



Ana Carla Gomes Bibiano

**Understanding Characteristics and Structural
Effects of Batch Refactorings in Practice**

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-graduação em
Informática da PUC-Rio in partial fulfillment of the requirements
for the degree of Mestre em Informática .

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
April 2019



Ana Carla Gomes Bibiano

Understanding Characteristics and Structural Effects of Batch Refactorings in Practice

Dissertation presented to the Programa de Pós-graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática . Approved by the undersigned Examination Committee.

Prof. Alessandro Fabricio Garcia

Advisor

Departamento de Informática – PUC-Rio

Prof. Leonardo Gresta Paulino Murta

Universidade Federal Fluminense – UFF

Prof. Marcos Kalinowski

Departamento de Informática – PUC-Rio

Rio de Janeiro, April 26th, 2019

All rights reserved.

Ana Carla Gomes Bibiano

Ana Carla Bibiano is a Master's student in Informatics at Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. She holds a Bachelor's degree in Computer Science from Federal University of Alagoas (2017). In 2014, she earned the Academic Excellence award by the Brazilian National Council for Scientific and Technological Development (CNPq) for her scientific initiation's work. She has worked as a software developer for companies from Maceió (AL), Blumenau (SC), and Rio de Janeiro (RJ). Ana Carla is currently a research scholar in Software Engineering for the OPUS Research Group at PUC-Rio. Her main research interests are (not limited to): software refactoring, maintenance, and evolution.

Bibliographic data

Gomes Bibiano, Ana Carla

Understanding Characteristics and Structural Effects of Batch Refactorings in Practice / Ana Carla Gomes Bibiano; advisor: Alessandro Fabricio Garcia. – 2019.

82 f: il. color. ; 30 cm

Dissertação (mestrado)—Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2019.

Inclui bibliografia

1. Informática – Teses.

2. Refatoração em Lote;. 3. Manutenção de Software;. 4. Anomalia de Código-Fonte;. 5. Revisão de Literatura;. 6. Estudo Quantitativo.. I. Garcia, Alessandro. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

First and foremost, I would like to thank God Almighty for the gift of living, growing up, and contributing the His plan the way I can. I also thank the Virgin Mary for being my role model and intercessor along the way. I am full of gratitude for my parents, Mrs. Lourinete Bibiano and Mr. Antonio Bibiano (*in memoriam*), my grandparents, brothers, and sister. Thanks for the kind support support. Especial thanks to Mr. José Pedro Luiz (*in memoriam*), my dear grandfather, for inspiring me to be a scientist. Thanks to all my friends for the support and patience along the years, especially Jedson Amaro, who helped me to reach my goals and was present in both happy and hard times, my dear friend Jakson Leao that brought me happiness and advice, and my brother Ladovanio Gomes that was my support all the time.

Many people became special to my academic formation throughout this journey. I am grateful for their support along the last two years of study along my Master's course. I would like to thank Prof. Dr. Alessandro Garcia, my advisor, for believing in my potential as a scientist. Thanks for helping me reach my dreams. I also thank my dear friend MSc. Eduardo Fernandes who became my research colleague but also a friend for life. During the last two years, we grew together spiritually and academically. Thanks for believing and supporting me, even when I did not believe in myself. I am grateful for the reviews and improvements suggested to this Master's dissertation, including the literature review protocol.

Thanks to Dr. Diego Cedrim e MSc. Isabella Ferreira for supporting me in the early phases of my research. You provided me with key support to build the code smells *versus* batch refactoring database. Thanks also to BS Daniel Oliveira for the support in collecting and validating data used in this work. I extend my thanks to all OPUS Research Group members and collaborators from other groups, especially Profs. Drs. Balduino Fonseca (UFAL) and Marcos Kalinowski (PUC-Rio). I am thankful for your feedback aimed to shape the directions of my research. Especial thanks to the (anonymous) software development companies that provided me with data for analysis. Thanks to Profs. Drs. Leonardo Murta (UFF), Marcos Kalinowski (PUC-Rio), and Simone Barbosa (PUC-Rio) for kindly accepting to integrate this Master's dissertation defense committee.

Doing science in Brazil is challenging and, in the context, I would like to thank all agencies that provided me with funding, especially the Brazilian National Council for Scientific and Technological Development (CNPq), plus the partner companies of the Software Engineering Laboratory (LES/PUC-Rio) for the student scholarship.

Abstract

Gomes Bibiano, Ana Carla; Garcia, Alessandro (Advisor). **Understanding Characteristics and Structural Effects of Batch Refactorings in Practice**. Rio de Janeiro, 2019. 82p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Code refactoring means applying transformations on the code structure of a software project. Refactoring usually intends to remove poor code structures that harm the software maintenance. Each single transformation rarely suffices to fully remove poor code structures, even the simplest ones. For instance, shortening a long method often requires many method extractions. Up to 60% of the refactorings in software projects are constituted of a set of interrelated transformations, the so-called *batches*, rather than single transformations applied in isolation. Although batches are frequent in practice, the knowledge of batch characteristics is fragmented across studies. What is the usual size of batches? How do transformations vary within a batch? There is no summary that helps to address these questions. More critically, there is little empirical evidence of the batch effect on maintenance. Are batches more likely to introduce or remove poor code structures, especially those spotted by code smells? The current answer to questions like this is insufficient to support the batch application in practice. This Master's dissertation presents two complementary empirical studies that address both aforementioned literature gaps. The dissertation starts with a literature review of batch refactoring with 29 studies. We identified seven batch characteristics such as the scope in which batches are applied to code structures, plus seven types of batch effect on software maintenance, including code smell removal. All batch characteristics and types of effect were summarized in a conceptual map. The dissertation ends with the quantitative analysis of 57 open and closed software projects. From 4,607 heuristic-computed batches, we found that most batches occur entirely within one commit (93%) but affect more than just one method (90%). Surprisingly, batches mostly end up introducing (51%) or not removing (38%) code smells. Our results enabled us to reveal certain forms of batches, not documented by previous studies, that are useful to fully remove certain types of code smells.

Keywords

Batch Refactoring; Software Maintenance; Code Smell; Literature Review; Quantitative Study.

Resumo

Gomes Bibiano, Ana Carla; Garcia, Alessandro. **Entendendo Características e Efeitos Estruturais de Refatoração em Lotes na Prática**. Rio de Janeiro, 2019. 82p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Refatorar código-fonte consiste em aplicar transformações sobre a estrutura de código-fonte de projetos de software. Refatoração é bastante usada para remover estruturas pobres que dificultam a manutenção de sistemas de software. Poucas transformações isoladas são capazes de remover por completo estruturas pobres, mesmo as mais simples. Por exemplo, encurtar um método longo usualmente requer a extração de vários métodos. Até 60% das transformações são inter-relacionadas e aplicadas em lotes, durante a dita refatoração em lote, ao invés de aplicadas isoladamente. Embora lotes serão frequentes na prática, o conhecimento sobre as características que constituem lotes está fragmentado na literatura. Qual o tamanho usual de lotes? As transformações internas a lotes costumam variar? Não há uma sumarização de conhecimento que responda tais questões. Ademais, são poucas as evidências sobre o efeito de lotes sobre a manutenção de sistemas. Lotes tendem a introduzir ou remover estruturas pobres, especialmente aquelas indicadas por anomalias de código-fonte? A resposta a perguntas como essa é insuficiente para apoiar a aplicação de lotes. Esta dissertação de mestrado apresenta dois estudos experimentais complementares visando resolver as limitações supracitadas. A dissertação começa com uma revisão da literatura sobre refatoração em lote baseada em 29 estudos. Nós identificamos sete características de lotes tais como o escopo de código-fonte afetado pela aplicação de um lote, mais sete tipos de efeito de lotes sobre a manutenção de sistemas, tais como a remoção de anomalias. As características e tipos de efeito identificadas foram sumarizadas por um mapa conceitual. A dissertação encerra-se com uma análise quantitativa de 57 projetos de sistemas abertos e fechados. Ao computar 4.607 lotes com uma heurística, nós descobrimos que a maioria dos lotes leva um único commit para ser aplicada (93%) mas afeta mais do que um só método (90%). Surpreendentemente, a maioria dos lotes introduz (51%) ou não remove (38%) anomalias. Revelamos também lotes até então desconhecidos mas capazes de remover por completo certas anomalias. Esta dissertação sugere trabalhos futuros com base em conflitos identificados na literatura quanto a características e tipos de efeito de lotes.

Palavras-chave

Refatoração em Lote; Manutenção de Software; Anomalia de Código-
Fonte; Revisão de Literatura; Estudo Quantitativo.

Table of contents

1	Introduction	12
1.1	Problem Statement and Limitations of Related Work	13
1.2	A Literature Review of Batch Refactoring	16
1.3	A Large Study of Batch Characteristics and Structural Effect	17
1.4	Dissertation Outline	19
2	Background and Related Work	20
2.1	Code Refactoring and Transformation Types	21
2.2	Batch Refactoring at a Glance	22
2.3	Poor Code Structures and Code Smells	24
2.4	Final Remarks	25
3	A Literature Review of Batch Refactoring	27
3.1	Literature Review Protocol	28
3.2	Study Steps	29
3.3	Conceptual Map of Batch Refactoring	32
3.4	Batch Characteristics	34
3.5	Batch Effect on Software Projects	37
3.6	Conflicting Batch Characteristics and Types of Effect	39
3.7	Threats to Validity	42
3.8	Final Remarks	43
4	Batch Characteristics and Effect on Code Smells in Practice	44
4.1	Study Design	45
4.1.1	Goal and Research Questions	45
4.1.2	Study Steps and Definitions	49
4.2	Results and Discussions	52
4.2.1	Manifestations of Batch Characteristics (RQ ₁)	52
4.2.2	Nature of Code Transformations within Batches (RQ ₂)	54
4.2.3	Batches Affecting Smelly versus Smell-Free Code Elements (RQ ₃)	57
4.2.4	Structural Effect of Batches on Code Smells (RQ ₄)	58
4.3	Threats to Validity	63
4.4	Final Remarks	65
5	Conclusion	67
5.1	Summary of Study Contributions	68
5.2	Insights to Enhance Current Refactoring Support	70
5.3	Research Publications	72
	Bibliography	73

List of figures

Figure 2.1	An Example of Batch Refactoring Extracted from Elasticsearch	23
Figure 3.1	Steps of the Literature Review	29
Figure 3.2	Number of Papers by Publication Year	33
Figure 3.3	Conceptual Map of Batch Refactoring	34
Figure 4.1	Study Steps	49

List of tables

Table 2.1	Transformation Types Exploited by this Dissertation	22
Table 2.2	Code Smell Types Exploited by this Dissertation	25
Table 3.1	Research Questions of the Literature Review	29
Table 3.2	Papers Selected through the Pilot Search	30
Table 3.3	Data Types Extracted from Each Paper	31
Table 3.4	Papers Selected from the Literature Review	32
Table 4.1	Manifestations of each batch characteristic	47
Table 4.2	Nature of Transformation Types	48
Table 4.3	Manifestations by batch characteristic	53
Table 4.4	Frequency of Batches According to their Single or Hybrid Nature	54
Table 4.5	Frequency by Batch Category	58
Table 4.6	Batch effect on code smells by batch characteristic	59
Table 4.7	Effect of batches by category	60
Table 4.8	Batch Effect on Code Smells according to the Nature of Code Transformations	62
Table 5.1	List of Research Publications	72

*It's just a spark but it's enough to keep me
going / And when it's dark out and no one's
around it keeps glowing*

Hayley Williams & Taylor York, Paramore's Last Hope.

1

Introduction

Code refactoring consists of applying one or more transformations on the code structure of a software project (25). Refactoring has been largely employed by developers, in major companies such as Google (71) and Microsoft (35), to remove poor code structures that represent threats to code maintenance (25). Software projects are usually considered maintainable when their code structures are easy to understand and modify (72). The motivations behind code refactoring vary significantly by developer (60). These motivations often range from enabling the addition of new software features to supporting developers in fixing software bugs (46, 60). However, regardless the developer motivation, it is expected that the code transformations applied along with code refactoring can enhance the code structures (35, 60).

Applying code refactoring in practical settings is quite complex (35). There is a plenty of code transformation types that developers can use according to their maintainability enhancement goals (25). Each type defines how a developer should modify the code elements, such as attributes, methods, and classes (25, 43). An example is the Extract Method transformation type, which consists of creating a new method from code statements of an existing method (67). Extract Method can be used to better separate the software features across different methods (25). Another example is Move Method, which consists of moving a method across classes (66). This transformation type is usually applied to allocate a method to the most appropriate class (8).

The types of code transformations that can be applied by developers are quite varied, and so are the possible types of refactoring effect on code maintenance. Fowler' refactoring book (25) and other studies (29, 45) present mechanisms for applying code transformations towards removing poor code structures that indicate maintenance problems (35, 54, 60). These poor code structures are usually represented by the so-called code smells (25, 74). One of the most frequently investigated code smell types (16, 48, 54, 66, 74) is Feature Envy. Instances of this type are characterized by (parts of) a method that uses too much resources provided by other classes rather than its host class (38). Removing Feature Envy instances can be enabled by the application of Move Method from the original to the “envied” class (66).

A single transformation rarely suffices to achieve the developer motivations behind refactoring (46). That is the case of code smell removal, for which two or more code transformations have to be combined to fully remove a code smell instance (8). For instance, previous studies recommend the combination of Extract Method and Move Method transformations aimed to fully remove a Feature Envy instance (8, 25). A recent study has shown that about 40-60% of code transformations are applied in *batches*, i.e., sets of two or more interrelated transformations, rather than in isolation (46). The phenomenon of applying one or more batches on software projects is called *batch refactoring* (7). However, although batches are frequently applied in practice, there are major problems concerning the knowledge about batch refactoring that deserves attention. We summarize some of these problems in the next section.

1.1

Problem Statement and Limitations of Related Work

The current knowledge of batch refactoring is fragmented – As aforementioned, batches have been largely applied by developers in real settings (7, 46). Unfortunately, the current knowledge of batch refactoring is considerably fragmented across previous studies. As far as the characteristics that constitute a batch are concerned, there are major limitations in the literature. Each researcher seems to have a particular view of what constitutes batch refactoring. In fact, past studies assume one or another batch characteristic, such as the variety of transformation types within a batch (43, 55, 64). However, the authors do not systematically evaluate if these characteristics are common in practice. In other words, previous studies usually mention or assume that batches are often constituted of a certain characteristic without any empirical validation. More critically, due to the lack of empirical evidence, previous studies usually contradict one another with respect to the assumptions underlying batch characteristics. That is the case of studies that consider batches as composed of transformations that only share a single transformation type (46), contrarily to others that assume batches as composed of varied transformation types (35, 64).

Moreover, there is limited knowledge about the effect of batches on code maintenance. Some previous studies assume that batches are beneficial to the code maintenance by fully removing poor code structures that each transformation in isolation cannot remove (7, 37, 43). Conversely, certain studies discuss that batches can sometimes be detrimental to the code structure quality (43), thereby introducing poor code structures like code smells (7) and hindering maintenance tasks. Most of these studies lack empirical evidence, which ul-

timately makes their assumptions debatable. Such lack of empirical evidence can lead to conflicts among studies as it occurs for batch characteristics. Thus, developers may keep reluctant in refactoring their projects, because they fear to worsen rather than enhance the code structures (35).

Future research on batch refactoring cannot be properly performed without a summary of the currently fragmented knowledge about batch refactoring. This summary could be formalized by a unified conceptual map of batch characteristics and types of effect on code maintenance. A recent doctoral thesis (7) defended by a PhD student of the OPUS Research Group ¹ in 2018 has started to elicit batch characteristics for investigation, but not in a systematic way. Consequently, the previously proposed summary is quite limited and may not comprise a considerable number of studies. We hypothesize that a comprehensive conceptual map could guide future research by pointing out: (i) which batch characteristics and types of batch effect have been empirically investigated and could be used as a basis for future work; and (ii) which characteristics and types of effect are poorly investigated or have been reported with conflicts among studies – thus, they should be also further investigated.

Research Problem 1: The currently fragmented knowledge of batch refactoring leads to conflicts among studies and hinders future investigations.

The most frequent manifestations of batch characteristics remain unknown – According to the literature, each batch characteristic can manifest differently in practice. For instance, the scope of a batch can vary. Some batches can affect just one method of a particular class, while others can have effect on a wider scope of the code structure (7, 25, 43, 55), i.e., batches can affect multiple methods, a whole class or even multiple classes. Some batches can also be composed of the same transformation type, e.g., Extract Method only (46), while others can combine multiple types like Extract Method and Move Method (7). Unfortunately, none of the previous studies have systematically investigated what are the most frequent manifestations of batch characteristics, especially through empirical studies. The closest that we could find was the study presented by the aforementioned PhD thesis (7). However, there are characteristics and manifestations that still lack investigation.

A clear understanding of how batch characteristics usually manifest in software projects can be beneficial to future research in many ways. By

¹<http://opus.les.inf.puc-rio.br/wordpress/>

characterizing the usual structure and size of batches, researchers can propose more accurate heuristics for identifying batches applied along the version history of existing software projects. Additionally, researchers could draw new strategies to guide the batch application by taking advantage of less frequent manifestations to enhance code structures. For instance, if batches rarely combine multiple code transformation types, we could provide developers with batch recommendations that combine types according to literature findings and assist them to improve the code structure of their programs.

Research Problem 2: The limited empirical knowledge about the most frequent manifestations of batch characteristics makes hard to guide batch application in practice.

Empirical evidence of the batch effect on code maintenance is quite scarce – The current knowledge about the types of batch effect on code maintenance is limited. In fact, the literature (2, 8, 9, 10) has ultimately focused on assessing the effect of each single code transformation applied along refactoring rather than batches. Some studies like (2, 21) have assessed to what extent code transformations prevent the introduction of software bugs. The previous study results are mixed and point out that, although single code transformations are prone to reduce bug introduction (2), bugs do not occur that far from the transformations in the commit history of software projects (21). Other studies show that not always the single code transformations are beneficial to code structures (8, 10). In fact, single transformation tend to either introduce 3% or not fully remove 95% code smells (8). Unfortunately, little is discussed about the batch effect on code maintenance, especially in terms of code smell introduction, which has been exploited *ad nauseam* by previous work on isolated transformations.

The aforementioned PhD thesis (7) presented the very first study aimed to investigate the effect of batches on code smells. The thesis has analyzed both introduction and removal of code smells in a total of 48 software projects. However, the effect of batches analysis conducted by the previous work is limited because they only evaluated the effect of batches, such as the number of code smells before and after the batch application. This previous study did not report which categories of batches, with their different characteristics, have different effects on the code structure.

Research Problem 3: The limited knowledge about the batch effect on code smells hinders the recommendation of batches for use in practice.

This Master's dissertations addressed the three aforementioned research problems through two complementary empirical studies. With a literature

review of batch refactoring, we aimed to build a conceptual map of batch characteristics and types of effect, thereby addressing Research Problem 1. Through a large quantitative study, our goal was understanding the most frequent manifestations of batch characteristics (Research Problem 2) and the effect of batches on code smells (Research Problem 3) in order to contrast our study results with previous ones in the context of either single code transformations or batches. We summarize each study as follows.

1.2

A Literature Review of Batch Refactoring

The first study that composes this Master's dissertation was a literature review of batch refactoring. This study is intended to address our first research problem. Due to the knowledge fragmentation presented by the literature, we decided to perform a literature review based on well-known guidelines (36). Thus, we expected to properly characterize the state-of-the-art on the topic. Our study goal was three-fold:

1. Eliciting the *batch characteristics* either mentioned or explored by previous studies. A summary of characteristics could give us the big picture of what does constitute batches from a researcher perspective.
2. Eliciting the *types of batch effect on code maintenance* as assumed by the literature. The summary of assumed types of batch effect could guide future research with focus on empirical validation.
3. Identifying conflicts among previous studies with respect to batch characteristics and types of effect. Such identification could reveal opportunities for future research aimed to solve conflicts and leverage the current empirical knowledge of batch refactoring.

We have found a total of 29 previous studies published on international conferences and journals. From the full-text read of these studies, we were able to identify: seven batch characteristics that regard both structural or other aspects of batches; seven types of batch effect on code maintenance, which range from internal to external effects on software projects; and seven conflicts among studies about what does characterize batches and which effects to expect from the application of batches in practice. Based on the knowledge acquired from our literature review, we followed the feature model principles (32) to build a conceptual map of batch refactoring. Thus, the first contribution of this dissertation can be stated as follows.

Contribution 1: A conceptual map of batch refactoring that encompasses batch characteristics and types of effect reported by the literature so far. Our conceptual map can guide future research towards leveraging the current knowledge on what does characterize a batch and how it affects code maintenance.

1.3

A Large Study of Batch Characteristics and Structural Effect

The second study that composes this Master's dissertation was a large-scale empirical study of batch refactoring. This study has two parts aimed to address Research Problems 2 and 3 respectively. Both parts share a common study design, which relied on the collection and analysis of data from 57 open and closed projects. These projects were either downloaded from public GitHub repositories or provided by Brazilian companies. We run a batch detection heuristic introduced by a previous work (7) in order to collect a set of 4,607 batches applied on these projects. We were able to address our last two research problems as explained below.

We first focused on understanding how the batch characteristics elicited from our literature review (Chapter 3) manifest more frequently in real software projects. For this purpose, we cherry-picked four batch characteristics. For example, we selected the number of code transformations within a batch. We analyzed the frequency of the possible manifestations of each characteristic. Each characteristic had two possible manifestations. For example, the characteristic *number of transformations* had two manifestations in our study: (i) batches with the minimum cardinality. i.e, two transformations only, and (ii) batches with three or more transformations.

We derived categories of batches based on the manifestations of each batch characteristic. In total, we had 16 possible categories for batches based on the possible combinations of the two manifestations of the four characteristics. As a result, we have found that most batches follow a general trend: 93% occur in one commit, 72% are constituted of the same transformation type and 22% range from four to ten code transformations. Surprisingly, 60% of batches applied on code elements affected by code smells were constituted by transformations types of either extraction or motion natures (e.g., Extract Method and Move Method). The second contribution of this dissertation can be stated as follows.

Contribution 2: An empirical study on the frequent manifestations of batch characteristics in real software projects. The fact that most batches (72%) are constituted of the same transformation type suggests that batches recommended by Fowler’s refactoring book (25) to fully remove code smells have been underutilized in practice. Additionally, the high rate of batches (60%) composed by extractions and movements reinforces that developers require guidance to apply batches in practice.

Finally, we assessed the effect of batches on code maintenance with respect to the introduction and removal of code smells. We have used automated tools for identifying 19 different types of code smells, such as Large Class and Long Method. We then analyze the structural effect of applying the candidate batch refactorings on the programs. We have computed the total number of code smells before and after the application of each batch. We only consider the code smells affecting code elements within the scope of the transformations in batches. In other words, we do not consider the other code elements not affected by the refactoring. This procedure is important to make sure we increase the likelihood of discarding changes that have no relationship with the transformations in batches. Thus, we have computed three types of batch effect on the code structure of software projects:

1. *Positive effect:* the number of code smell instances was reduced after the batch application when compared to the number of instances affecting the refactored code before the batch application.
2. *Neutral effect:* the number of code smells instances affecting the refactored code has not changed after the batch application.
3. *Negative effect:* the number of code smell instances increased after the batch application when compared to the number of instances affecting the refactored code before the batch application.

Our results suggest that most batch refactorings either introduce (51%) or do not suffice to remove (38%) code smells. This observation is quite similar to that obtained for single code transformations by a previous work (8). More importantly, this result suggests that, even with the potential to enhance code structures, batches are still poorly applied by developers. Thus, mechanisms for recommending useful batches capable to fully remove code smells are desired. Nevertheless, our results pointed out some combinations of code transformations that can be recommended to users but were not yet exploited by the literature. For instance, a batch composed by Pull Up Methods to

remove a Message Chain code smell. Thus, our third contribution can be stated as follows.

Contribution 3: An empirical study of the batch effect on code smells. Similarly to single code transformations, batches are poorly exploited by developers and tend to introduce (51%) rather than remove code smells. Developers need guidance to apply batches in practice, and our study provided some hints of recommendable batches.

1.4

Dissertation Outline

The remainder of this Master's dissertation is structured as follows.

Chapter 2 provides background information aimed to support the understanding of this dissertation. We discuss basic concepts of code refactoring, batch refactoring (with an illustrative example), and the relationship between poor code structures and code smells.

Chapter 3 presents our literature review of batch refactoring. We introduce our review-based conceptual model of batch characteristics and types of effect on code maintenance. Thus, we track some opportunities for future work. We also discuss how this conceptual model is used as basis to the next chapter.

Chapter 4 presents the quantitative results of our empirical study aimed to investigate (i) the most frequent manifestations of batch characteristics and (ii) the structural effect of batches on code smell instances in real software projects. We discuss how these results can support future work on the support to batch application.

Chapter 5 concludes this Master's dissertation by summarizing our research contributions, the outputs of this dissertation by means of ongoing research papers, and some opportunities for future work.

Code refactoring has been extensively employed to enhance code structures of software projects (35, 59, 60). Through the application of one or more transformations to code structures, developers can make the resulting program easier to read, understand, and modify (46). Code transformations applied along code refactoring vary in terms of type, which is defined by the nature of modification to be applied on the code structure (25). Examples of code transformations are extractions and movements of class members – whose application has been often made by developers in real software projects (8, 10, 46, 60). Each code transformation affects one or more code elements, e.g., methods and classes (25).

Previous studies – e.g., (8, 10, 46) – have investigated the basic characteristics of batches and their effect on code maintenance. Some studies assessed how these characteristics are likely to manifest on software projects. These studies have found that Rename Method, Extract Method, and Move Method are the most frequently applied transformation types (7, 46). Other studies suggest that developers have varied motivations behind code refactoring. These motivations range from purely improving code structures to enabling the addition of new features or even fixing software bugs (52, 60). In any case, by definition (25), code transformations applied along with code refactoring affect the program structure. It should ideally improve or at least not worsen code structures.

Contrary to expectations, recent studies reveal the drawbacks of performing code refactoring. One study discusses that most single transformations can worsen internal code attributes, such as cohesion and coupling, in a non-ignorable rate (10). Another study shows that certain code transformations tend to either introduce or not fully remove poor code structures represented by code smells (8). These observations can at least partially explain why developers often have to apply two or more interrelated code transformations, thereby forming *batches* (46), so that such developers can fully achieve their motivations behind refactoring. This particular phenomenon, as known as *batch refactoring* (7), has only been observed in code smell removal (7) so far.

Although batch refactoring has been frequently applied in practice, there

is little knowledge about batches. This Master's dissertation aims at addressing two key literature gaps through complementary empirical studies. The first study (Chapter 3) relies on a literature review for building a conceptual map of batch characteristics and types of effect on code maintenance. The second study (Chapter 4) consists of quantitative analyzes aimed to understand the frequent manifestations of batch characteristics in real projects, besides the actual batch effect on introducing and removing code smells.

Aimed to support the understanding of this Master's dissertation, this chapter presents key concepts used throughout the dissertation chapters. We summarize the definition and principles of code refactoring (Section 2.1). Then, we introduce batch refactoring (Section 2.2) based on recent research and a real-world example. After that, we provide a general view on the relationship between poor code structures and code smells (Section 2.3). Section 2.4 concludes this chapter and introduces the next one.

2.1

Code Refactoring and Transformation Types

Major companies like Google (71) and Microsoft (35) have employed code refactoring for leveraging the internal quality of their software projects. As a software development practice, code refactoring consists of applying two or more transformations on the code structure of a given project (25). Each code transformation affects the code elements in order to make code structures easier to read and modify (25, 43, 46, 66). Thus, the application of code transformations along with code refactoring has been seen as beneficial to the code maintenance (25).

There is a myriad of code transformations catalogued by the literature in order to guide developers to enhance code structures. Examples of frequently applied transformation types are Extract Method and Move Method (46, 60). Extract Method is defined by extraction of specific code statements from an existing method (66). The extracted code statements are used to build a new method (25). Move Method consists of moving an existing method from the original class to another class (67). Both transformation types can be used to separate software features across methods and classes (25).

Table 2.1 lists and defines the 13 code transformation types exploited by this Master's dissertation. We carefully selected transformation types from existing catalogs, especially Fowler's refactoring book (25), in order to support our study. The four types listed in the table regard *class-level transformations*, i.e., code transformations that ultimately affect a class or an interface – considering object-oriented programming languages like Java. The following

six types listed in the table regard *method-level transformations* and, therefore, they affect methods within a class. The last three types regard the *attribute-level transformations*, i.e., they consist of modifying attributes of a given class. We chose all those 13 transformations based on their practical popularity as reported by previous work such as (46) and (60).

Table 2.1: Transformation Types Exploited by this Dissertation

Type	Definition based on (60) and (66)
Extract Interface	Create common interface for existing classes
Extract Superclass	Extract superclass from code shared by existing classes
Move Class	Move class from one package to another package
Rename Class	Rename a class
Extract Method	Create method based on statements of an existing method
Inline Method	Incorporate the body of a method into an existing method
Move Method	Move method from one class to another class
Push Down Method	Move method from a parent class to one or more child classes
Pull Up Method	Move method from a child class to its parent class
Rename Method	Rename a method
Move Attribute	Move attribute from one class to another class
Push Down Attribute	Move attribute from a parent class to one or more child classes
Pull Up Attribute	Move attribute from a child class to its parent class

There is a plenty of possible developer motivations behind code refactoring. A recent study (60) has empirically concluded that developers are often motivated by either enabling the addition of new program features or fixing software bugs. However, regardless the developer motivation behind refactoring, it is expected that code transformations improve the code structures underlying a software project (8, 9, 10). By definition, each transformation aims to make code elements easier to maintain (25). Therefore, guidance aimed to support code refactoring practices is desired. In this context, automated tools such as JDeodorant (23) and FaultBuster (64) have been proposed for supporting developers in applying isolated transformations for enhancing code structures, regardless the major developer motivation with refactoring. Unfortunately, these tools are insufficient to support developers in certain recurring refactoring practices (e.g. batch refactoring), as discussed in the next section.

2.2

Batch Refactoring at a Glance

Developers are more likely to apply sets of interrelated code transformations in conjunction rather than single transformations (35, 46). Up to 60% of code transformations are applied in batches (46). This Master's dissertation relies on previous studies (7) for defining a *batch* as follows: **a set of two or more interrelated code transformations applied in conjunction along with code refactoring**. An example of a batch is the resulting composi-

tion of Extract Method and Move Method aimed to separate program features across methods (8, 25). We represent batches based on the traditional set notation. For instance, the batch exemplified above is represented as $b = \{\text{Extract Method, Move Method}\}$. The phenomenon of applying one or more batches on software projects is called *batch refactoring* (19, 43, 64).

A real-world example of batch refactoring. Figure 2.1 illustrates how developers have been applying batches to their software projects so far. This example of batch refactoring was extracted from three commits performed in the context of the Elasticsearch software project. The source code is publicly available at GitHub for consultation^{1,2,3}. Other well-documented examples of real batches applied on open source projects can be found in recent studies like (19) and (20). The example below is intended to illustrate the application of a non-trivial batch in practice.

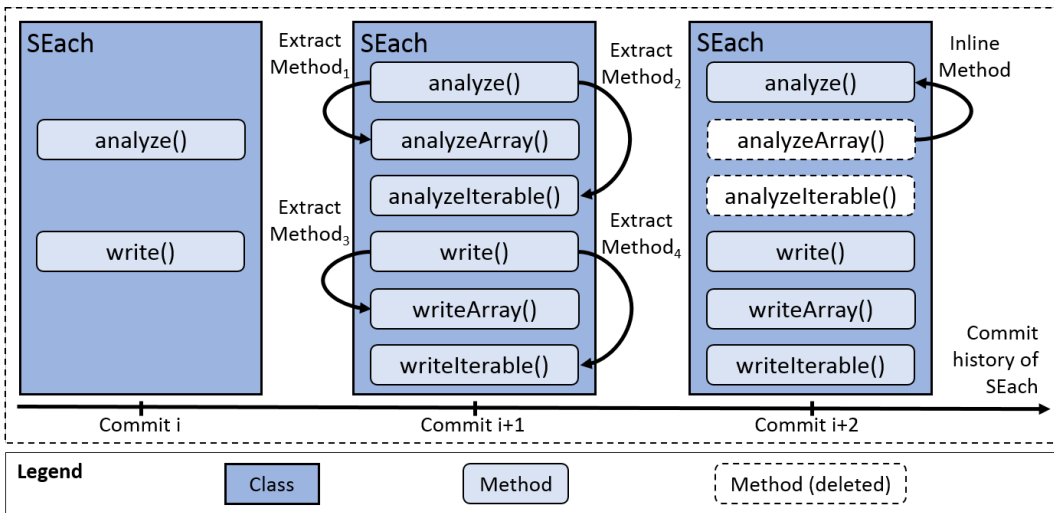


Figure 2.1: An Example of Batch Refactoring Extracted from Elasticsearch

Along the commit history of the project, especially in Commits i , $i + 1$, and $i + 2$, one developer has applied various code transformations on the `SEach` class. In Commit $i + 1$, the developer has applied four `Extract Method` instances on two methods implemented by the class in Commit i : `analyze()` and `write()`. We hypothesize that the suffixes used for naming the new methods ("`...Array`" and "`...Iterable`"), resulting from `Extract Method` transformations, suggest an intentional separation of software features implemented by the original methods, i.e., `analyze()` and `write()`. Thus, the `Extract Method` instances applied in Commit $i + 1$ are somehow interrelated.

¹Commit i: <https://github.com/elastic/elasticsearch/commit/8db9a971>

²Commit i+1: <https://github.com/elastic/elasticsearch/commit/6dace47>

³Commit i+2: <https://github.com/elastic/elasticsearch/commit/b3804c4>

Additionally, in Commit $i + 2$, the developer has removed one obsolete method, namely `analyzeIterable()` and applied Inline Method on `analyzeArray()`, thereby removing this method as well. We hypothesize that these two last code transformations interrelate with the previous ones because the latter were applied on methods created in Commit $i + 1$. At the end, we could state that the developer has applied the batch $b = \{\text{Extract Method}_1, \text{Extract Method}_2, \text{Extract Method}_3, \text{Extract Method}_4, \text{Inline Method}\}$.

Applying batches in practice is quite complex. As illustrated by Figure 2.1, a developer can apply two or more interrelated transformations in conjunction aimed at achieving his motivation. In this case, we assume that the developer motivation behind code refactoring was separating the software features across methods of the `SEach` class. However, applying these sets of transformations can be challenging for developers due to various reasons. Indeed, applying each transformation requires a careful analysis of the code structure and how the refactored code elements interrelate to realize a software feature (8, 20, 25). Developers may also reason about how the applied batch will probably affect the code structure (8).

Developers are often reluctant in applying sets of interrelated code transformations (35). Especially, it has been observed that developers may give up if major refactoring is required in a certain code structure. Part of this reluctance may be reinforced if they do not know the types of effect of (certain types of) batches on code maintenance. Indeed, the aforementioned study reports the real-world developers avoid refactoring because they fear to worsen rather than improve code structures while applying batches (7). Aimed to leverage the current refactoring support and make these developers less reluctant in refactoring their projects, we assume that performing empirical studies about batches is mandatory.

2.3

Poor Code Structures and Code Smells

Maintaining a software project requires from developers to properly read and understand the existing code structures of that project (25, 35). The easier for developers to read and understand a code structure, the higher is to maintain the software project (25). Unfortunately, along the life cycle of any software project, developers will eventually introduce, intentionally or not, poor code structures that threaten the code maintainability (8). Poor code structures are often represented by the so-called code smells (25). Shortly, a code smell is an anomalous code structure that developers should remove whenever possible, otherwise it can make hard to perform maintenance tasks

on the affected code structures (8, 10, 35).

The literature provides developers with several definitions of code smells that vary by type (16, 25). Each code smell type characterizes a particular poor code structure that affects specific program elements, such as methods and classes. Table 2.2 lists the 19 code smell types exploited throughout this Master’s dissertation. We carefully selected types that affect the code structures at two basic levels: *classes*, whose types are presented in Lines 2 to 11 of the table, and *methods*, whose types are presented in Lines 12 to 20.

Table 2.2: Code Smell Types Exploited by this Dissertation

Code Smell Type	Definition based on (25) and (38)
Brain Class	Class overloaded with features
Class Data should be Private	Class that overexposes attributes
Complex Class	Too complex classes with sophisticated logic
Data Class	Too simple class (only data persistence logic)
God Class	Too many features centralized within a single class
Large Class	Too long class
Lazy Class	Class whose features do not pay off the class existence)
Refused Bequest	Child class refuses to use feature from parent class
Spaghetti Code	Class with complex control structures
Speculative Generality	Useless abstract class
Brain Method	Method overloaded with features
Dispersed Coupling	Method that calls too many methods
Divergent Change	Method that often changes when other change
Feature Envy	Method envying features of classes other than its host
Intensive Coupling	Method that depends too much from a few others
Long Method	Too long and complex method
Long Parameter List	Too many method parameters
Message Chain	Too long chain of method calls across the classes
Shotgun Surgery	Method whose changes affect many methods

Various studies such as (53) and (73) provide evidence that developers actually perceive code smells as threats to code maintenance. Real-world developers surveyed by these studies pointed out that code smell types like Large Class and Long Method make hard to read, understand, and modify certain code structures. As a consequence, code smells may cause delays in the delivery of new software features, besides hindering bug fixes and other maintenance tasks (40, 72). The bottom line is that developers should identify and remove code smell instances from their projects whenever possible (17, 48).

2.4

Final Remarks

This chapter summarized key concepts to the understanding of this Master’s dissertation. First, we overviewed code refactoring based on the variety of transformation types and developer motivations reported by the literature. Second, we discussed why batch refactoring is a recurring practice,

and we illustrate the complexity of applying batches in real settings based on a real-world example. Finally, we discuss how code smells can indicate poor code structures that are actually harmful to code maintenance. We have focused on types of code transformations and code smells that are further exploited along the empirical studies presented in Chapters 3 and 4.

The next chapter introduces the first empirical study that composes this Master's dissertation. Such study aimed to summarize the current knowledge of characteristics and types of effect of batches on code maintenance. Our major goal was solving the current knowledge fragmentation about batches in order to guide future research on the topic.

Code refactoring is a key practice for companies concerned about enhancing the maintainability of their software projects (35, 46, 60). Development teams and developers worldwide have taken advantage of refactoring for improving certain code structures that became hard to read and modify along the life cycle of their projects (35). Refactoring is performed through the application of program transformations (25). These transformations can at least partially make code structures easier to read and modify (8, 10, 25). However, applying code refactoring is quite complex in practice. Developers often apply two or more interrelated code transformations in conjunction rather than a single transformation in isolation (46). About 40-60% of transformations are applied in batches, and this phenomenon has been called *batch refactoring* (7).

Although batch refactoring is quite common in practice, little is known about batches. Each researcher seems to have a particular view of what constitutes batch refactoring. Moreover, most of these views lack empirical ground. Due to knowledge fragmentation, previous studies are often conflicting about what does characterize a batch. For instance, certain studies (46, 64) consider that all code transformations within a batch have the same type. Conversely, other studies (35, 43) assume that batches may be composed by two or more code transformations with different but complementary types. In summary, a key question remains without being properly answered: *What does actually constitute a batch?* Unless solid information is obtained about the batch characteristics, researchers will find it hard to both identify and recommend batches in the context of real software projects.

More critically, the current knowledge on batch refactoring helps little in understanding the batch effect on the code maintenance. Again, the knowledge fragmentation across studies generates lots of conflicting viewpoints about what to expect from the batch application in practice. On the one hand, previous publications such as (25, 35) assume that the application of batches often removes poor code structures, which would otherwise threaten code maintenance. On the other hand, certain studies like (7, 43) argue that certain batches have the opposite effect, thereby harming the code maintenance. In this particular case, one question remains: *What is the structural effect of batches*

on real software projects? The limited scope and conclusions of past research makes hard to guide developers along their daily refactoring practices.

In this chapter, we present and discuss the results of a literature review of batch refactoring. Our study goal is threefold. First, we want to understand what are characteristics associated with batches in the literature so far. *From a researcher perspective, which are the characteristics that constitute a batch?* is the major question that we aim to address. Second, we intend to understand the types of batch effects either mentioned or investigated by previous studies. Third, we aim to identify recurring conflicts among previous studies on either batch characteristics or their expected types of effect. We introduce the very first conceptual map of batch refactoring aimed to summarize the currently fragmented knowledge about the topic and guide future research.

The remainder of this chapter is organized as follows. Section 3.1 describes the literature review protocol. Section 3.2 introduces our study steps. Section 3.3 provides a general view of the papers selected for analysis. Section 3.4 discusses the batch characteristics extracted from the literature. Section 3.5 discusses the types of batch effect assumed by previous work. Section 3.6 presents the conflicts among previous studies on batch characteristics and batch effect. Section 3.7 discusses threats to validity. Section 3.8 concludes this chapter and introduces the next one.

3.1 Literature Review Protocol

We carefully designed and performed our literature review protocol based on previous work (31, 36, 70). We describe below the study protocol.

Study Goal. Based on the Goal Question Metric framework (1), we defined our study goal as follows: *analyze* the literature of batch refactoring; *for the purpose of* (i) summarizing the characteristics of batch refactoring either mentioned or exploited by previous studies, (ii) eliciting the possible types of batch effect on software projects according to the assumptions of previous studies' authors, and (iii) identifying eventual conflicts among previous studies about the batch characteristics and types of effect; *from the viewpoint of* software engineering researchers; *in the context of* papers published from 1999, i.e., Fowler's refactoring book publication year (25), to the first half of 2018.

Research Questions. Table 3.1 presents the research questions (RQs) that we designed for guiding our study. **RQ₁** aims at summarizing the fragmented knowledge about the characteristics that constitute a batch refactoring from the viewpoint of previous studies. **RQ₂** aims at eliciting the types of batch effect according to the literature reports. **RQ₃** aims at identifying

eventual conflicts among previous studies with respect to batch characteristics and types of effect on code maintenance. With our three RQs, we expect to support researchers in guiding their future research. For instance, researchers could prioritize addressing the conflicting knowledge in their future empirical studies.

Table 3.1: Research Questions of the Literature Review

ID	Description
RQ ₁	Which batch characteristics have been reported by previous work?
RQ ₂	Which batch effect on software projects are assumed by previous work?
RQ ₃	Do previous studies assume conflicting batch characteristics and effects?

Paper Search Settings. We chose three web search engines to collect papers: ACM Digital Library¹, Google Scholar², and IEEE Xplore³. ACM and IEEE provide a wide variety of publications in computer science, including software engineering. Google Scholar is a general purpose engine that can complement the search made through the other engines. By using each engine, we selected papers that were: (i) published from 1999 to the first half of 2018; (ii) published in workshops, symposia, conferences, or journals ranked as at least B2 based on the Brazilian Qualis classification system⁴; (iii) have been written in English; and (iv) are available online to download.

3.2 Study Steps

Figure 3.1 presents the ten study steps designed to guide our literature review. We describe each step as follows.

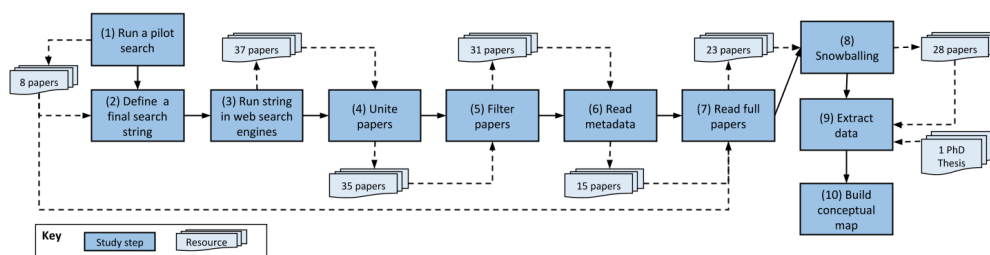


Figure 3.1: Steps of the Literature Review

Step 1: Run a pilot search. We have performed a pilot search in early 2017 as a preparation to the literature review. We retrieved the top-20 most relevant

¹<https://dl.acm.org/>

²<https://scholar.google.com>

³<https://ieeexplore.ieee.org>

⁴<https://sucupira.capes.gov.br/sucupira/> (in Portuguese)

papers from ACM Digital Library and Google Scholar only in order to assess the most common key-terms used by these studies. We run a three-keyword based search string based on our personal knowledge of batch refactoring: (*sequence of refactoring** OR *batch refactoring** OR *continuous refactoring**). The “*” character indicates the inclusion of any variant words whose prefix precedes this character. We have complemented our pilot search with a snowballing procedure based on the references of each retrieved paper (31). As a result, we selected eight papers listed in Table 3.2. These papers were considered in our final set of papers for analysis because, after reading them, we noticed the papers were explicitly addressing batch refactoring. Most of them were published in journals or conferences with high reputation, including IEEE Transactions on Software Engineering (TSE) and International Conference on Automated Software Engineering (ASE).

Table 3.2: Papers Selected through the Pilot Search

Paper Title and Reference
Searching for opportunities of refactoring sequences: Reducing the search space (55)
Algebraic and cost-based optimization of refactoring sequences (37)
Search-based refactoring based on unfolding of graph transformation systems (57)
How we refactor, and how we know it (46)
Identifying refactoring sequences for improving software maintainability (43)
An empirical study of refactoring: Challenges and benefits at Microsoft (35)
FaultBuster: An automatic code smell refactoring toolset (64)
Designing and developing automated refactoring transformations (65)

Step 2: Define a final search string. Based on the pilot search, we decided to discard *continuous refactoring* from our search string, because many of the papers have used this term with purposes that extrapolates the scope of batch refactoring. Thus, we have refined our literature review and the final search string is: (*sequence of refactoring** OR *batch refactoring**). We justify each term as follows. *Sequence of refactoring* is commonly adopted by previous work such as (43, 55, 57, 64) to refer to interrelated code transformations applied in conjunction. Additionally, *batch refactoring* has been more recently adopted by the literature (7, 46, 65).

Steps 3 to 7: Select papers for analysis. By running our search string on the three web search engines (**Step 3**), we have identified a total of 37 papers, from which: 11 papers were retrieved from ACM Digital Library, 20 from Google Scholar, and six papers from IEEE Xplore. When uniting all papers (**Step 4**), we identified two duplicates that, after removal, resulted in a total of 35 papers to be analyzed. When applying our selection criteria on the papers (**Step 5**), 31 papers remained for analysis. After reading the metadata for each paper

(**Step 6**), a half of the paper was discarded. Thus, only 15 papers remained to be fully read. Finally, after the full-text read (**Step 7**) of all 15 papers plus the eight paper retrieved from the pilot search, none of them was discarded. At this point, our data set consisted of 23 selected papers.

Step 8: Snowballing. We have followed an existing guideline (31) to perform additional paper research through snowballing procedures. We applied a single round of backward snowballing procedures to retrieve the previous works that already mentioned batch refactoring. Our goal to apply backward snowballing was to understand: (i) how was the past knowledge characteristics and effect of batches through the previous works; (ii) how this knowledge can contributed for future citations of these characteristics and effect of batches. Basically, we analyzed the list of papers cited by each study in order to complement our data set. In the end, we have obtained five additional papers. Thus, our data set was composed by a total of 28 research papers for analysis.

Step 9: Extract data. We relied on a previous work (36) to define which data could help us in addressing our RQs. Table 3.3 lists the data types extracted for each selected paper. All data was extracted from each paper, plus a recent doctoral thesis (7) about batch refactoring defended later in 2018. We decided to add this thesis due to the extensive investigation of both batch characteristics and the study targeting the effect of batches on code smells. We will refer to this thesis as a regular paper for convenience.

Table 3.3: Data Types Extracted from Each Paper

Data type	Description
Paper Title	Full paper title
Authors	Full authors list
Year	Year of publication
Type	Paper type, e.g., conference and journal
Venue	Full name of the publication venue
Abstract	Full abstract as constant in the paper
Keywords	Paper keywords
Batch-related Term	Synonyms of batch refactoring used by the paper
Batch Characteristics	Any characteristics that constitute a batch refactoring

Step 10: Build conceptual map. We have applied some basic Grounded Theory procedures (13, 63) on the extracted data. Basically, we performed both *coding* and *classification* on excerpts extracted by paper in order to identify the batch characteristics and types of effect. We followed a four-step protocol: (i) we tabulated sentences that mention batch characteristics or effect; (ii) two

researchers validated the tabulated sentences in order to assure they mention characteristics and types of effect; (iii) two researchers together grouped the sentences by semantics, thereby extracting final characteristics and types of effect; finally, (iv) we built a conceptual map of batch characteristics and types of effect based on the current knowledge.

3.3

Conceptual Map of Batch Refactoring

Table 3.4 presents the full list of 29 papers (with the PhD thesis included and assigned with “*”) selected for analysis. By the paper titles, we observed that most of the previous studies have either proposed or evaluated optimization strategies for composing batch refactoring in practice. In fact, understanding how to compose batches that achieve varied goals while enhancing code structures is important to support developers in their daily work (20).

Table 3.4: Papers Selected from the Literature Review

Paper Title and Reference
A methodology for the automated introduction of design patterns (11)
A new software maintenance scenario based on refactoring techniques (69)
Algebraic and cost-based optimization of refactoring sequences (37)
An empirical study of refactoring: Challenges and benefits at Microsoft (35)
Composite refactorings for Java projects (12)
Designing and developing automated refactoring transformations (65)
DRACO: Discovering refactorings that improve architecture using fine-grained co-change dependencies (14)
Evolving transformation sequences using genetic algorithms (15)
Experimental assessment of software metrics using automated refactoring (47)
FaultBuster: An automatic code smell refactoring toolset (64)
How we refactor, and how we know it (46)
Identifying refactoring sequences for improving software maintainability (43)
Improving refactoring speed by 10x (34)
Interactive and guided architectural refactoring with search-based recommendation (39)
Pareto optimal search based refactoring at the design level (30)
Recommendation system for software refactoring using innovation and interactive dynamic optimization (44)
Refactoring with synthesis (58)
Scripting parametric refactorings in Java to retrofit design patterns (33)
Search-based detection of high-level model changes (5)
Search-based refactoring based on unfolding of graph transformation systems (57)
Search-based refactoring detection (41)
Search-based refactoring using recorded code changes (50)
Search-based refactoring: Towards semantics preservation (49)
Searching for opportunities of refactoring sequences: Reducing the search space (55)
Template-based reconstruction of complex refactorings (56)
The use of development history in software refactoring using a multi-objective evolutionary algorithm (51)
TrueRefactor: An automated refactoring tool to improve legacy system and application comprehensibility (28)
*Understanding and Improving Batch Refactoring in Software Systems (7)
WitchDoctor: IDE support for real-time auto-completion of refactorings (24)

Figure 3.2 presents the paper publication rate across the years, based on the 29 selected papers. We have observed a slight increase in frequency of papers published from 1999 (Fowler’s refactoring book publication year) to 2018 – according to the tendency line that crosses the figure). There seems to

be a growing interest on batch refactoring, which demonstrates the relevance of this topic at least in academic settings.

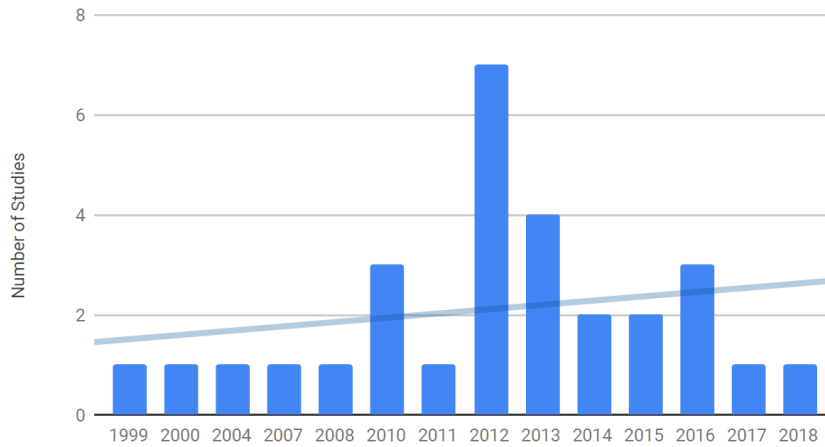


Figure 3.2: Number of Papers by Publication Year

In total, we were able to find 29 studies published in different venues: 20 (70%) out of 29 studies were published in conferences such as International Conference on Software Engineering (ICSE); 3 (10%) out of 29 studies were published in workshops such as the International Workshop on Model-driven Product Line Engineering (MDPLE); 2 (7%) out of 29 studies were published in symposium such as International Symposium on Foundations of Software Engineering (FSE); 2 (7%) out of 29 studies were published in journals such as IEEE Transactions on Software Engineering (TSE); and one (3%) out of 29 studies was published in a Joint Meeting on Foundations of Software Engineering (FSE). Note that one (3%) of out the 29 referred studies is actually a Doctoral thesis (7) and it was not published in conference or journal.

The Conceptual Map. Figure 3.3 introduces our conceptual model of batch refactoring based on the literature. The blue-colored boxes represent those batch characteristics related to the batch *application*, which is about the mechanisms to apply a batch, related to: Who applies it? When was applied it? How many commits was necessary to apply this batch? The gray-colored boxes correspond to characteristics that say something about the batch *structure*, i.e., they regard the internal structure of a batch. The green-colored boxes are the *positive* types of batch effect, i.e., those types whose effect is beneficial to the code maintenance. Finally, the red-colored boxes are the *negative* types of batch effect, that is, those types whose effect is detrimental to code maintenance. We scrutinize the conceptual map content in Sections 3.4 and 3.5.

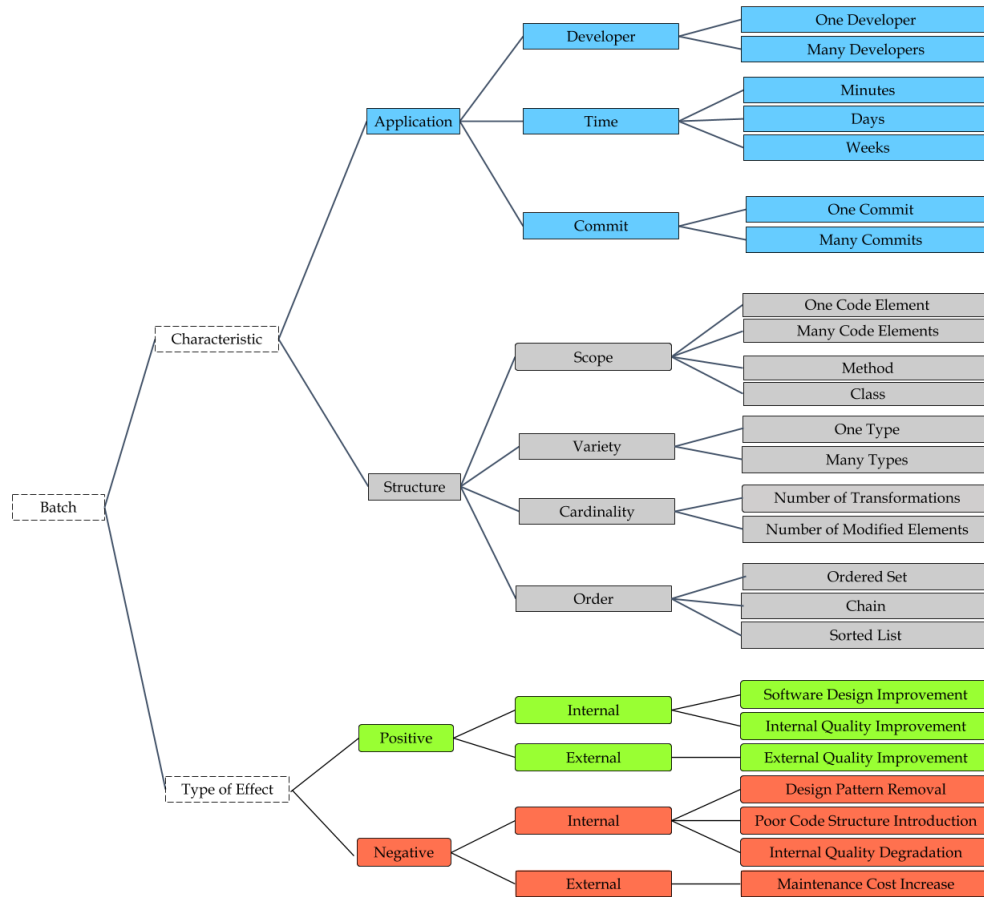


Figure 3.3: Conceptual Map of Batch Refactoring

3.4 Batch Characteristics

The Application group is composed of three batch characteristics: *developer*, *time* and *commit*. These characteristics regard the way who and how to apply batches. Thus, these characteristics say something about the developer practices along the batch application. These characteristics had two or more manifestations described by previous studies. We discuss each characteristic and its respective manifestations as follows.

Developer regards the number of software developers that contribute with the batch application on a software project. Various studies (46, 50, 58) suggest that each batch is applied by only *one developer*. This view is quite reasonable when considering that motivations behind refactoring are often associated with a problem faced by a particular developer (7). However, a few studies like (35) assume that *two or more developers* can work together in order to fully apply a certain batch. This approach works as a response to cases in which developers

join forces to plan and perform a complex batch.

The way how this characteristic manifests in practice strongly depends on some organization aspects. On the one hand, large software projects have entire development teams allocated to refactor code structures, which increases the chances of batches being composed and applied by two or more developers in conjunction. Conversely, small projects may have only a few (or just one) developers allocated to perform code refactoring. By investigating the *developer* characteristic, researchers can better understand how the allocation of developers to refactor the code may eventually affect the quality of the resulting code.

Time regards the time spent by developers for applying a batch in a software project. Only a few studies refer to this batch characteristic, but still, the authors have varied viewpoints. One particular study assumed that each code transformation into a batch should be applied in up to 60 seconds after the previous transformation (46), then the time spent for applying a batch can be *minutes*. Other studies (35, 50, 64, 65) suggest *do not constrain the time* so that consecutive code transformations can be applied at any time.

We have found different manifestations of the *time* characteristic across the studies. They suggest the time computation by measuring working *minutes* (46), *days* (64), or *weeks* (35). In this case, the manifestations of this batch characteristic also depend on specific team or organization practices. It may be the case of developers having too little time to complete their batches due to high demands for other maintenance tasks than code refactoring. Thus, the time span for the batch application can be shorter than in teams or organizations in which developers can spend entire weeks with code refactoring. In summary, we observed a clear opportunity for future research in order to: (i) identify ways to compute *time* in real settings, and (ii) understand the relationship between the resulting quality of a batch and the time spent on it.

Commit regards the how many commits were performed by developers to apply a batch. A study assumes that a batch can be completed in *one commit* (7). However, other studies have mentioned that a batch may take *many commits* to be completely applied (50, 35). Note that how the *commit* characteristic manifests in practice also strongly depends on organizational practices.

The Structure Group is composed of four batch characteristics: *scope*, *variety*, *cardinality*, and *order*. These characteristics regard internal aspects of the batch composition. In other words, these characteristics reflect the internal batch structure. All four identified characteristics had at least two

manifestations mentioned by previous studies. We provide below a detailed discussion about each characteristic and its respective manifestations.

Scope regards the scope of code elements affected by the code transformations that constitute a batch. Certain studies assume that a batch is performed on a single *method* (43, 55), *class* (14, 37, 55). Conversely, other studies such as (35, 50) consider that each batch may affect *multiple code elements* together. Certain limitations may emerge depending on the *scope of code elements* considered for computing a batch. For instance, by assuming that batches are constituted by transformations exclusively applied on methods, the code transformations at attribute and class levels, such as Pull Up Attribute and Extract Class, are overlooked. Thus, the understanding of how batches affect the code structure of a software project is limited to the method scope only.

Variety regards the variety of code transformation types applied along a batch. Some studies (44, 46, 51) consider the *number of occurrences by transformation type* as a means to differentiate batches. Other studies discuss the so-called *refactoring patterns*, i.e., the varied combinations of different transformation types in order to compose a batch (43, 55, 57, 58). The *variety* of transformation types within a batch depends on the assumed *scope* of batches. In fact, if batches strictly affect methods, all transformations at attribute and class levels could be ignored in the batch composition. We highlight that the approach used by a specific study (46) considers only transformations of the *same type*. In this case, certain batches suggested by previous work (43, 44, 57, 58), which combine transformations at different levels to remove code smells, would be ignored. Future work could empirically validate if the variation of types has different effects on code maintenance.

Cardinality regards the extension of a batch applied to the code structure. Previous studies have different approaches for measuring the *batch cardinality*. Many studies (14, 24, 34, 44, 46, 47, 55, 57, 64) rely on *the number of code transformations* that constitute a batch. Other studies like (43) count how many code elements are modified by the code transformations that constitute the batch. These studies do not present empirical evidence about what the common *cardinality* of batches in practice. But, one of the selected studies present that most batches have a *cardinality* with a median of two transformations only (7) on the context of 48 software projects. Besides, only one study has speculated an upper limit to the *batch cardinality* (58). However, we understand that the lack of consensus about which code elements are usually affected by batches (see *Scope*) makes hard to compute the *cardinality* of batches.

Order regards on how the code transformations are organized in a batch. Previous studies such as (28, 37, 43, 57) considered batches as two or more code transformations whose their order does matter to distinguish one batch from another. Certain studies state that transformations within a batch are *ordered*. These studies refer to batches as *chains* (12), *ordered lists* (43, 49, 50), and *ordered sets* of code transformations (5, 12, 58, 69). The transformation order can be influenced by the developer motivation behind the application of a batch. For instance, applying an Extract Method before a Move Method can be used to remove a Feature Envy smell (25). The application of these transformations in the reverse order would not remove such a smell. Other study like (35, 7) considers that the transformation order within a batch does not matter. These studies that do not consider the *order* to batch computation suggest that this approach facilitate the computation of applying batches from software projects that use platforms of version control such as Github. In fact, project version data provided by platforms like GitHub are commit-focused and provide little or no data about what happens within a commit. Thus, it is quite hard to precisely characterize the order of transformations within a commit.

3.5

Batch Effect on Software Projects

Three positive types of batch effect were either mentioned or exploited by previous studies: *software design improvement*, *internal quality improvement*, and *external quality improvement*. These types of effect reflect an enhancement of design, internal and external quality aspects of software projects. With design level, we mean that batches can have an effect on the architecture or detailed design. With internal aspects, we mean any characteristic of the internal code structure of a software project, such as coupling, cohesion and code complexity. With external aspects, we mean those characteristics of a project that are manifested externally to the code structure, usually by means of requirements that system users can interact with. We list and define each positive types of batch effect as follows.

- *Software Design Improvement* regards the effect that batches have on the architecture or detailed design level. Various studies assume that, by applying batches on the software projects, developers can achieve a better software architecture. Such enhancement can manifest in terms of *design metrics* (39) or *architectural quality indicators* (14, 34, 39, 50, 57) such as *low coupling* or *high cohesion*, and *architecture decomposition* (14, 39)

(specifically, a division of architecture's modules). These design improvement assumptions are quite reasonable if we consider that removing poor code structures from a project, through a set of transformations, can also help to eliminate high-level design problems (62).

- *Internal Quality Improvement* regards the possible effect of batches on the internal code structures. Some studies assume that applying batches can positively affect the code elements that constitute a project, thereby improving software metrics such as code size, code coupling that can be indicators of a maintainability improvement (10). This assumption is quite expected due to the traditional expectation that single code transformations can affect the indicators of code maintainability as well. Such improvement can be assessed by means of *code metrics* (14, 35, 44, 49, 50) or special indicators of poor code structures like *code smells* (28, 37, 43, 44, 51). Certain studies assume batches can remove code smells (28, 37, 43, 44, 51) even without empirical evidence. In fact, existing studies have observed that single transformations (8) can remove certain types of code smells. However, other studies (7, 46) have also reported that most of the transformations are applied in batches. Thus, the effect that was observed for single transformations can be different from the effect observed from the batch perspective.
- *External Quality Improvement* regards the effect that batches may have on external attributes of software projects, such as evolvability. We have found a considerable number of studies that assume that batches can leverage the project *evolvability* (35, 43, 51), and *correctness* (49, 50). Transformations that constitute a batch directly affect the code structure and, therefore, it is expected an internal quality improvement of software projects. However, studies show that each code transformation can also affect external quality attributes such as correctness through code transformations that aim to prevent code elements from becoming buggy in the future (2). Thus, it is reasonable that some authors assume an external effect of batches.

Four negative types of batch effect were identified in the literature: *design pattern removal*, *internal quality degradation*, *poor code structure introduction*, and *maintenance cost increase*. These types reflect different aspects of decay in code maintainability. We introduce and discuss each characteristic as discussed by previous studies as follows.

- *Design Pattern Removal* regards the unexpected effect of removing design patterns (26) that leverage the code maintainability after applying the interrelated code transformations that constitute a batch. A previous work (39) reports that some batches may non-intentionally remove a design pattern from a project, thereby harming the code maintainability. In fact, design patterns are realized by certain code structures with the additions required by each software project in particular. Thus, the assumption provided by the previous work (39) is reasonable, once code transformations target code structures that may realize design patterns.
- *Internal Quality Degradation* regards how batches may negatively affect the internal code structure of software projects. A previous study (47, 43) discusses that batches may negatively affect certain parts of the code structure, especially in terms of basic code metrics.
- *Poor Code Structure Introduction* regards the negative effect of batches on the code structure of software projects. In fact, one study (43) has warned that an undisciplined application of batch refactoring might degrade the internal code structure of a project, especially by introducing code smells. This phenomenon is also quite interesting and has been explored by a very recent PhD thesis (7). Particularly, the thesis author has shown that most batches tend to either introduce (5%) or not fully remove (89%) code smells. However, additional studies are still required to understand to what extent the code transformations that constitute a batch are the root-cause for such poor code structure introduction.
- *Maintenance Cost Increase* regards the effect on applying batches on the effort spent by companies and software developers with future maintenance tasks. Software maintenance is highly expensive in practice (75) and code refactoring has been employed by developers for decreasing future maintenance costs. Thus, it is quite intriguing that a previous work (43) points out that certain batches can harm the project maintainability when improperly applied in practice. This particular phenomenon should be further investigated by future work.

3.6

Conflicting Batch Characteristics and Types of Effect

We have found some cases of conflict among studies with respect to the characteristics that constitute a batch. We discuss below each of these cases.

Conflict 1: Scope of a batch: one code element or several code elements? The current knowledge about what code elements are affected by a

batch refactoring is ultimately conflicting. Some studies consider that the code transformations constituting a batch should be constrained to the *same code element*, e.g., a method or a class (14, 37, 43, 55). Conversely, other studies assume that a batch refactoring can affect *multiple classes* (35, 50, 7). Each study may have adopted a different manifestation of the scope characteristic because it could facilitate their study goals. However, these studies did not explain why they did not use other manifestations. There is a need for a proper understanding of what are boundaries that determine the code elements affected by a batch refactoring. Otherwise, it is hard to elaborate or choose a heuristic to identify existing batches in a software project.

Conflict 2: Who is responsible for applying a batch refactoring?

There is also a conflict on the current knowledge about who is usually responsible for applying the code transformations that constitute a batch refactoring. Some studies assume that a batch refactoring is usually applied by a *single developer* (46, 50, 58, 7). However, another study assumes that a batch refactoring can be started by one developer and complemented by other developers (35). We have found that each study may have used different manifestations of the developer characteristic to facilitate the investigations of their study goals. For instance, Kim et al. (35) have investigated on the refactoring practices of a single *team of developers* in a specific software company. Murphy-Hill et al. (46) have investigated on batches that were applied in a specific timespan using the Eclipse IDE⁵. This work only took into consideration the batches applied by a single developer. Cedrim (7) has investigated batches as a set of interrelated code transformations performed by a single developer. In summary, there is a lack of consensus about how many developers are responsible for applying a batch refactoring. This lack of consensus makes it difficult to identify batches in existing projects.

Conflict 3: How long does applying a batch refactoring last?

Our literature review also identified a conflict regarding the *time* spent by developers to apply a batch refactoring. For convenience, one study assumes that two subsequent code transformations that compose a batch refactoring should be applied in up to 60 seconds one from another (46). One might question whether this conservative threshold is reasonable. For instance, this work unlikely captures batches that last for more than an hour as a developer often has to intertwine refactoring edits with his other routine activities. Conversely, other studies *do not constrain the time spent* to apply these code transformations, thereby making possible to compute batch refactorings that last *days, weeks*, and even *months* to be completed (35, 50, 64, 65, 7). In

⁵IDE - Integrated Development Environment

these cases, the lack of a threshold may lead one to the consideration of non-cohesive batches. For example, there might be cases where two groups of transformations are interrelated because they affected the same program element. However, the time separating the two commits (where each group of transformations took place) is higher than one year. If there is no time-related constraint, one would consider these two groups of transformations as composing a batch. In any case, existing studies lack empirical evidence about how long should the application of a batch refactoring last in practice. Thus, we have insufficient information to constrain the set of code transformations that constitute a batch refactoring. Consequently, it makes difficult to identify "cohesive" batches applied to exist projects.

Conflict 4: A single or various transformation types? We have observed some conflicts in the way previous studies consider the *variety* of transformation types that may constitute a batch refactoring. Some studies suggest that *multiple transformation types* can co-occur into a batch refactoring (44, 55, 58, 7). Conversely, another study assumes that the code transformations that constitute a batch refactoring should have a *single type* (51). This conflict can affect the analysis of the effect of batches in a program. If a heuristic is used in a empirical study to detect only batches with a *single transformation type*, the conclusions will be constrained to only these types of batches. Batches of this kind might be not effective to remove various types of structural problems in the source code.

We also found cases of conflict among studies with respect to the expected effect of batches on software projects. We discuss these cases as follows.

Conflict 1: Do batches make a software project easier to maintain? Some studies assume that batches can effectively improve the maintainability of software projects (35, 43, 51). They mainly expect an *internal code structure improvement* through the *removal* of code smells (43). On the other hand, one study (7) has presented batches frequently can either *introduce* or *end up not removing* code smells. In order to better support developers in applying batches that are effective in making the code easier to maintain, future work should empirically investigate these positive and negative types of batch effect in depth. In fact, the study of Meananeatra (43) presents that certain forms of batches can remove code smells, thereby improving the program maintainability. However, Cedrim (7) presents that batches often have a negative effect on the code smell removal. Thus, the current knowledge is limited and conflicting.

Conflict 2: Are batches more likely to improve internal code structures rather than degrade these structures? We have found studies,

such as (35, 44, 51), that point out batches as means for *improving the internal quality of software projects*. Conversely, a particular study (47) discussed that undisciplined batch application can lead to the *degradation of code structures*. Similarly to Conflict 1, we expect that future work can address this particular issue in order to draw more assertive conclusions about the batch effect on the internal quality of software projects.

Conflict 3: What is the actual batch effect at the architectural level? Some studies (14, 34, 57) assume that batches likely affect *negatively the current architecture of software projects*. These studies suggest that batches can *increase the coupling* between modules of a software architecture. On the other hand, a particular study (39) proposes that batches can be applied to *improve the software architecture* by improving the cohesion and the coupling of the software's components. Thus, there is some lack of consensus on how batches affect the architecture of software projects.

3.7

Threats to Validity

We discuss threats to the study validity (70) as follows.

Construct Validity. We carefully defined our study protocol prior the conduction of the literature review. We defined the study goal and research questions according to the Goal Question Metric framework (1). Thus, we expected to minimize the chances of changing the focus of our study while the literature review was performed and the data were analyzed.

Internal Validity. All data collection procedures were performed by a pair of researchers in order to mitigate problems with missing, duplicated, and invalid data. These procedures include running our search string in the web search engines, for instance. Similarly, we paired two researchers in order to tabulate the data so that we could easily perform the Grounded Theory (GT) procedures of open and axial coding (13). By strictly following the GT procedures, we expected to reduce problems with the identification of batch characteristics and types of effect from the selected papers through the literature review.

Conclusion Validity. We carefully analyzed all tabulated data in order to: (i) aggregate the batch characteristics and types of effect extracted from the literature in our conceptual mapping, and (ii) validate all analyzed data in a pair. Thus, we expected to minimize biases in the identification of characteristics and types of effect, but especially in the characterization of conflicts among previous studies. In this particular case, we have promoted meetings to discuss which characteristics and types of effect are conflicting

and deserved an explicit consideration.

External Validity. We have performed a limited search for papers in the selected web search engines. As a consequence, our final set of papers used to analyze and extract batch characteristics, effect types and conflicts may not represent the whole current body of knowledge on batches. In order to mitigate this threat, we have applied snowballing procedures in order to collect as many papers as possible that have not being included through the search in web engines.

3.8

Final Remarks

In this chapter, we presented the results of a literature review of batch refactoring. We aimed to summarize the current knowledge on batches for: (i) eliciting the characteristics that previous studies consider as constitutive of a batch refactoring, (ii) eliciting the types of batch effect on software projects as assumed by previous studies, and (iii) identifying eventual conflicts in the literature about batch characteristics and types of effect. Our study enabled us to identify seven batch characteristics, seven batch types of effect, and a considerable number of conflicts that spot opportunities for future work.

The next chapter presents a quantitative study aimed at assessing the frequency of certain characteristics of batch refactoring. For this purpose, we present our first attempt at combining some of the characteristics elicited by our literature review into a heuristic for identifying batch refactoring in existing projects. With this heuristic, we have: collected batch refactorings applied in 57 closed and open software projects. Then, we classified batches in categories according to their batch characteristics. At the end, we have analyzed the effect of batches on poor code structures represented by various types of code smells.

In Chapters 2 and 3, we have introduced the concept of a batch refactoring as the application of two or more interrelated code transformations along with code refactoring (7, 46). Each set of interrelated transformations forms a batch. A previous study shows that batches are frequently applied by developers on their software projects (7). Unfortunately, some developers are still reluctant to apply batches in practice (35). The reason is due to the scarce empirical knowledge about the effect of batches on code structures. Consequently, developers may not benefit of batches to enhance code structures of their projects.

Code smells represent code structures that are often hard to read and modify (25). Thus, code smells indicate poor program structures that harm code maintenance. In fact, past research has shown that developers actually perceive code smells are harmful to code maintenance (53, 73). In summary, code smells should be removed from the source code whenever possible (40, 48). Code refactoring has been largely advertised as capable to remove code smells. However, a single code transformation performed in isolation is rarely enough to fully remove a code smell (8). Hence, one could wonder whether smells are more often removed when developers combine code transformations in batches. It might also be that batches often have a negative effect. Given their complexity, they might end up introducing, rather than removing code smells. Unfortunately, there is little empirical evidence about the effect of batches on code smells.

To the best of our knowledge, a recent doctoral thesis (7) was the first study aimed to empirically assess (i) how batches often manifest in practice, based on presumed batch characteristics, and (ii) what is the effect of batches on code maintenance in terms of code smell introduction and removal. However, that study has a few limitations that should be addressed to reveal more precisely how batches manifest and affect code smells in real software projects. The previous study did not rely on the current knowledge of batches provided by the literature to deeply understand the batch characteristics and their manifestations. Besides, this study did not analyze the relationship between batch characteristics and their effect on code smells. These gaps

limit the understanding about how developers apply batches in practice, on what the code transformation types that often compose batches, and whether the manifestations of a batch can be related to its effect on the code smell introduction or removal.

In this chapter, we present a large-scale empirical study that extends the previous study proposed by the aforementioned doctoral thesis (7). Our study goal is two-fold. The first goal is analyzing the manifestations of four batch characteristics most frequently mentioned by previous work but without a definite understanding on how they manifest in practice. The second goal is acquiring an empirical understanding of batch characteristics and the effect of batches on code smell introduction and removal. We chose this effect of batches because: (i) the literature showed a major concern about it in practical settings (7, 35, 43), (ii) we observed non-ignorable conflicts in the literature concerning whether batches are able to fully remove code smells (Section 3.6); and (iii) the aforementioned doctoral thesis (7) lacked an in-depth investigation about the relationship between batch characteristics and code smell introduction or removal.

Through a quantitative study with 57 open and closed software projects, we expect to reveal insights about batches in practice. We also expect our study outcomes to support future research on batch recommendations to developers. The remainder of this chapter is organized as follows. Section 4.1 describes our empirical study settings. Section 4.2 presents our study results on (i) the characteristics and effect of batches on code smells, (ii) how existing recommendations for batch refactoring are applied in practice, and (iii) how existing automated tools for batches can be used. Section 4.3 discusses threats to the study validity. Section 4.4 concludes the chapter by summarizing our results and their implications in practice.

4.1 Study Design

This section introduces the study goal and research questions that we designed to investigate characterization and classification of batches and types of batch effect.

4.1.1 Goal and Research Questions

We defined our study goal according to the Goal Question Metric (GQM) template (1) as follows: *analyze* batch refactorings performed by developers on their programs; *for the purpose of* understanding how batches

manifest in practice and how they affect code smells; *with respect to* batch characteristics mentioned by previous work and the effect of batches on code smell introduction and removal; *from the point of view of* software engineering researchers; *in the context of* 4,607 batches applied by developers on 57 open or closed software projects implemented in Java. All batches were categorized in terms of their characteristics as summarized in our literature review (Chapter 3). We considered all 13 types of code transformations (Table 2.1) and 19 code smell types (Table 2.2) listed in Chapter 2. We designed four research questions (**RQ_s**) in order to address our study goal.

RQ₁. *How often do batch characteristics manifest in real software projects?* – This research question intends to reveal the frequency of the manifestations of certain batch characteristics. Our literature review (Chapter 3) revealed seven batch characteristics. Three characteristics (*developer*, *time*, and *commit*) regard the application of batches performed by developers. The other four characteristics (*variety*, *cardinality*, *scope*, and *order*) regard the internal structure of a batch. These characteristics can manifest differently in each actual batch instance. For instance, the characteristic *variety* has two manifestations: *one type* (i.e., one batch is composed by code transformations that share a common type) and *many types* (i.e., one batch is composed by code transformations with varied types). A batch can be classified according to these manifestations. An investigation on these characteristics and their manifestations can reveal how often certain forms of batches are applied in practice by developers.

We selected four batch characteristics based on our literature review (Chapter 3) to investigate in this study: *commit*, *scope*, *variety*, and *cardinality*. We followed two criteria for selecting these four characteristics. First, the characteristics had to be feasible to compute based on data publicly available at GitHub repositories. This criterion has made us to discard the *time* and *order* characteristics since we are not able to track the application order and application time between code transformations within a single commit. Most batches (70.7%) indeed occur in a *single commit* (7). Second, the characteristics had to be incorporated by our batch computation heuristic used to collect batches in real projects (Section 4.1.2). We relied on an heuristic that computes batches based on the *developer* characteristic. This heuristic considers a batch has to be fully applied by the *same developer*. Thus, it does not make sense to investigate the manifestations of this characteristic. Our heuristic does not detect batches performed by *multiple developers*.

Table 4.1 describes the possible manifestations of each batch characteristic. Previous studies (35, 43, 46, 64, 65) have focused on some of these

manifestations based on assumptions made by the authors. They have not empirically evaluated the frequency of these manifestations. It might be the case that certain manifestations rarely occur in practice. In this work, we aim to address the lack of empirical knowledge by assessing the frequency of such manifestations through the analyses of the software projects.

Table 4.1: Manifestations of each batch characteristic

Charact.	Manifestation	Description
<i>commit</i>	<i>one commit</i>	Batch completed in only one commit
	<i>many commits</i>	Batch completed in two or more commits
<i>scope</i>	<i>method</i>	Batch transformed only one method
	<i>class</i>	Batch transformed multiple methods or class
<i>variety</i>	<i>one type</i>	Batch has only one transformation type
	<i>many types</i>	Batch has two or more transformation types
<i>cardinality</i>	2	Batch has two transformations
	≥ 3	Batch has three or more transformations

RQ₂. *What is the nature of code transformations that usually compose a batch in practice?* – One of the batch characteristics that we selected for exploration in RQ₁ is *variety*. A batch can be composed of transformations of a single or multiple types. We analyzed the frequency of each of these manifestations (RQ₁). However, this analysis provides only a starting point towards the understanding of the nature of transformations often comprising batches. The nature of a transformation defines how this transformation operates on a code element. For instance, some transformation types have the Extraction nature. The transformation types of this nature (e.g Extract Method) extract a source code from a code element, and create another code element with the extracted source code.

An understanding on the natures of code transformations in batches is relevant to know how developers compose batches. In particular, we can reveal: (i) what are the most frequent natures of transformations used by developers to compose a batch, and (ii) whether developers compose batches with transformations of the same nature or from different natures. Through RQ₂, we aim to conduct a deeper analysis focused on the natures of code transformations that often compose batches.

Table 4.2 presents the natures derived from the 13 code transformation types listed in Table 2.1 and investigated in this dissertation. The first column presents the identifier of each nature that we will use throughout this chapter. For instance, the identifier EX represents the Extraction nature. The second column characterizes the possible natures in six categories: Extraction, Inline, Motion, Pull Up, Push Down, Renaming. For instance, Extraction regards all code transformations that a extract new code element (e.g., a method) from (parts of) an existing code element. Each nature may encompass code

transformations that modify the source code at different levels. For instance, Extraction includes transformation types affecting methods (e.g., Extract Method) and classes (e.g., Extract Class).

Table 4.2: Nature of Transformation Types

ID	Nature	Description	included Transformation Types
EX	Extraction	The nature that extracts a source code from a code element to another code element	Extract Method, Extract Class, Extract Interface, and Extract Superclass
IN	Inline	The nature that removes a code element and transfer its source code to an existing code element	Inline Method
MO	Motion	This nature moves a code element from a source code to another source code	Move Attribute, Move Method, and Move Class
PU	Pull Up	This nature aims to pull up a code element from a subclass to a super class	Pull Up Attribute and Pull Up Method
PD	Push Down	This nature aims to push down a code element from a superclass to a subclass	Push Down Attribute, and Push Down Method
RE	Renaming	A nature that renames a code element	Rename Attribute, Rename Method, and Rename Class

RQ₃. *Are batches more likely to be applied on smelly code elements rather than smell-free code elements?* – The literature reports that code refactoring has been largely employed by developers for many reasons, including the removal of poor code structures represented by code smells (8, 25). In fact, there is empirical evidence that a significant number of refactored code elements were previously affected by code smells (8). The most studies have investigated the application of single transformations on smelly code elements only. There is little empirical knowledge (7) about the relationship between batches and the smelliness of code elements. We aimed to address this literature gap through RQ₃. Thus, we aim to understand whether batches tend to be applied on smelly code elements. In addition, we aim to understand what manifestations of batch characteristics are often applied on smelly or smell-free code elements.

To do so, we classified all computed batches in terms of 16 possible categories, which represent the 16 possible combinations of the characteristics' manifestations (Section 4.1.2). We then analyzed the application of batches in smelly (or smell-free) code elements for each batch category. Our expectation is acquiring an understanding of the typical characteristics of batches that are applied on smelly code elements. A previous study has presented that often batches introduce or do not remove code smells (7). It may be the case of most batches affecting smelly code elements are composed of a *single transformation type*. In this case, we could recommend batches proposed by previous work (7, 25) that combine *multiple transformation types*.

RQ₄. *How do batches affect code smells in software projects?* – Through RQ₄, we aimed to assess the effect of batches on code smells based on a

four-fold analysis. First, we assess the batch effect from the viewpoint of each batch characteristic. Second, aimed at scrutinizing the effect of batches on code smells, we assess such an effect by batch category. Third, we assess the effect of each batch group through the nature of code transformations that compose a batch. Based on insights of the previous analyses, we discuss the effect of batches and their characteristics and classifications. We then compared the effect of batches with the effect of single transformations (8). Moreover, we discuss our results on the effect of batch refactoring in comparison with the effect of batches already previously reported by the literature (7). Besides, we evaluated whether certain forms of batches, which were previously recommended to remove certain smell types (7, 25), indeed had the expected effect in the analyzed projects.

4.1.2 Study Steps and Definitions

Figure 4.1 presents the six steps that we designed for guiding our study. Each step is described as follows.

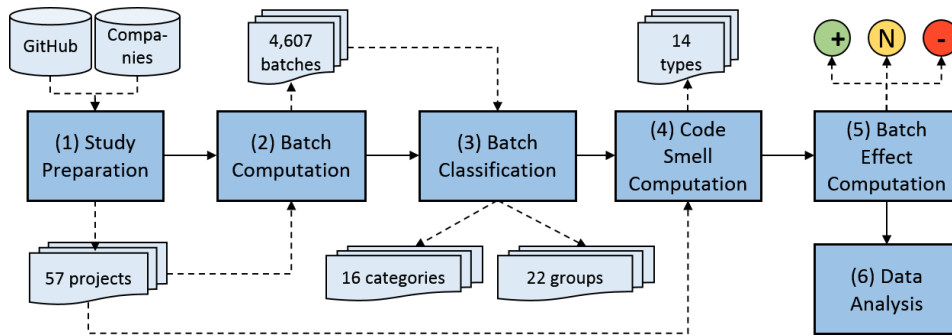


Figure 4.1: Study Steps

Step 1: Study Preparation. This phase consisted of selecting software projects for analysis. We searched for open source software projects available online at GitHub, which should be developed in the Java programming language due to its worldwide popularity¹ and ease of study replication. We sorted all software projects by stars (6) and selected the 100 most popular ones. Thereafter, we filtered the software projects in order to keep those with at least 90% of source code written in Java and a variety of code transformation types previously applied by the developers. The resulting set of 54 open source projects sums up 13,400,686 Lines of Code (LOC) and 151,391 commits. Finally, we added three closed source projects kindly provided to us by three anonymous companies.

¹<https://www.tiobe.com/tiobe-index/>

Step 2: Batch Data Computation. We have used a heuristic (7) to detect batches along the commits of the software projects under analysis. This heuristic detects every batch that satisfies all the following constraints: (i) it consists of a set of two or more transformations, (ii) the transformations are applied on *one code element* - either a *class* or a *method*, and (iii) those transformations are applied by *one developer*. Once the *scope* of each detected batch is constrained to as *single code element*, this heuristic is from now on called **element-based** heuristic.

This heuristic returns a set of batches, each composed of a set of inter-related transformations, i.e., those performed in a *single code element*, which are performed by the *same developer*, even if the transformations affecting the same code element were applied across different versions. For each software project, we consider all the versions that are produced after each commit in the system. This heuristic can also capture batches that are applied at long term, i.e., the set of transformations may span several commits made in the period of months or years. Thus, the heuristic does not take into consideration whether the developer, while touching the element in more than one commit, might be addressing different issues.

Our goal is to analyze how a developer is affecting the structure of a method or class at short or long term after a set of transformations in this element are applied, regardless what are the issues being resolved. In this study, we have not analyzed batches on code elements larger than a class (e.g., a package). The consideration of a method or a class as the *batch scope* is appropriate for assessing the effect of batches on code smells. In fact, the code smells considered in our study are associated with either a class or a method. We plan to study other forms of batches with larger scopes in the future. The consideration of larger scopes (e.g., packages or the set of all elements edited in a commit) may also increase the likelihood of detecting transformations that are not cohesively related.

Step 3: Batch Classification by Nature. We have classified a total of 4,607 batches according to the nature of their transformation types (Section 4.1.1). All batches were grouped based on natures listed in Table 4.2 (Extraction, Motion, etc.) that constitute a batch. Some batches are constituted by a single nature (e.g., Extraction only), but others have multiple natures (e.g., Extraction and Motion combined). From the 4,607 batches and the six natures investigated in this work, we derived 22 groups of batch by nature. This information is used to address our second research question.

Step 4: Batch Classification by Category. In **Step 2**, we have collected a total of 4,607 batches. We have classified all batches in categories

according to their characteristics (Table 4.1). Each *batch category* is defined as $C = [commit, scope, variety, cardinality]$, which is a combination of manifestations for the four batch characteristics listed in Table 4.1. For instance, let us consider a batch referred to as $b = \{\text{Extract Method, Move Method, Extract Method}\}$. b was completed in only one commit and it has transformed two methods through Extract Method and Move Method refactorings. Thus, b belongs to the category $C = [one\ commit, method, many\ types, \geq 3]$. Once we elicited four batch characteristics, each with two possible manifestations, we had a total of 16 batch categories.

Step 5: Code Smell Data Computation. We identified the 19 code smell types listed in Table 2.2 as follows. First, we computed static code metrics, such as Lines of Code (LOC) and Cyclomatic Complexity (CC) (42), via the Understand tool². Second, we combined these metrics in metric-based strategies (18) for identifying code smells. We adopted the strategies, including the metric thresholds, proposed by (4, 38, 40). These strategies were validated by a previous work (4) with a resulting precision and recall (27) of 72% and 81%, respectively.

Step 6: Batch Effect Computation. We computed three possible types of effects that batches have on code smells. The *positive* effect means that the total number of code smell instances in the code elements affected by the batch has reduced after the batch application. Conversely, the *negative* effect the total number of code smell instances in the code elements affected by the batch has increased after the batch application. In the borderline, the *neutral* effect means that even if the code smell types affecting the refactored code has changed, the total number of code smell instances remained unaffected. We opted for an analysis of the code smell introduction and removal because, by definition, program transformations are designed for (at least partially) improving the program structure, thereby potential contributing to remove code smells. However, as typically happens with each single transformation applied in isolation, batches may also tend to not remove code smell instances.

Step 7: Data Analysis. This step consisted of analyzing the collected data in order to address our **RQ_s**. Aimed to answer **RQ₁** and **RQ₂**, we have computed the frequency in which the five batch characteristics summarized in Table 4.1 manifest in practice. In order to answer **RQ₃** and **RQ₄**, we performed two analyses: we computed the probability of each batch characteristic to remove smells; we also computed the the frequency in which batches introduce and remove code smells.

We computed the data distribution for the 57 projects analyzed with

²<https://scitools.com/features/>

respect to four metrics. About the number of code transformations: one quarter of the projects have at least 376 transformations detected, which suggests a considerable refactoring activity across projects. About the number of batches: only a few batches were found in half of the projects (median = 15), but 25% of projects had at least 71 batches; we found this result reasonable given the restrictiveness of our heuristic (see Step 2). About the ratio of batches by code transformations for each project: there seems to be a balance between the number of projects with high ratio and low ratio (minimum = 4, median = 24, and maximum = 33). About the number of commits performed during the life cycle of each project: although values tend to be low (median = 144), 25% of the projects have at least 790 commits.

4.2

Results and Discussions

This section presents our study results. Section 4.2.1 reports on the frequency of each characteristic manifestation (Table 4.1). Section 4.2.2 presents the frequency of the natures of code transformations composing batches. Section 4.2.3 presents the most frequent batch categories applied by developers in 57 software projects. Section 4.2.4 discusses the effect of batch characteristics and their manifestations on code smells.

4.2.1









Manifestations of Batch Characteristics (RQ₁)

Table 4.3 presents the frequency of all possible manifestations by batch characteristic, as documented in Table 4.1. The frequencies are given in terms of the 4,607 batches collected from the 57 software projects under analysis. We discuss the main observations about the frequencies as follows.

Characteristic 1: *commit*. Most batches (93%) require only *one commit* to be completed by the developer. Let us remind that our batch computation heuristic considers all transformations applied on a *single code element* as part of a batch. By combining this information with the high rates of batches completed in *one commit*, one could assume that commits often mark a change in the developers' motivation for program refactoring. This observation is reinforced by the fact that refactoring in the successive changes rarely touch elements coupled to the elements affected by a batch. This observation can also be further validated by interviewing actual developers in future studies.

Characteristics 2 and 3: *scope* and *variety*. Our study results suggest that most batches (91%) affect *multiple methods* into a class and/or the class itself. Only 9% affect a *single method*. This result is somehow expected be-

Table 4.3: Manifestations by batch characteristic

Charact.	Manifestation	# Batches	Frequency
<i>commit</i>	<i>one commit</i>	4,265	93% 
	<i>many commits</i>	342	7% 
<i>scope</i>	<i>method</i>	428	9% 
	<i>class</i>	4,179	91% 
<i>variety</i>	<i>one type</i>	3,330	72% 
	<i>many types</i>	1,277	28% 
<i>cardinality</i>	2	2,605	57% 
	≥ 3	2,002	43% 

cause many code smell types affect *multiple methods* of a class together. Thus, it is reasonable that batches are mostly constituted of transformations affecting various methods (or the entire class) together. As for variety of transformation types, we observed that most batches (72%) consist of transformations with *one type*; still, 28% are batches consisting of *many types*. This result was unexpected because many batch recommendations (25) (39) strongly depend on the combination of different transformation types for fully removing poor code structures, especially code smell instances.

Characteristic 4: *cardinality*. More than half of batches (57%) are constituted of exactly two code transformations. This result is quite intriguing and unexpected. Again, we used a batch computation heuristic that considers all transformations applied by the *same developer* on the *same code element* (without other transformations applied on other elements in between) to form the batch (7). We have calculated the quartile of *cardinalities* aiming to found frequent ranges of *cardinalities* from our dataset (36). The third quartile shows that 411 out of 4,607 (9%) of batches have *cardinality* = 4. Moreover, the fourth quartile has a large concentration around *cardinality* = 5 to *cardinality* = 10 representing 600 out of 4,607 batches (13%). Therefore, we have 1,011 out of 4,607 (22%) batches range from four and ten code transformations. Moreover, 768 (17%) batches has only three transformations. This result suggests that developers may be sub-using the possible combinations of code transformations for removing code smell instances. For instance, in order to fully remove a Large Class instance, developers may have to apply various Extract Class, Move Method, and Extract Interface transformations in conjunction (25). This observation may suggest that developers require proper guidance for performing batches in order to fully remove complex poor structures, such as God Classes.

What is the novelty of our findings? Summary of RQ₁. Through a systematic characterization of batches, we found most batches (93%) occur

in *one commit*. Moreover, we found that most batches are constituted of a *one transformation type*. This latter finding is interesting because recommended batches in Fowler's catalog tend to combine *many transformations types*. In any case, a non-ignorable frequency (28%) of batches combine various transformation types. We also found a considerable amount of batches (22%) with four to ten code transformations. Finally, most batches (90%) affect *multiple methods* into a class and/or the class. These results reveal that many batches affecting a program element are quite complex in practice.

4.2.2

Nature of Code Transformations within Batches (RQ₂)

This section answers our second research question. We classified the detected batches according to six natures (Section 4.1.2) considered in this Master's dissertation. Each nature represents transformations of a similar kind. Each batch may have either a single nature or a hybrid nature (i.e., two or more natures). Table 4.4 presents the frequency of batches according to their single or multiple natures. The first column presents the ID of a batch group, while the second column captures the nature(s) of the transformations in batches of that batch group. The ID of each nature was presented in Table 4.2. The third column presents the amount of batches that belong to a batch group; the amount is followed by the frequency in parentheses. We have found eight most frequent batch groups (G1, G2, G3, G4, G5, G6), which are represented in grey in Table 4.4. These groups are detailed as follows.

Table 4.4: Frequency of Batches According to their Single or Hybrid Nature

Group ID	Nature ID(s)	Batches (%)
G1	EX	1,449 (31.4%)
G2	MO	1,142 (25%)
G3	RE	580 (12.6%)
G4	PU	352 (7.6%)
G5	IN, EX	233 (5%)
G6	IN	221 (5%)
G7	EX, MO	153 (3.3%)
G8	EX, RE	105 (2.3%)
G9	PU, MO	84 (2%)
G10	MO, RE	59 (1.3%)
G11	PD	58 (1.3%)
G12	IN, MO	52 (1.1%)
G13	IN, EX, MO, RE	28 (0.6%)
G14	IN, RE	25 (0.5%)
G15	PU, EX	18 (0.4%)
G16	PD, EX	11 (0.2%)
G17	PD, PU, IN, EX, MO, RE	11 (0.2%)
G18	PU, IN	8 (0.2%)
G19	PU, RE	7 (0%)
G20	PD, PU	6 (0%)
G21	PD, IN	3 (0%)
G22	PD, RE	2 (0%)
	Total	4607 (100%)

Extraction. The most frequent batch group (G1) is defined by the Extraction nature. A tally of 1,449 out of 4,607 (31.4%) batches have at least two extraction transformations. In this group, we found: 1,419 (30.8%) out of 4,607 batches are composed by Extract Methods only; 13 batches are composed by Extract Interfaces only; 14 batches are composed by Extract Methods and other transformation types of extraction; two batches composed by Extract Superclasses only; and only one batch composed by Extract Interface and Extract Superclasses. Besides, developers performed 559 hybrid batches (12%) – out of 4,607 batches – involving both extractions and transformations of other nature(s). This is the case of the following batch groups: G5, G7, G8, G13, G15, G16, G17.

Thus, we observed that Extraction is a nature which developers often apply batches. The most common case (1,449 batches) is the application of Extract Methods only. However, they may also compose extraction-related batches involving more than one extraction-related transformation type and also involving transformation of other nature(s). In group of G7, developers produced 153 batches composed of transformations with both the Extraction and the Motion natures; 43 (out of these 153) batches are composed by Extract Methods and Move Methods. Fowler (25) recommends batches composed by these code transformation types to remove certain code smell types. However, our results suggest that batches composed by these code transformation types are not often applied. Developers also apply hybrid batches involving both extractions and renames (group G8). The most frequent code transformation types, applied in conjunction in this group, are Extract Method and Rename Method. Developers applied 90 batches which extracted and renamed a method. This form of batch represents the cases where developers are extracting to separate the responsibilities of methods, and renaming the method for a most appropriate name according to its responsibility.

On the groups of G15 and G16, developers apply 29 batches of the extraction nature combined with Pull Up and Push Down natures; these natures involve modifications on the classes' hierarchy. Developers are possibly extracting methods to either pulling them up to a superclass or a pushing them down to a subclass. Developers also applied 39 batches involved four or more natures (see groups of G17 and G13) with at least one transformation type of each nature. These batches were probably applied to support complex modifications on the source code, which were required to extract methods or classes. The group of G5 represents another interesting case. It is composed by batches that have Inline Methods in conjunction with extraction transformations. The vast majority (230) of these batches consist of Inline Methods and Extract

Methods.

Motion. The second most frequent group of batches (G2) in our data set consist of transformation types of the motion nature. We found that 1,142 (25%) out of 4,607 batches have at least two motion-related transformations. In this group, we have that 281 (6.1%) out of 4,607 batches are composed by Move Methods only, 583 (12.7%) out of 4,607 batches are composed by Move Attributes only. One batch is composed by Move Classes only, and two batches composed by Move Attributes and Move Classes, that three (0.2%) of 4,607 batches. Besides, 273 (6%) out of 4,607 batches are composed by Move Attributes and Move Methods. This result reveals that developers often move methods or attributes in conjunction. Developers often apply complex batches in which they need to move several methods or attributes for other classes.

Pull Up. The group of G4 has 352 (7.6%) out of 4,607 batches with transformations that pulled up a code element from a subclass to a superclass. We have found that 143 batches are composed by Pull Up Methods only, and 59 batches are composed by Pull Up Attributes only. We have also found that 150 batches were composed by Pull Up Attributes and Pull Up Methods. This can indicate that developers can be often applying batches to improve the hierarchy of classes. In the group G20, developers applied four batches composed by Pull Up Methods and Push Down Methods.

Inline. Developers applied more than 200 batches composed by Inline Methods only (see group G6). This indicates that developers remove several methods in conjunction and move the source code of these methods to existing methods of the same class. Developers can be removing these methods because they are possibly implementing the same responsibility of other existing methods of the same class.

Renaming. The group of G3 has 580 (12.6%) out of 4,607 batches composed by two or more renaming operations. We have found that 573 batches are composed by Rename Methods only. Our results also reveal that developers also renamed methods and their hosting class, but there were only three batches composed by Rename Methods and Rename Class. A previous study has reported that Rename Method is the most common transformation type when is evaluated a single transformation (60). Interestingly, our data reveal that this transformation type is not the most commonly applied in batches.

What is new about this finding? Summary of RQ₂. Our results reveals that more than 56% batches are composed by transformation types of extraction or motion natures. Moreover, we can observe that Fowler's catalog (25), which are aimed to guide developers on improving the code

structure, recommend batches composed of code transformations of different natures. For instance, when a method is more interested in methods of another class, then is recommended to apply a batch composed by Extract Methods and Move Methods, aiming to extract and move the source code of the method for the interested class. However, these recommendations are not very often applied in practice. In our results, we observed that some batch recommendations in Fowler’s catalog, such as Extract Methods and Move Methods only (43 batches only), Extract Superclasses and Extract Interfaces (one batch only) are not often applied.

Our results have also revealed that batches are quite complex because they are composed by code transformations of different natures. For instance, they involve movement of source code among methods and classes as well as pull up or push down methods or attributes among subclasses and superclasses. These code transformations may have a considerable impact on the design as they affecting the coupling or cohesion of multiple classes. This finding might indicate the relevance of existing tools (39) that recommend batches to improve the software design.

4.2.3

Batches Affecting Smelly versus Smell-Free Code Elements (RQ₃)

In this research question, we aim to understand if batches are more likely applied on smelly or smell-free code. To do so, we created the second batch classification in terms of the four batch characteristics and their possible dual manifestations (Section 4.2.1), thereby leading to 16 batch categories. Table 4.5 presents the frequency of batches by category (see Section 4.1.1 for details). The first column lists all 16 batch categories (C01 to C16). Each cell under the first column is composed by the ID of each batch category, followed by the corresponding category details. The second and third columns present the absolute number (Abs.) and relative (%) frequency of each batch category. The third column is relative to the total number of batches (4,607 instances in total) identified in the 57 analyzed software projects.

Our results suggest that the most frequent categories are C01, C02, C05, and C06. Each of these categories comprise at least 448 (10%) of the batches in our data set. Curiously, all these four categories share two characteristic manifestations in common: they occur in the *same commit* (*commit* = *one commit*) and affect just a *single class* (*scope* = *class*). This result reinforces the prevalence of manifestations reported in Table 4.3. Surprisingly, the sum of absolute numbers of batches of all four categories equals 3,929 batches (86%).

Table 4.5 also discriminates the frequency of batches applied on code

Table 4.5: Frequency by Batch Category

Batch Category	Total		Smelly	
	Abs.	%	Abs.	%
C01=[one commit, class, one type, ≥ 3]	1,142	25	730	16
C02=[one commit, class, one type, 2]	1,700	37	1,152	25
C03=[many commits, class, one type, ≥ 3]	54	1	43	1
C04=[many commits, class, one type, 2]	70	2	41	1
C05=[one commit, class, many types, ≥ 3]	639	14	465	10
C06=[one commit, class, many types, 2]	448	10	301	7
C07=[many commits, class, many types, ≥ 3]	71	2	49	1
C08=[many commits, class, many types, 2]	55	1	31	1
C09=[one commit, method, one type, ≥ 3]	54	1	12	0
C10=[one commit, method, one type, 2]	234	5	53	1
C11=[many commits, method, one type, ≥ 3]	29	1	2	0
C12=[many commits, method, one type, 2]	47	1	7	0
C13=[one commit, method, many types, ≥ 3]	7	0	4	0
C14=[one commit, method, many types, 2]	41	1	19	0
C15=[many commits, method, many types, ≥ 3]	6	0	0	0
C16=[many commits, method, many types, 2]	10	0	2	0
All categories	4,607	100	2,911	63

elements affected by code smells (e.g., smelly code elements). The fourth and fifth columns present absolute (Abs.) and relative (%) number of batches affecting smelly code elements by category. Thus, we have to first characterize to what extent batches are applied by developers on code elements, such as methods and classes, that have code smells before, during, or after the batch application. Surprisingly, we observed that only 63% of batches affect smelly code elements (according to the fifth column). About the remaining 37% of batches, we hypothesize that developers have other motivations than removing code smells when applying these batches.

What is new about this finding? Summary of RQ₃. We analyzed the most frequent code transformation types affecting smelly versus smell-free code elements in batches. We have found that: (i) 952 out of 1,696 (56%) of batches that affect smell-free code elements are also constituted by renames or code transformations types affecting attributes only, and (ii) 1,734 out of 2,911 (60%) of batches affecting smelly elements are constituted by code transformation types involving either extractions or motions of methods and/or classes. Thus, batches with certain non-trivial code transformation types (e.g., moves) are often applied to the smelly program elements.

4.2.4

Structural Effect of Batches on Code Smells (RQ₄)

Effect by Batch Characteristic. As discussed in Section 4.2.1, each batch characteristic may assume varied manifestations in practice. This is the case of the *scope* in which a batch is applied: batches can affect a *method* only or a *class*. In our first analysis, we explore the batch effect on code smells by batch characteristic, according to their respective manifestations. We are concerned

about identifying those manifestations that are more likely to either remove or introduce code smells. For this purpose, we have computed the Fisher's exact test (22) using a confidence interval equals 95% (p -value < 0.05). This test aims at computing the probability of a property (e.g., the manifestations of a given batch characteristic) to co-occur with another property (e.g., the introduction or removal of code smells).

Table 4.6 presents the Fisher's test results by batch characteristic. For each characteristic, we have created a contingency table in which: the lines correspond to the possible manifestations (e.g., Lines 2 and 3); and the columns are the absolute numbers of positive and negative effect of batches on code smells (e.g., Columns 3 and 5). We then computed the Fisher's test with such table as an input. The test has provided us with two values: Odds Ratio (OR) (17) and p -value. Odds Ratio informs the probability of manifestation (e.g., *one commit* in Line 2) to have a positive effect on code smells (Absolute in Column 3) when compared to the same probability for the opposite manifestation (*many commits* in Line 3). Achieving a statistically significant OR requires a p -value < 0.05 .

Table 4.6: Batch effect on code smells by batch characteristic

Characteristic	Manifestation	Positive		Negative		OR
		Absolute	%	Absolute	%	
<i>commit</i>	<i>one commit</i>	328	11%	1,414	49%	0.92
	<i>many commits</i>	21	1%	80	3%	
<i>scope</i>	<i>method</i>	30	1%	52	2%	*2.60
	<i>class</i>	319	11%	1,442	50%	
<i>variety</i>	<i>one type</i>	240	8%	1,051	36%	0.92
	<i>many types</i>	109	4%	443	15%	
<i>cardinality</i>	2	189	6%	847	29%	0.90
	≥ 3	160	5%	647	22%	

* p -value < 0.05

For the sake of illustration, let us consider the characteristic of *commit*. Batches applied in *one commit* are 92% more likely to have a positive effect on code smells than batches applied in *many commits*. In this case, the table indicates that statistical significance was not achieved by the *commit* characteristic. In fact, as the only batch characteristics whose OR is statistically significant is *scope*. In this particular case, we have found that batches applied on a *single method* are 260% more likely to have a positive effect on code smells when compared to batches applied at the *class* level (i.e., batches applied on either multiple methods or a class).

Effect by Batch Category. Aimed at complementing our analysis by characteristic, we have investigated the batch effects by batch category. As presented

in Section 4.1.2, a batch category is a combination of different manifestations for the batch characteristics following this format: $C = [commit, scope, variety, cardinality]$. An example of a batch category is $C01 = [one\ commit, class, one\ type, \geq 3]$. Our goal was understanding which batch categories are more likely to either negatively or positively affect code smells. Table 4.7 presents the effect of batches on code smells by category. The first column lists all 16 batch categories (C01 to C16). The following columns present the absolute (Abs.) and relative (%) frequency of batches whose effect is negative, neutral, and positive. We discuss our results as follows.

Table 4.7: Effect of batches by category

Batch Category	Negative		Neutral		Positive	
	Abs.	%	Abs.	%	Abs.	%
C01	366	13%	280	10%	84	3%
C02	611	21%	418	14%	123	4%
C03	17	1%	25	1%	1	0%
C04	23	1%	15	1%	3	0%
C05	230	8%	176	6%	59	2%
C06	156	5%	110	4%	35	1%
C07	25	1%	15	1%	9	0%
C08	14	0%	12	1%	5	0%
C09	5	0%	1	0%	6	0%
C10	28	1%	4	0%	21	1%
C11	0	0%	1	0%	1	0%
C12	1	0%	5	0%	1	0%
C13	4	0%	0	0%	0	0%
C14	14	0%	5	0%	0	0%
C15	0	0%	0	0%	0	0%
C16	0	0%	1	0%	1	0%
All categories	1,494	51%	1,068	38%	349	11%

A previous work (8) has found that most code transformations, when analyzed in isolation, tend to either introduce (33%) or not fully remove (57%) code smells from software projects. However, analyzing the effect of each single transformation on code smells may not suffice for understanding the whole effect of code refactoring on the program comprehension. In the particular case of batch refactoring, we have observed a similar scenario: most batches are likely to introduce (51%) and not fully remove (38%) code smells rather than removing them. This result is revealing in many ways. The batch categories C01, C02, C05, and C06 have 1,363 (47%) out of 2,911 batches that affect smelly code elements. These four categories represent batches that were applied in one commit and on the class scope. Besides, the categories C05 and C06 sums 386 (13%) out of 2,911 batches that affect smelly code, and these categories have 286 (10%) batches that are neutral. This result can indicate that even non-trivial batches, i.e., composed by more than one transformation type, do not remove code smells and they can introduce code smells. This

finding suggest that developers are often not combining transformation types to remove code smells. Specific compositions of transformation types in batches are actually introducing code smells.

In summary, the results reveal that 2,347 (81%) out of 2,911 batches affecting smelly code have either a negative or neutral effect. We observed that 58% of these batches are composed by *one transformation type* only, while 23% of such batches contain *many transformation types*. We also noticed that batches applied on *one commit*, which are applied on the *class scope*, are often either introducing or not removing code smells.

Effect by Batch Group. We observed that the analysis of the batch categories may not be sufficient to reveal certain details about the batch effect. As a complement to that analysis, we decided to analyze the effect of the nature of the transformation types on the introduction or removal of code smells.

We analyzed the batch effect based on the nature of code transformations that constitute a batch. We followed a three-step analysis procedure. **Step 1:** we grouped all 4,607 batches (Table 4.3) according to the natures of their code transformations (Table 4.2). It is worth mentioning that a batch may be composed of one or more natures; cases of multiple natures occurring together in a single batch were grouped in isolation from the “pure” groups (with only one nature). **Step 2:** we computed how many code smells were either introduced or removed by group of batches. **Step 3:** we computed the rate of code smells introduced by group according to the formula $I(g) = \frac{i(g)}{i(g)+r(g)}$, where: $i(g)$ is the number of smell instances introduced by a group g , and $r(g)$ is the number of smell instances removed by g .

Table 4.8 summarizes the batch effect on code smells according to the nature of transformations. The table data consider all 4,607 batches regardless the manifestations of batch characteristics. In the second and third columns, no batch was counted for more than one category. The remaining columns present the $I(g)$ values by code smell type analyzed in this work. We marked with “*” all $I(g)$ values for which $i(g)+r(g) < 5$; we aimed to warn about $I(g)$ values computed on only a few smell instances (which may not be as relevant as values computed on many instances). Red-colored cells indicate $I(g) > 50$, i.e., groups of batches that tend to introduce rather than remove code smells. Green-colored cells indicate $I(g) < 50$, i.e., groups that tend to remove rather than introduce smells. White-colored cells indicate either $I(g) = 50\%$, i.e., groups of batches that introduce and remove smells in an equivalent rate, or values marked with “*” and, therefore, considered of little practical relevance. We display the results for ten out of the 19 code smell types under analysis

(Table 2.2). This decision was taken because the occurrence of many smell types is very rare, which made it unfeasible to compute $I(g)$ values. After filtering the code smell types based on the criteria above, ten code smell types remained. Batches were more likely to remove code smells for two smell types only: Feature Envy and Message Chain.

Table 4.8: Batch Effect on Code Smells according to the Nature of Code Transformations

Nature	Total		Code Smell Type									
	Abs.	%	Class Data s/b Private	Complex Class	God Class	Intensive Coupling	Spaghetti Code	Speculative Generality	Feature Envy	Long Method	Long Param. List	Message Chain
G1	1,449	31	100	78.7	92.3	87.1	83.2	100	83.8	61.4	95.1	82.2
G2	1,142	25	98.5	85.5	89.3	*100	88.5	86.8	82.4	*100	*100	41.7
G3	580	13	100	100	91.7	100	80	100	98.7	94.1	100	97.8
G4	352	8	87.5	88.9	77.8	*100	57.1	95	47.8	*100	*33.3	28.6
G5	233	5	100	87.5	92.3	87.5	81	*100	84.8	88.2	81.3	80
G6	221	5	100	73.9	*66.6	88.9	75.0	*100	45.1	60	72.2	76.5
G7	153	3	100	78.4	88.9	*100	94.4	*100	98.1	73.7	100	37.5
G8	105	2	100	92.6	100	*100	100	*100	87.5	87.5	100	100
G9	84	2	*50	85.7	*50	*0	*50	100	*0	*0	*0	*0
G10	59	1	100	100	*100	*50	*50	*0	77.3	*66.6	*0	37.2
G11	58	1	*100	75.0	*0	*0	*0	100	*0	*0	*0	*0
G12	52	1	*100	75.0	*50	*100	*33.3	*0	35	66.7	40	*50
G13	28	1	100	100	*100	*100	*100	*0	100	100	*100	*100
G14	25	1	*100	100	100	*100	*100	*100	*100	*100	*100	*0
G15	18	0	*0	*0	*100	*0	*0	*0	50.0	*0	*100	*0
G16	11	0	*100	*66.6	*0	*0	*100	*100	80	*0	*0	*100
G17	11	0	*0	*0	*0	*0	*0	*66.6	*66.6	*0	*100	*0
G18	8	0	*0	*0	*0	*0	*0	*0	*0	*0	*0	*0
G19	7	0	*0	*100	*0	*0	*0	*0	*0	*0	*0	*0
G20	6	0	*100	*0	*0	*0	*0	*100	*100	*0	*0	*0
G21	3	0	*0	*0	*0	*0	*0	*0	*20	*0	*0	*100
G22	2	0	*0	*0	*0	*0	*0	*0	*0	*0	*0	*0
All	4,607	100	98.3	84.2	89.4	88.6	83.0	94.4	81.9	69.9	91.9	74.6

We discuss below our study findings by code smell type.

Regarding **Feature Envy**, batches of three natures (*Pull Up* (Group of G4), *Inline* (Group of G6), and *Inline/Motion*) (Group of G12) removed rather than introduced smells. *Pull Up* batches (all with at least one Pull Up Method) removed 12 Feature Envy instances; the “envious” methods were possibly moved to the superclass so that the smell affecting the subclass vanished. *Inline* batches with only Inline Method transformations removed 39 Feature Envy instances. Our results are quite surprising because previous catalogs (25) (7) recommend *Extraction/Motion* batches to fully remove Feature Envy. An example is combining Extract Method and Move Method (25), which introduced 26 Feature Envy instances (contrary to expectations). Curiously, *Extraction/Motion* batches had $I(g) = 98.1$ (Group of G7); thus, they rarely remove Feature Envy, at least as they have been applied so far. Two scenarios may justify this observation: (1) developers correctly applied Extract Method on the “envious” code, but they did not apply Move Method precisely on the

extracted (and "envious") methods; (2) many "envious" methods were created via Extract Method but not all of them were moved. Note that batches with only Extract Method introduced 841 Feature Envy instances (Group of G1); it reinforces how important is to combine Extract Method with other transformation type to fully remove smells.

Regarding **Message Chain**, batches of four natures (*Motion* (Group of G2), *Pull Up* (Group of G4), *Extraction/Motion* (Group of G7), and *Motion/Rename*) removed rather than introduced smells (Group of G10). Our results are quite revealing, once Fowler's recommendation to fully remove Message Chain instances is limited to *Extract/Method* batches (25). In the case reported by Fowler, the recommended batch can reduce a too long chain of method calls by moving methods that are closely related to the same class. As expected, batches of this nature were successful in fully removing Message Chain instances ($I(g) = 37.5$). As a complement, our results indicate alternative batches, such as those of *Motion/Renaming* nature. In this particular case, batches were able to remove 36 Message Chain instances; it is possible that methods within a chain of method calls was moved across classes, thereby reducing the chain size. Similar reasoning applies to *Pull Up* batches, which removed 5 Message Chain instances. This particular case is tricky: moving a method from subclass to superclass may have removed the smell instance from the subclass but introduced a smell instance in the superclass.

Long Method. A batch composed by several Extract Methods is recommended to remove a Long Method (25). However, we observed that 95 long methods can have removed by batches composed by this transformation type (batch group G1) possibly introduced this code smell type rather than removed it. This result suggests that developers may not have extracted enough parts of the long methods. Thus, the code transformations were insufficient to eliminate all Long Method instances. Similar scenario may apply to other batch groups that often introduce long methods: batches from groups G5, G7, G8, and G13 were usually composed of one or more Extract Method instances. We highlight that the code smell detection tool employed to collect our data (details in Section 4.1.2) captures as Long Method instances those methods with 30 or more lines of code. Thus, fully removing Long Methods with one hundred lines of code or more would require several code transformations.

4.3

Threats to Validity

We tried to carefully design and conduct the study. However, some threats to validity (70) may have affected our empirical study. We discuss

each threat to validity and their mitigation strategies hereafter.

Construct Validity. Identifying instances of code transformations applied by developers along with program refactoring is quite complex (61). This complexity is mostly due to the fact that developers usually apply these transformations together with other changes (46). Aimed at mitigating threats regarding misidentifying code transformations, we adopted the Refactoring Miner tool (60) (68). This tool was designed for identifying 14 types of code transformations with about 93% and 98% of recall and precision. The batch computation heuristic proposed by a previous work (7) and selected to support our work has several threats. This heuristics computes batches applied by the same developer on the same code element, which may have limited a lot our data set. Nevertheless, we assumed that, from all heuristics proposed by the previous work (7), this one is more likely to compute batches whose code transformations are interrelated; we assume that a same developer applying transformations on a specific code element has a common purpose with all the applied transformations. Finally, we have built a Python script for automating the batch identification through our heuristics (Section 4.1.2). We have double validated our script for detecting and fixing bugs.

Internal Validity. We carefully designed our data extraction protocol. We collected all code transformations and batches from the 57 projects under analysis in our study. The data was tabulated and validated. Thus, we aimed at mitigating threats regarding missing and duplicated data. For instance, our procedures have helped us to delete duplicated batch instances.

Conclusion Validity. To conduct the quantitative data analysis, we relied on similar studies (8) (10) and on guidelines provided by the literature (70) for conducting the descriptive analysis. All analysis results were peer reviewed to mitigate biases in the data analysis and misapplication of the analysis procedures. More importantly, counting the number of code smells before and after the batch application may not suffice capture the batch effect on code smells. In fact, we do not guarantee that the code transformations of a batch actually caused the code smell introduction or removal. We were not able to address this threat because we lacked fine-grained data about additional changes that co-occurred with a batch. Future work could address such limitation by separating code transformations from additional changes at the code level.

External Validity. In this work, we have investigated the program refactoring practices only on Java software projects. Because of that, our study results may be biased by the underlying code structure of Java-based programs. Although this threat remained not addressed, we highlight that Java is one of the most popular programming languages in both industry

and academia (Section 4.1.2). Additionally, although we have assessed both open and closed source projects, the number of closed source projects is quite low (only 3 projects) when compared to the number of open projects (the other 54 projects) and we did not conduct additional analyses applying the blocking principal to this characteristic yet. Hence, collecting data from more (in particular closed source) projects and conducting such additional analyses is part of future work. Finally, we have employed only one out of the three batch computation heuristics proposed by a recent PhD thesis (7). This heuristic was shaped to capture a very specific type of batches, i.e., batches composed of transformations applied by the same developer on the same code element. Consequently, our study results may be biased to this particular context and, therefore, they may not represent cases in which batches are composed by multiple developers and involve more than one class, for instance. Anyway, from the heuristics proposed by the previous work (7), the element-based heuristic explored in this dissertation seemed to be more reliable than the others in capturing actual batches. We recommend researchers to further explore the batch effect based on different heuristics in order to compare results and understand to what extent our findings are general to any batch application context.

4.4

Final Remarks

This chapter has introduced a large study about the batch characteristics and the structural effect of batches on code smells. We have mined 57 open and closed software projects in order to identify and analyze a total of 4,607 batches applied by developers. We relied on four batch characteristics elicited from our literature review (Chapter 3), such as the number of code transformations that constitute a batch. In this chapter, we presented a series of data analyses summarized as follows. The first analysis aimed at understanding the most typical batch characteristics in the selected projects. We aimed to address the following question: *How do batch characteristics often manifest in real projects?* We have found that most batches follow a particular trend: 93% occur in a single commit, 72% are constituted by a single code transformation type, and 22% have from four to ten code transformations. The second analysis investigated the natures of code transformations that more often compose batches. We found that 56% batches are composed by Extractions and/or Motions, and they revealed that larger batches (by means of the number of internal code transformation) tend to have transformations of varied natures.

The third analysis investigated batches by means of categories that

reflect their typical manifestations. We observed that batches affecting multiple methods in conjunction with the same class (and, eventually, the class structure itself) are very common. These batches, when applied on a single commit, correspond to 85% of our data set. We concluded that batches are usually complex in practice, once they affect the code structure in varied levels, from multiple methods to whole class. The fourth analysis aimed to understand the batch effect on code smells. Our results are quite promising: the program scope affected by a batch (i.e., the extent of code elements changed along with the batch refactoring) is strongly related with the removal of code smells. Batches that affect the same method are 260% more prone to remove a code smell than batches that affect many methods and/or the class. Moreover, we found that regardless of the batch categories, there is a common trend: batches have a negative effect on code smells rather than a positive or neutral one.

By scrutinizing our batch effect data, we revealed that quite interesting aspects of the code smell introduction and removal. Batches composed by transformations of the Pull Up, Inline, and Motion natures often may have removed Feature Envies. Batches composed by transformations of Motion, Extraction, and Pull Up natures may have removed Message Chains. Finally, batches composed by transformations of Extraction natures may have introduced Feature Envy or Long Method frequently. Based on these observations, we stated that simply recommending a series of Extract Methods, as made by Fowler (25), may not sufficient to fully remove a Long Method instance. As a future work, we plan to extend our analyses to cover additional code transformations and code smell types. Thus, we expect to can further evolve the understanding on the batch application towards removing code smells of varied types.

5 Conclusion

In this Master's dissertation, we have investigated a particular phenomenon that has been poorly explored by previous studies in spite of its frequency in practice: *batch refactoring*. Batch refactoring consists of applying sets of interrelated code transformations on the code structures of software project (7). These transformations are interrelated in such a way that they may help developers to fully reach a common goal, such as fully removing poor code structures that harm code maintenance (20, 64). Each set of interrelated code transformations is called a *batch* (7). An example of a batch consists of combining method extractions with method movements in order to better separate the software features across different methods (8, 25).

Unfortunately, the current knowledge about batch refactoring is quite scarce, which makes hard to support developers in their daily refactoring practices. Empirical studies tend to analyze each single transformation in isolation even if they are often performed in conjunction. The analysis of characteristics that constitute each single code transformation, e.g. the frequency of transformation types like Extract Method and Move Method (25), have been largely reported in the literature (8, 10, 4). However, little is known about the characteristics that constitute a batch in practice. Such a limited knowledge hinders researchers in properly studying batch refactoring. It also hinders developers in identifying and reasoning about batches applied on their software projects. Moreover, there is limited empirical evidence about the effect of batches on code maintenance (7). As an implication, developers may still feel reluctant in applying batches on their projects (35), due to the fear of worsening rather than enhancing code structures.

Aimed to address the literature gaps mentioned above and contribute with a more empirically-grounded knowledge about batch refactoring, this Master's dissertation compiles two complementary studies. The first study relies on previous work and summarizes the knowledge produced so far about (i) characteristics that constitute batch refactoring and (ii) expected types of batch effect on code maintenance. Thus, we were able to evaluate the possible conflicts on the assumptions made by researchers on which characteristics constitute batches and how batches affect software projects. The second study

relies on a large set of software projects and hundreds of heuristic-computed batches. We analyzed the frequency of certain characteristics in such batches. We also analyzed a particular type of batch effect: the introduction and removal of code smells, which usually represent poor code structures that are detrimental to code maintenance. We expect that some of the outcomes of this Master's dissertation can guide developers in their daily work while revealing opportunities for future research work on the topic. Particularly, we expect to provide insights for support the enhancement of current refactoring techniques and tools, such as (39, 64).

5.1

Summary of Study Contributions

Through a series of empirical studies presented in this Master's dissertation, we have provided some contributions. We summarize these contributions and their implications as follows.

A Conceptual Map of Batch Refactoring – Chapter 3 presented the outcomes of a literature review of batch refactoring. We rely on a set of 29 studies collected with the support of literature review and snowballing procedures (31, 36). From these studies, we extracted seven batch characteristics and seven types of batch effect on code maintenance. Many studies provide only assumptions about these characteristics and types of effect without any empirical evaluation. Not surprisingly, a consequence is that past studies often provide either fragmented or conflicting viewpoints about: (i) which characteristics actually constitute batches, and (ii) how batches affect the maintenance of real software projects. All the knowledge obtained from previous studies was summarized in a conceptual map of batch refactoring. We expect that this map, combined with our discussions on some literature conflicts, can better guide future research aimed to empirically evaluate both batch characteristics and types of effect.

An Empirical Exploration of Batch Characteristics in Real Projects – Part of Chapter 4 presents an empirical study on the most frequent manifestations of batch characteristics. Aimed to fill the literature gap about which characteristics more often apply to batches in practical settings, we mined and analyzed 57 open and closed software projects. We have collected batches through the element-based heuristic (7) in which this heuristic allows us to do a restrictive investigation on the manifestations of a batch that was applied on a code element. As a result, we observed that batches are quite complex in practice: although most batches (93%) require only one commit to be applied by developers, they are often composed of three or more code

transformations (28%) and affect much more than only one method (9%) on the same class.

These results both reinforce and complement the findings of a study recently reported in a PhD thesis on the topic (7). We were able to confirm the inherent batch complexity observed by that thesis. We were also able to elicit a more comprehensive set of batch characteristics, which were derived through our literature review (Chapter 3). In that thesis, the batch characteristics were arbitrarily chosen for investigation. The empirical understanding of frequent manifestations of each batch characteristic may be a useful instrument for researchers. This understanding can help them in refining heuristics for computing batches applied by developers in practical settings. For instance, we observed that it is common that code transformations in batches move a code element to another class. Thus, a heuristic can compute a batch through the code transformations applied on multiple classes.

A Large Study about the Batch Effect on Code Smells – Code smells have been shown useful for assisting developers on spotting poor code structures that harm code maintenance (17, 48, 53, 73). Therefore, characterizing the actual effect of batches on smell introduction and removal is essential to guide developers in enhancing code structures via code refactoring. In fact, code refactoring has been largely employed by the industry for removing poor code structures (35, 46, 60, 71). In this context, a previous study (7) has presented the very first study aimed to empirically assess the batch effect on code smell introduction and removal. Unfortunately, that study had some key limitations that hindered a comprehensive view of such effect on code maintenance, which implies little support to enhance current refactoring practices.

Based on a larger data set of 57 software projects than the previous work (7), plus a set of batch characteristics extracted from our literature review (Chapter 3), we extended the current knowledge and revealed additional aspects of the batch effect on code smells. Similarly to the previous work (7), we found that most batches end up either introducing (5%) or not fully removing (89%) code smells. Perhaps due to the extended data set, our study pointed out a 51% higher rate of batches introducing code smells. However, by scrutinizing the nature of code transformations that constitute these batches, we observed that code smells are often introduced by batches composed of Extract Methods and Move Methods. More critically, certain batches recommended by previous work (8, 25) were more likely to introduce rather than remove code smells. For these cases, our results revealed certain code transformations that can complement existing batches to fully remove code smells.

5.2

Insights to Enhance Current Refactoring Support

The empirical studies of this Master's dissertation have enabled us to reveal limitations on the current batch refactoring support. In other words, the understanding of batch characteristics, their most frequent manifestations, literature conflicts, and the batch effect on code smells have revealed the existing limitations and some opportunities for refining current tools and methods aimed to guide developers along with code refactoring. We discuss these limitations and some suggestions for improvement as follows.

Extending Refactoring Tools to Recommend Batches for Code Smell Removal – Previous studies (3, 66, 67) introduced tooling support and recommendation systems for guiding the application of isolated code transformations along code refactoring aiming the code smell removal. Unfortunately, these studies provide little or no support to the batch application. In fact, most of the current tools do not guide developers (i) to reason about the inter-relations of code transformations, and (ii) to understand how isolated transformations can be composed towards a code smell removal. Once batches are frequently applied by developers (7, 35, 46), the automated guidance of batch refactoring is desired.

In this Master's dissertation, we have observed that batches are usually quite complex in practice (Chapter 4). Nevertheless, developers seem not to make full use of varied code transformations while composing batches. We expect that our results about the manifestations of batch characteristics in real projects provide useful insights for researchers concerned about enhancing the current refactoring practices. Thus, existing tools can better support the composition and application of batches. Some examples are discussed below.

- **Improving the Refactoring Recommendation for Long Method Removal.** An existing support tool, called JDeodorant (66), recommends the Extract Method application aiming to remove a Long Method. However, our results present that mere application of a single Extract Method may lead to an extracted method that is also a Long Method. The original method, which was the target of the Extract Method, was too long and, even extracting part of it, the refactoring was not able to fully solve the smell. In other words, the smell is being propagated to the other method produced along the refactoring. The JDeodorant tool could be extended to recommend a batch with the number of Extract Methods to fully resolve the Long Method smell.
- **Improving the Refactoring Recommendation for Feature Envy Removal.** The existing catalogs recommend batches composed by Ex-

tract Methods and Move Methods to remove a Feature Envy (25). In our results, we have found that batches composed by these transformations have been introduced Feature Envies. This is because developers can be applying more Extract Methods than Move Methods or they are moving other methods that are not envious. Thus, the extracted methods are envious code and they are not moved to another class. In that context, an automated support tool can recommend the batch application to remove this code smell, but this tool also can alert the developer what methods need be moved.

Batches are Relevant for the Software Architecture Improvement. In our results, we have found that more than 42% batches are composed by transformations that move code elements across classes. These classes may play a key role in the architecture of a system. Moreover, the classes affected by a batch are possibly located in different packages (which often represent components in the software architecture). These observations indicate that batches may affect (positively or negatively) the architecture of a software. Thus, developers should be provided with tools that recommend batches for software architecture improvement. Lin et al. (39) proposed the Refactoring Navigator, a tool-supported approach that allows the developer to indicate the new desired architectural design. The system needs to be re-structured to achieve the new desired architectural design. The goal of the Refactoring Navigator approach is to support this transition. To do so, the tool calculates one or more batches that must be applied to achieve the intended architecture design, while also improving the cohesion and coupling attributes of the program.

Future Work. We plan to work on providing developers with tool support for performing their batches. In other words, we plan to address the above insights in our next steps. We envisage to build a new tool that support developers in applying our batch recommendations in a way that is not detrimental to the quality of the software architecture. We plan to build an interactive tool where developers are providing feedback for our recommendations so that the tool takes it into consideration when suggesting the new batches. We also plan to replicate our study considering: (i) the other two batch characteristics (i.e., *developer* and *time*) not addressed in our study of Chapter 4, and (ii) the use of other batch detection heuristics (e.g., those defined in (7)), in particular heuristics that consider other forms of scope (e.g., packages, versions and the like).

5.3

Research Publications

This Master's dissertation is composed of various empirical studies which are also being reported in papers. We plan to submit these papers to international conferences and journals for publication in the near future. Table 5.1 lists the publications planned for each study presented throughout the dissertation chapters. The table also lists those papers that are not derived from the core research of this dissertation. We have either published them or written drafts along the course of this Master's dissertation.

Table 5.1: List of Research Publications

Type	Paper Metadata	Chapter
Master's dissertation	To be Defined. (2019). A Literature Review of Batch Refactoring. In: 13th ESEM, pp. 1–12. (To submit)	3
	Bibiano, A.C. , Fernandes, E., Oliveira, D., Garcia, A., Kalinowski, M., Fonseca, B., Oliveira, R., Oliveira, A., Cedrim, D. (2019). How Do Batch Refactorings Affect Code Smells? A Large Study of 57 Software Projects. In: 13th ESEM, pp. 1–12. (To submit)	4
	Uchôa, A., Fernandes, E., Bibiano, A.C. , Garcia, A. (2017). Do coupling metrics help characterize critical components in component-based SPL? An empirical study. In: 5th VEM, co-located with the 8th CBSOft, pp. 46–53	N/A
Other studies	Ferreira, I., Fernandes, E., Cedrim, D., Uchôa, A., Bibiano, A.C. et al. (2018). The buggy side of code refactoring: Understanding the relationship between refactorings and bugs. In: 40th ICSE: Poster Track, pp. 406–407	N/A
	Fernandes, E., Uchôa, A., Bibiano, A.C. , Garcia, A. (2019). On the Alternatives for Composing Batch Refactoring. In: 3rd IWoR, co-located with 41st ICSE, pp. 1–4	N/A
	Oliveira, D., Bibiano, A.C. , Garcia, A. (2019). On the Customization of Batch Refactoring. In: 3rd IWoR, co-located with 41st ICSE, pp. 1–4	N/A
	To be Defined. (2019). About the Developers' Intentions behind Batch Refactoring. In: 35th ICSME: Short Paper Track, pp. 1–5 (To submit)	N/A

Bibliography

- [1] BASILI, V.; ROMBACH, H.. **The TAME project: Towards improvement-oriented software environments.** IEEE Transactions on Software Engineering (TSE), 14(6):758–773, 1988.
- [2] BAVOTA, G.; DE CARLUCCIO, B.; DE LUCIA, A.; DI PENTA, M.; OLIVETO, R. ; STROLLO, O.. **When does a refactoring induce bugs? An empirical study.** In: PROCEEDINGS OF THE 12TH WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 104–113, 2012.
- [3] BAVOTA, G.; LUCIA, A. D. ; OLIVETO, R.. **Identifying extract class refactoring opportunities using structural and semantic cohesion measures.** Journal of Systems and Software (JSS), 84(3):397–414, 2011.
- [4] BAVOTA, G.; LUCIA, A. D.; PENTA, M. D.; OLIVETO, R. ; PALOMBA, F.. **An experimental investigation on the innate relationship between quality and refactoring.** Journal of Systems and Software (JSS), 107:1–14, 2015.
- [5] BEN FADHEL, A.; KESSENTINI, M.; LANGER, P. ; WIMMER, M.. **Search-based detection of high-level model changes.** In: PROCEEDINGS OF THE 28TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 212–221, 2012.
- [6] BORGES, H.; VALENTE, M. T.. **What’s in a GitHub star? Understanding repository starring practices in a social coding platform.** Journal of Systems and Software (JSS), 146:112–129, 2018.
- [7] CEDRIM, D.. **Understanding and Improving Batch Refactoring in Software Systems.** PhD thesis, Informatics Department (DI), Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil, 2018.
- [8] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects.** In: PROCEEDINGS OF THE 11TH JOINT

- MEETING OF THE EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE (ESEC/FSE), p. 465–475, 2017.
- [9] CHAPARRO, O.; BAVOTA, G.; MARCUS, A. ; DI PENTA, M.. **On the impact of refactoring operations on code quality metrics.** In: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 456–460, 2014.
- [10] CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How does refactoring affect internal quality attributes? A multi-project study.** In: PROCEEDINGS OF THE 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 74–83, 2017.
- [11] Ó CINNÉIDE, M.; NIXON, P.. **A methodology for the automated introduction of design patterns.** In: PROCEEDINGS OF THE 7TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 463–472, 1999.
- [12] Ó CINNÉIDE, M.; NIXON, P.. **Composite refactorings for java programs.** In: PROCEEDINGS OF THE WORKSHOP ON FORMAL TECHNIQUES FOR JAVA PROGRAMS, CO-LOCATED WITH THE 14TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP), p. 1–6, 2000.
- [13] CRESWELL, J.. **Research Design: Qualitative, Quantitative, and Mixed Methods Approaches.** SAGE Publications, 4th edition, 2014.
- [14] DE OLIVEIRA, M. C.. **DRACO: Discovering refactorings that improve architecture using fine-grained co-change dependencies.** In: PROCEEDINGS OF THE 11TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 1018–1021, 2017.
- [15] FATIREGUN, D.; HARMAN, M. ; HIERONS, R.. **Evolving transformation sequences using genetic algorithms.** In: PROCEEDINGS OF THE 4TH INTERNATIONAL WORKSHOP ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 65–74, 2004.
- [16] FERNANDES, E.; OLIVEIRA, J.; VALE, G.; PAIVA, T. ; FIGUEIREDO, E.. **A review-based comparative study of bad smell detection tools.** In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING (EASE), p. 18:1–18:12, 2016.

- [17] FERNANDES, E.; VALE, G.; SOUSA, L.; FIGUEIREDO, E.; GARCIA, A. ; LEE, J.. **No code anomaly is an island: Anomaly agglomeration as sign of product line instabilities.** In: PROCEEDINGS OF THE 16TH INTERNATIONAL CONFERENCE ON SOFTWARE REUSE (ICSR), p. 48–64, 2017.
- [18] FERNANDES, E.; SOUZA, P.; FERREIRA, K.; BIGONHA, M. ; FIGUEIREDO, E.. **Detection strategies for modularity anomalies: An evaluation with software product lines.** In: PROCEEDINGS OF THE 14TH INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY: NEW GENERATIONS (ITNG), p. 565–570. 2018.
- [19] FERNANDES, E.. **Stuck in the middle: Removing obstacles to new program features through batch refactoring.** In: PROCEEDINGS OF THE 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE): DOCTORAL SYMPOSIUM (DS), p. 1–4, 2019.
- [20] FERNANDES, E.; UCHOA, A.; BIBIANO, A. C. ; GARCIA, A.. **On the alternatives for composing batch refactoring.** In: PROCEEDINGS OF THE 3RD INTERNATIONAL WORKSHOP ON REFACTORING (IWOR), CO-LOCATED WITH THE 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 1–4, 2019.
- [21] FERREIRA, I.; FERNANDES, E.; CEDRIM, D.; UCHÔA, A.; BIBIANO, A. C.; GARCIA, A.; CORREIA, J. L.; SANTOS, F.; NUNES, G.; BARBOSA, C. ; OTHERS. **The buggy side of code refactoring: Understanding the relationship between refactorings and bugs.** In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE): POSTER TRACK, p. 406–407, 2018.
- [22] FISHER, R.. **On the interpretation of χ^2 from contingency tables, and the calculation of p.** Journal of the Royal Statistical Society, 85(1):87–94, 1922.
- [23] FOKAEFS, M.; TSANTALIS, N. ; CHATZIGEORGIOU, A.. **JDeodorant: Identification and removal of feature envy bad smells.** In: PROCEEDINGS OF THE 23RD INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 519–520, 2007.
- [24] FOSTER, S.; GRISWOLD, W. ; LERNER, S.. **WitchDoctor: IDE support for real-time auto-completion of refactorings.** In: PROCEEDINGS OF THE 34TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 222–232, 2012.

- [25] FOWLER, M.. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley Professional, 1st edition, 1999.
- [26] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design patterns: Abstraction and reuse of object-oriented design**. In: PROCEEDINGS OF THE 7TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP), p. 406–431, 1993.
- [27] GOUTTE, C.; GAUSSIER, E.. **A probabilistic interpretation of precision, recall and F-score, with implication for evaluation**. In: PROCEEDINGS OF THE 27TH EUROPEAN CONFERENCE ON INFORMATION RETRIEVAL (ECIR), p. 345–359, 2005.
- [28] GRIFFITH, I.; WAHL, S. ; IZURIETA, C.. **Truerefactor: An automated refactoring tool to improve legacy system and application comprehensibility**. In: PROCEEDINGS OF THE 24TH INTERNATIONAL CONFERENCE ON COMPUTER APPLICATIONS IN INDUSTRY AND ENGINEERING (CAINE), p. 1–6, 2011.
- [29] GUERRA, E.; FERNANDES, C.. **Refactoring test code safely**. In: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING ADVANCES (ICSEA), p. 44–44, 2007.
- [30] HARMAN, M.; TRATT, L.. **Pareto optimal search based refactoring at the design level**. In: PROCEEDINGS OF THE 9TH GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO), p. 1106–1113, 2007.
- [31] JALALI, S.; WOHLIN, C.. **Systematic literature studies: Database searches vs. backward snowballing**. In: PROCEEDINGS OF THE 6TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 29–38, 2012.
- [32] KANG, K.; COHEN, S.; HESS, J.; NOVAK, W. ; PETERSON, A.. **Feature-oriented domain analysis (FODA) feasibility study**. Technical report, CMU-SEI-90-TR-21 and ESD-90-TR-222, Software Engineering Institute (SEI), Carnegie Mellon University (CMU), 1990.
- [33] KIM, J.; BATORY, D. ; DIG, D.. **Scripting parametric refactorings in Java to retrofit design patterns**. In: PROCEEDINGS OF THE 31ST INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 211–220, 2015.

- [34] KIM, J.; BATORY, D.; DIG, D. ; AZANZA, M.. **Improving refactoring speed by 10x.** In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 1145–1156, 2016.
- [35] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **An empirical study of refactoring: Challenges and benefits at Microsoft.** IEEE Transactions on Software Engineering (TSE), 40(7):633–649, 2014.
- [36] KITCHENHAM, B.; CHARTERS, S.. **Guidelines for performing systematic literature reviews in software engineering.** Technical report, EBSE 2007-001, Version 2.3, Keele University and University of Durham, 2007.
- [37] KUHLEMANN, M.; LIANG, L. ; SAAKE, G.. **Algebraic and cost-based optimization of refactoring sequences.** In: PROCEEDINGS OF THE 2ND INTERNATIONAL WORKSHOP ON MODEL-DRIVEN PRODUCT LINE ENGINEERING (MDPLE), CO-LOCATED WITH THE 6TH EUROPEAN CONFERENCE ON MODELLING FOUNDATIONS AND APPLICATIONS (ECMFA), p. 37–48, 2010.
- [38] LANZA, M.; MARINESCU, R.. **Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems.** Springer Science & Business Media, 1st edition, 2006.
- [39] LIN, Y.; PENG, X.; CAI, Y.; DIG, D.; ZHENG, D. ; ZHAO, W.. **Interactive and guided architectural refactoring with search-based recommendation.** In: PROCEEDINGS OF THE 24TH INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 535–546, 2016.
- [40] MACIA, I.; ARCOVERDE, R.; GARCIA, A.; CHAVEZ, C. ; VON STAA, A.. **On the relevance of code anomalies for identifying architecture degradation symptoms.** In: PROCEEDINGS OF THE 16TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), p. 277–286, 2012.
- [41] MAHOUACHI, R.; KESSENTINI, M. ; Ó CINNÉIDE, M.. **Search-based refactoring detection.** In: PROCEEDINGS OF THE 15TH GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO), p. 205–206, 2013.

- [42] MCCABE, T.. **A complexity measure**. IEEE Transactions on Software Engineering (TSE), SE-2(4):308–320, 1976.
- [43] MEANANEATRA, P.. **Identifying refactoring sequences for improving software maintainability**. In: PROCEEDINGS OF THE 27TH INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 406–409, 2012.
- [44] MKAOUER, M. W.; KESSENTINI, M.; BECHIKH, S.; DEB, K. ; Ó CINNÉIDE, M.. **Recommendation system for software refactoring using innovization and interactive dynamic optimization**. In: PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 331–336, 2014.
- [45] MONTEIRO, M.; FERNANDES, J.. **Towards a catalog of aspect-oriented refactorings**. In: PROCEEDINGS OF THE 4TH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD), p. 111–122, 2005.
- [46] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A.. **How we refactor, and how we know it**. IEEE Transactions on Software Engineering (TSE), 38(1):5–18, 2012.
- [47] Ó CINNÉIDE, M.; TRATT, L.; HARMAN, M.; COUNSELL, S. ; HEMATI MOGHADAM, I.. **Experimental assessment of software metrics using automated refactoring**. In: PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 49–58, 2012.
- [48] OIZUMI, W.; GARCIA, A.; SOUSA, L.; CAFEO, B. ; ZHAO, Y.. **Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems**. In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 440–451, 2016.
- [49] OUNI, A.; KESSENTINI, M.; SAHRAOUI, H. ; HAMDI, M. S.. **Search-based refactoring: Towards semantics preservation**. In: PROCEEDINGS OF THE 28TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 347–356, 2012.
- [50] OUNI, A.; KESSENTINI, M. ; SAHRAOUI, H.. **Search-based refactoring using recorded code changes**. In: PROCEEDINGS OF THE 17TH EU-

- ROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), p. 221–230, 2013.
- [51] OUNI, A.; KESSENTINI, M.; SAHRAOUI, H. ; HAMDİ, M. S.. **The use of development history in software refactoring using a multi-objective evolutionary algorithm.** In: PROCEEDINGS OF THE 15TH GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO), p. 1461–1468, 2013.
- [52] PAIXAO, M.; KRINKE, J.; HAN, D.; RAGKHITWETSAGUL, C. ; HARMAN, M.. **Are developers aware of the architectural impact of their changes?** In: PROCEEDINGS OF THE 32ND INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 95–105, 2017.
- [53] PALOMBA, F.; BAVOTA, G.; DI PENTA, M.; OLIVETO, R. ; DE LUCIA, A.. **Do they really smell bad? A study on developers' perception of bad code smells.** In: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (IC-SME), p. 101–110, 2014.
- [54] PETERS, R.; ZAIDMAN, A.. **Evaluating the lifespan of code smells using software repository mining.** In: PROCEEDINGS OF THE 16TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), p. 411–416, 2012.
- [55] PIVETA, E.; ARAÚJO, J.; PIMENTA, M.; MOREIRA, A.; GUERREIRO, P. ; PRICE, R. T.. **Searching for opportunities of refactoring sequences: Reducing the search space.** In: PROCEEDINGS OF THE 32ND INTERNATIONAL CONFERENCE ON COMPUTER SOFTWARE AND APPLICATIONS (COMPSAC), p. 319–326, 2008.
- [56] PRETE, K.; RACHATASUMRIT, N.; SUDAN, N. ; KIM, M.. **Template-based reconstruction of complex refactorings.** In: PROCEEDINGS OF THE 26TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 1–10, 2010.
- [57] QAYUM, F.; HECKEL, R.; CORRADINI, A.; MARGARIA, T.; PADBERG, J. ; TAENTZER, G.. **Search-based refactoring based on unfolding of graph transformation systems.** In: PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION (ICGT): DOCTORAL SYMPOSIUM (DS), p. 1–14, 2010.

- [58] RAYCHEV, V.; SCHÄFER, M.; SRIDHARAN, M. ; VECHEV, M.. **Refactoring with synthesis**. ACM SIGPLAN Notices, 48(10):339–354, 2013.
- [59] SADOWSKI, C.; SÖDERBERG, E.; CHURCH, L.; SIPKO, M. ; BACCHELLI, A.. **Modern code review: A case study at Google**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE): SOFTWARE ENGINEERING IN PRACTICE TRACK (SEIP), p. 181–190, 2018.
- [60] SILVA, D.; TSANTALIS, N. ; VALENTE, M. T.. **Why we refactor? Confessions of GitHub contributors**. In: PROCEEDINGS OF THE 24TH INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 858–870, 2016.
- [61] SILVA, D.; VALENTE, M. T.. **RefDiff: Detecting refactorings in version histories**. In: PROCEEDINGS OF THE 14TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 269–279, 2017.
- [62] SOUSA, L.; OLIVEIRA, A.; OIZUMI, W.; BARBOSA, S.; GARCIA, A.; LEE, J.; KALINOWSKI, M.; DE MELLO, R.; FONSECA, B.; OLIVEIRA, R.; LUCENA, C. ; PAES, R.. **Identifying design problems in the source code: A grounded theory**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 921–931, 2018.
- [63] STOL, K.-J.; RALPH, P. ; FITZGERALD, B.. **Grounded theory in software engineering research: A critical review and guidelines**. In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 120–131, 2016.
- [64] SZŐKE, G.; NAGY, C.; FÜLÖP, L.; FERENC, R. ; GYIMÓTHY, T.. **Fault-Buster: An automatic code smell refactoring toolset**. In: PROCEEDINGS OF THE 15TH WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 253–258, 2015.
- [65] SZOKE, G.; NAGY, C.; FERENC, R. ; GYIMÓTHY, T.. **Designing and developing automated refactoring transformations: An experience report**. In: PROCEEDINGS OF THE 23RD INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 693–697, 2016.

- [66] TSANTALIS, N.; CHATZIGEORGIOU, A.. **Identification of move method refactoring opportunities**. *IEEE Transactions on Software Engineering (TSE)*, 35(3):347–367, 2009.
- [67] TSANTALIS, N.; CHATZIGEORGIOU, A.. **Identification of extract method refactoring opportunities for the decomposition of methods**. *Journal of Systems and Software (JSS)*, 84(10):1757–1782, 2011.
- [68] TSANTALIS, N.; GUANA, V.; STROULIA, E. ; HINDLE, A.. **A multidimensional empirical study on refactoring activity**. In: *PROCEEDINGS OF THE 23RD INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND SOFTWARE ENGINEERING (CASCON)*, p. 132–146, 2013.
- [69] VILLAVICENCIO, G.. **A new software maintenance scenario based on refactoring techniques**. In: *PROCEEDINGS OF THE 16TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR)*, p. 341–346, 2012.
- [70] WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.; REGNELL, B. ; WESSLÉN, A.. **Experimentation in Software Engineering**. Springer Science & Business Media, 1st edition, 2012.
- [71] WRIGHT, H.; JASPER, D.; KLIMEK, M.; CARRUTH, C. ; WAN, Z.. **Large-scale automated refactoring using ClangMR**. In: *PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM)*, p. 548–551, 2013.
- [72] YAMASHITA, A.; MOONEN, L.. **Do code smells reflect important maintainability aspects?** In: *PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM)*, p. 306–315, 2012.
- [73] YAMASHITA, A.; MOONEN, L.. **Do developers care about code smells? An exploratory survey**. In: *PROCEEDINGS OF THE 20TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE)*, p. 242–251, 2013.
- [74] YAMASHITA, A.; MOONEN, L.. **Exploring the impact of inter-smell relations on software maintainability: An empirical study**. In: *PROCEEDINGS OF THE 35TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE)*, p. 682–691, 2013.

- [75] YAU, S.; COLLOFELLO, J.. **Some stability measures for software maintenance.** IEEE Transactions on Software Engineering (TSE), SE-6(6):545–552, 1980.