, NY, USA, 2016. ACM. (cited on Page 54)

[Ste15] F. Steimann. Refactoring tools and their kin. In *GTTSE*, volume 10223 of *Lecture Notes in Computer Science*, pages 179–214. Springer, 2015. (cited on Page 55)

[Ste17] F. Steimann. Refactoring tools and their kin. In *Grand Timely Topics in Software Engineering*, pages 179–214, Cham, 2017. Springer International Publishing. (cited on Page 6)

[STV16] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 858–870, New York, NY, USA, 2016. ACM. (cited on Page 6, 16, 23, 26, 29, 36, 38, 40, and 48)

[TC09] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.*, 35(3):347–367, May 2009. (cited on Page 32)

[TGA17] E. Tempero, T. Gorschek, and L. Angelis. Barriers to refactoring. *Commun. ACM*, 60(10):54–61, September 2017. (cited on Page 5, 9, 17, 33, 35, 36, and 41)

[TGSH13] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 132–146, Riverton, NJ, USA, 2013. IBM Corp. (cited on Page 16, 23, 36, 44, and 47)

[TR03] S. Thompson and C. Reinke. A case study in refactoring functional programs. In *VII Brazilian Symposium on Programming Languages*, pages 182–196. Sociedade Brasileira de Computacao, May 2003. (cited on Page 17, 36, and 43)

[VCN+11] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, R. Zilouchian Moghaddam, and R. E. Johnson. The need for richer refactoring usage data. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 31–38, New York, NY, USA, 2011. ACM. (cited on Page 54)

[VCN+12] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 233–243, Piscataway, NJ, USA, 2012. IEEE Press. (cited on Page 6, 16, 23, and 47)

[VGSMD03] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent uml refactorings. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, pages 144–158, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. (cited on Page 3)

[Wan09] Y. Wang. What motivate software engineers to refactor source code? evidences from professional developers. In *2009 IEEE International Conference on Software Maintenance*, pages 413–416, Sept 2009. (cited on Page 17, 35, 38, and 47)

[WKK07] D. Wilking, U. F. Khan, and S. Kowalewski. An empirical evaluation of refactoring. *e-Informatica*, 1:27–42, 2007. (cited on Page 16)

[Wlo03] J. Wloka. Refactoring in the presence of aspects. In *13th Workshop for PhD Students in Object-Oriented Systems (PhDOOS), at ECOOP '03*, 2003. (cited on Page 54)

[XS06] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Sept 2006. (cited on Page 17, 36, and 38)

[YDL18] Y. Yongting, L. Dongsheng, and Z. Liping. Detection technology and application of clone refactoring. In *Proceedings of the 2018 2Nd International Conference on Management Engineering, Software Engineering and Service Sciences*, ICMSS 2018, pages 128–133, New York, NY, USA, 2018. ACM. (cited on Page 53)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 26. November 2018