On the Characterization, Detection and Impact of Batch Refactoring in Practice

Ana Carla Bibiano[1], **Alessandro Garcia (Advisor)**[1]

[1]Informatics Department – Pontifical Catholic University of Rio de Janeiro (PUC-Rio)

{abibiano,afgarcia}@inf.puc-rio.br

Abstract. *Up to 60% of the refactorings in software projects are constituted of a set of interrelated transformations, the so-called batches (or composite refactoring), rather than single transformations applied in isolation. However, a systematic characterization of batch characterization is missing, which hampers the elaboration of proper tooling support and empirical studies of how (batch) refactoring is applied in practice. This paper summarizes the research performed under the context of a Master's dissertation, which aimed at taming the aforementioned problems. To the best of our knowledge, our research is the first work published that provides a conceptual foundation, detection support and an large impact analysis of batch refactoring on code maintainability. To this end, we performed two complementary empirical studies as well as designed a first heuristic aimed at explicitly detecting batch refactorings. Our first study consisted of a literature review that synthesizes the otherwise scattered, partial conceptualization of batch refactoring mentioned in 29 studies with different purposes. We identified and defined seven batch characteristics such as the scope and typology of batches, plus seven types of batch effect on software maintainability, including code smell removal. All batches' characteristics and possible impacts were systematized in a conceptual framework, which assists, for instance, the proper design of batch refactoring studies and batch detection heuristics. We defined a new heuristic for batch detection, which made it possible to conduct a large study involving 4,607 batches discovered in 57 open and closed software projects. Amongst various findings, we reveal that most batches in practice occur entirely within one commit (93%), affect multiple methods (90%). Surprisingly, batches mostly end up introducing (51%) or not removing (38%) code smells. These findings contradict previous investigations limited to the impact analysis of each transformation in isolation. Our findings also enabled us to reveal beneficial or harmful patterns of batches that respectively induces the introduction or removal of certain code smells. These patterns: (i) were not previously documented even in Fowler's refactoring catalog, and (ii) provide concrete guidance for both researchers, tool designers, and practitioners.*

Keywords: *batch refactoring, code smell, literature review, internal structural quality*

Resumo. *Até 60% das transformações em projetos de software são constituídas de um conjunto de transformações inter-relacionadas, os chamados lotes (ou refatorações compostas), ao invés de aplicadas isoladamente. No entanto, a caracterização sistemática sobre lotes é faltante, isso dificulta a elaboração*

de ferramentas apropriadas que suportem refatoração e estudos empíricos sobre como refatoração em lote é aplicado na prática. Esse artigo sumariza a pesquisa elaborada em torno do contexto de uma dissertação de mestrado que tem foco sobre esses problemas. Para o nosso melhor conhecimento, nossa pesquisa é o primeiro trabalho publicado que fornece uma fundação conceitual, suporte de detecção e análises de grande impacto de refatoração em lote sobre a manutenibilidade do código. Para este fim, nós elaboramos dois estudos experimentais complementares bem como projetada a primeira heurística que almeja explicitamente suportar a detecção e análise de refatoração em lotes. Primeiro, nós elaboramos uma revisão da literatura que sintetiza o, caso contrário, espalhado, parcial conceitualização de refatoração em lote mencionada em 29 estudos. Nós identificamos sete características de lotes tais como o escopo no qual lotes são aplicados em estruturas de código, mais sete tipos de efeito de lotes sobre a manutenção de sistemas, tais como a remoção de anomalias. As características e seus possíveis efeitos identificados foram sistematizados em um framework conceitual. A dissertação encerra-se com uma análise de 4.607 lotes detectados com nossa heuristica em 57 projetos de sistemas abertos e fechados. Entre várias descobertas, nós revelamentos que a maioria dos lotes ocorre inteiramente dentro de um commit (93%) mas afeta múltiplos métodos (90%). Surpreendentemente, a maioria dos lotes principalmente acabam introduzindo (51%) ou não removendo (38%) anomalias. Esses resultados contradizem descobertas anteriores of estudos limitados a análise do impacto de cada transformação isolada. Nossas descobertas também permite-nos revelar formas benéficas ou prejudiciais de lotes que frequentemente induzem a introdução ou remoção de certas anomalias de código, que (i) não foram documentadas por estudos anteriores ou no catálogo de refactoring do Fowler, e (ii) fornece um guia concreto para ambos pesquisadores, designers e profissionais de ferramentas.

Palavras-chave: *refatoração em lote, anomalias de código, revisão da literatura, qualidade interna estrutural*

1. Introduction

Code refactoring consists of applying one or more transformations on the code structure of a software project [Fowler 1999]. Refactoring has been largely employed by developers, in major companies such as Microsoft [Kim et al. 2014], to remove poor code structures that represent threats to code maintenance. However, regardless the developer motivation or intent [Paixão et al. 2020] behind refactoring, the code transformations applied along the change should enhance or at least preserve the code structure quality [Kim et al. 2014].

Applying code refactoring in practical settings is quite complex [Kim et al. 2014]. There is a plenty of code transformation types that developers can use and compose to achieve their code structure enhancement goals [Fowler 1999]. Each type defines how a developer should modify the code elements, such as attributes, methods, and classes [Fowler 1999, Meananeatra 2012] in order to improve the code structure. Examples of common refactoring types are Extract Method and Move Method [Fowler 1999]. Recurring poor code structures are usually represented by the

so-called code smells [Fowler 1999]. Examples of code smell types frequently recognized as harmful by practitioners and cataloged by Fowler include God Classes, Feature Envies and Long Methods [Yamashita and Moonen 2012].

A single transformation rarely suffices to achieve the full removal of a code smell [Cedrim et al. 2017]. That is the case for most code smells, for which two or more code transformations have to be combined to fully remove them [Cedrim et al. 2017]. For instance, shortening a long method often requires many method extractions. Recent studies [Murphy-Hill et al. 2012, Bibiano 2019] has shown that about 40-60% of code transformations are applied in *batches* (or batch refactorings), i.e., sets of two or more interrelated transformations, rather than in isolation.

Although batches are frequent in practice, the knowledge of batch characteristics and their impact on code quality is quite fragmented across previous research. Even worse, the vast majority of the previous studies focus on the detection and impact analysis of single transformations, ignoring the prevalence of batch refactoring in practice. As a consequence, empirical knowledge about refactoring is possibly misleading, thereby misinforming further research as well as practitioners and tool designers.

Based on those limitations, this work presented a literature review and a quantitative study aiming to summarize the knowledge about batches according to the existing literature and to investigate how developers apply batches in practice. We have found a total of 29 previous studies published in 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. Based on the knowledge acquired from our literature review, we followed the feature model principles [Kang et al. 1990] to build a conceptual map of batch refactoring.

Our conceptual map can guide future research towards to reveal how batches are applied considering different characteristics and effect types. At motivating the potential of our conceptual map, we created an element-based heuristic to detect batches from characteristics revealed in our conceptual map. This heuristic can be used by future automated support tools to learn good practices of batch applications. In our quantitative study, we investigate 4,607 from 57 projects. This study revealed that 72% batches are constituted of the same transformation type suggested by Fowler's refactoring book [Fowler 1999]. Besides that, 51% batches tend to introduce code smells, indicating that batches are poorly exploited by developers for code smell removal.

The remainder of this paper is organized as follows. Section 2 provides background information on main concepts about complete composites. The study design and results of our literature review are presented in Section 3. Our quantitative study is presented in Section 4. A summary of the implications of our results and conclusions are presented in Section 5 and Section 6, respectively.

2. 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 [Bibiano et al. 2019, Murphy-Hill et al. 2012]. Unfortunately, the current knowledge of batch refactoring is

considerably fragmented across previous studies. Even worse, the vast majority of the previous studies focus on the detection and impact analysis of single transformations, ignoring the prevalence of batch refactoring in practice.

As far as the characteristics that constitute a batch are concerned, there are major limitations in the literature. In fact, past studies assume one or another batch characteristic, such as the variety of transformation types within a batch [Meananeatra 2012]. 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 [Murphy-Hill et al. 2012], contrarily to others that assume batches as composed of varied transformation types [Kim et al. 2014].

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 [Cedrim 2018, Meananeatra 2012]. Conversely, certain studies discuss that batches can sometimes be detrimental to the code structure quality [Meananeatra 2012], thereby introducing poor code structures like code smells [Cedrim 2018] and hindering maintenance tasks. Most of these studies lack empirical evidence, which ultimately 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 [Kim et al. 2014].

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 framework of batch characteristics and types of effect on code maintenance. We hypothesize that a comprehensive conceptual framework 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.

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 [Cedrim 2018, Fowler 1999, Meananeatra 2012], i.e., batches can affect multiple methods, a whole class. Some batches can also be composed of the same transformation type, e.g., Extract Method only [Murphy-Hill et al. 2012], while others can combine multiple types like Extract Method and Move Method [Cedrim 2018]. Unfortunately, none of the previous studies have systematically investigated what are the most frequent

manifestations of batch characteristics, especially through empirical studies.

A clear understanding of how batch characteristics usually manifest in software projects can be beneficial to future research in many ways. By characterizing the usual structure 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 to 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 [Cedrim et al. 2017, Chávez et al. 2017] has ultimately focused on assessing the effect of each single code transformation applied along refactoring rather than batches. Some studies like [Ferreira et al. 2018] 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, bugs do not occur that far from the transformations in the commit history of software projects [Ferreira et al. 2018]. Other studies show that not always the single code transformations are beneficial to code structures [Cedrim et al. 2017, Chávez et al. 2017]. In fact, single transformation tend to either introduce 3% or not fully remove 95% code smells [Cedrim et al. 2017]. 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.

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

3. A Literature Review of Batch Refactoring

The first study that composes this work [Bibiano 2019] 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 [Kitchenham and Charters 2007]. Our study goal was three-fold: (i) 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; (ii) 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; and (iii) 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.

Research Questions. Table 1 presents the research questions (RQs) that we designed for guiding our study. 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 1. Research Questions of the Literature Review

ID	Description
$\mathbf{RQ}_1$	Which batch characteristics have been reported by previous work?
$\mathbf{RQ}_2$	Which batch effect on software projects are assumed by previous work?
$\mathbf{RQ}_3$	Do previous studies assume conflicting batch characteristics and effects?

Figure 1 presents the ten study steps of our literature review. **Step 1: Run a pilot search.** We have performed a pilot search as a preparation to the literature review. We retrieved the top-20 most relevant papers in order to assess the most common key-terms used by these studies. **Step 2: Define a final search string.** Based on the pilot search, we decided our final search string. **Steps 3 to 7: Select papers for analysis.** We then selected the papers according to guidelines [Kitchenham and Charters 2007]. **Step 8: Snowballing.** We have followed an existing guideline [Jalali and Wohlin 2012] to perform snowballing procedures. **Step 9: Extract data.** We relied on a previous work [Kitchenham and Charters 2007] to define which data could help us in addressing our RQs. **Step 10: Build conceptual map.** We have applied some basic Grounded Theory procedures [Stol et al. 2016] on the extracted data

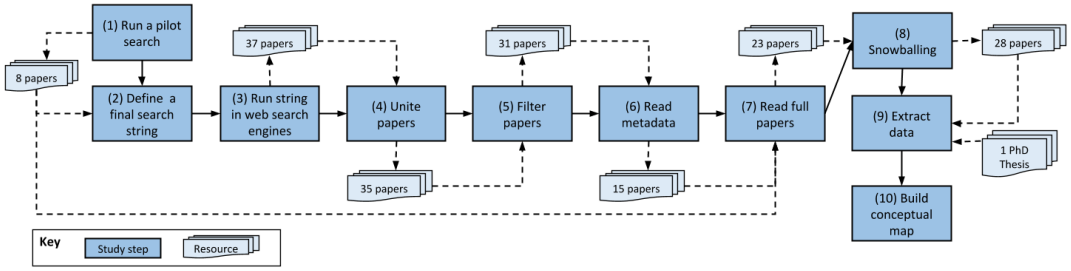

Figure 1. Steps of the Literature Review

Figure 2 presents our conceptual framework of a batch refactoring based on 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 (detailed in Master's Dissertation [Bibiano 2019]), 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 to build a conceptual framework 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. Thus, the first contribution of this dissertation can be stated as follows.

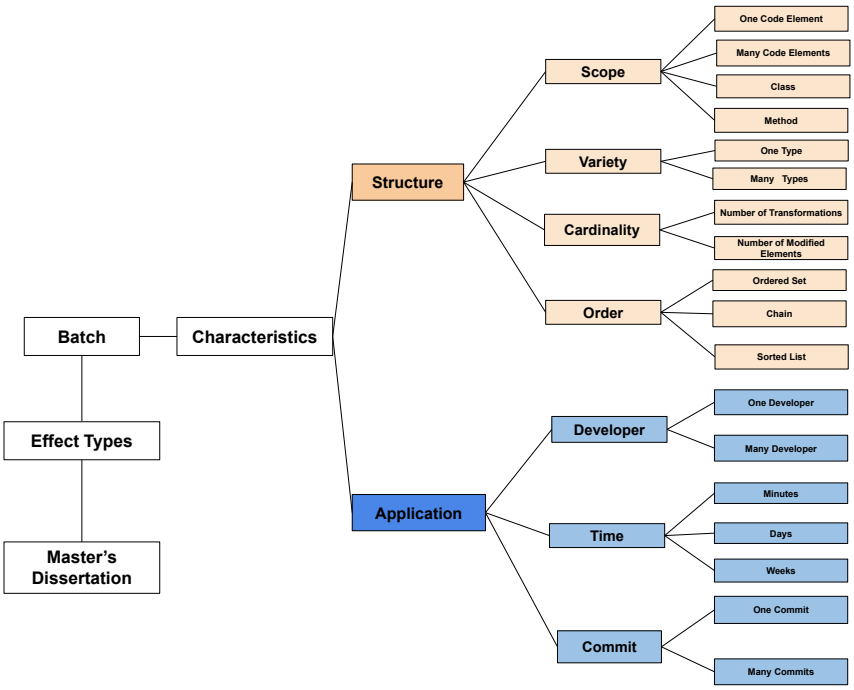

Figure 2. Conceptual Framework of Batch Refactoring

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

On our RQ_1, we classified the characteristics in two groups: Application and Structure. 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. The Structure Group is composed of four batch characteristics: *scope*, *variety*, *cardinality*, and *order*. These characteristics regard internal aspects of the batch composition. In order words, these characteristics reflect the internal batch structure.

On our RQ_2, we have positive and negative types of batch effect. Three positive types of 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. 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.

On our third RQ, we have found some cases of conflict among studies with respect to the characteristics and effect types that constitute a batch. **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 [Kuhlemann et al. 2010, Meananeatra 2012]. Conversely, other studies assume that a batch refactoring can affect *multiple classes* [Kim et al. 2014, Ouni et al. 2013a, Cedrim 2018]. 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: Are batches more likely to improve internal code structures rather than degrade these structures?** We have found studies, such as [Kim et al. 2014, Mkaouer et al. 2014, Ouni et al. 2013b], that point out batches as means for *improving the internal quality of software projects*. Conversely, a particular study [Ó Cinnéide et al. 2012] 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 [Kim et al. 2016, Qayum et al. 2010] 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 [Lin et al. 2016] 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.

After we defined our conceptual framework, we defined and implemented a heuristic to detect batches based on observations of our literature review. Our literature review revealed that previous studies often defined a batch refactoring as a set of interrelated code transformations applied by the *same developer* and on a *single code element*. Based on that, our 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 a *code element*, our heuristic is called *element-based*. However, even in situations were a batch refactoring may end up affecting two or more elements, the outcomes of our heuristic can be composed to study these batches with wider scope. Therefore, we have our second contribution:

Contribution 2: A element-based heuristic to detect batch refactorings. Our heuristic can guide future research towards collecting and investigating batches applied in practice as well as practitioners that want to reason about their previously performed

batch refactorings.

The use of this heuristic has enable us to study the relationship of batch refactoring and internal code attributes and code smells (Section 5). This heuristic was key to support followup research (in collaboration with other colleagues) to identify the relationship between batches and developer's intents [Paixão et al. 2020], refactoring evolution [Paixão et al. 2020], good or bad refactoring practices [Sousa et al. 2020], refactoring customization and non-functional concerns (papers under submission for these two latter studies). Our literature review and the empirical study in Section 5 along the MSc research served to pinpoint advantages and limitations of our proposed heuristic for batch detection. These limitations also have been addressed in a join work with other colleagues in our research group [Sousa et al. 2020].

4. A Large Study of Batch Characteristics and Structural Effect

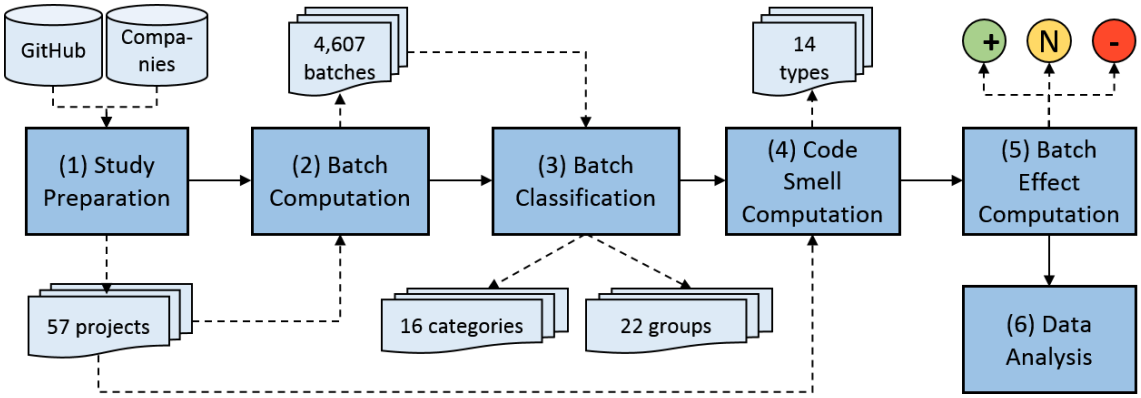

Figure 3. Study Steps

Figure 3 presents the steps of our second study [Bibiano 2019] 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 created a batch detection heuristic based in our literature review (Section 3) 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 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. Table 2 presents the batches identified according to their single or hybrid types. They are often composed of three or more code transformations (28%) and affect much more than only one method (9%) on the same class. 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 3: 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 [Fowler 1999] 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.

This study enable to elicit a comprehensive set of batch characteristics, which were derived through our literature review (Section 3). In previous studies, each batch characteristic was arbitrarily chosen (and defined) for investigation. The empirical understanding of frequent manifestations of each batch characteristic is a useful instrument for researchers. This understanding can help them in refining heuristics for computing batches applied by developers in practical settings, as we recently did in a study [Sousa et al. 2020]. For instance, we observed that it is common that code transformations in batches move code elements to one or more classes. Thus, a heuristic can compute a batch through the code transformations applied on multiple classes [Sousa et al. 2020].

Finally, we assessed the effect of batches on code maintainability 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 analyzed 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 batches.

Our results suggest that most batch refactorings either introduce (51%) or do not suffice to remove (38%) code smells. This observation is quite different to that obtained in previous studies of single code transformations, which found that the vast majority of refactorings our neutral (e.g., see [Cedrim et al. 2017]). More importantly, this finding suggests that, even with the potential to enhance code structures, batches are still poorly applied by developers. Thus, one should consider to propose mechanisms for recommending useful batches that are able to fully remove code smells while assisting the developer to achieve its other underlying (either structural or non-structural, explicit or implicit) goals. For instance, a batch composed by Pull Up Methods to induce the removal of a Message Chain code smell. Thus, our third contribution can be stated as follows.

Table 2. Frequency of Batches According to their Single or Hybrid Nature

Group ID	Nature ID(s)	Batches (%)
G1	**Extraction**	**1,449 (31.4%)**
G2	**Motion**	**1,142 (25%)**
G3	**Rename**	**580 (12.6%)**
G4	**Pull Up**	**352 (7.6%)**
G5	**Inline, Extraction**	**233 (5%)**
G6	**Inline**	**221 (5%)**
G7	Extraction, Motion	153 (3.3%)
G8	Extraction, Rename	105 (2.3%)
G9	Pull Up, Motion	84 (2%)
G10	Motion, Rename	59 (1.3%)
G11	Push Down	58 (1.3%)
G12	Inline, Motion	52 (1.1%)
G13	Inline, Extraction, Motion, Rename	28 (0.6%)
G14	Inline, Rename	25 (0.5%)
G15	Pull Up, Extraction	18 (0.4%)
G16	Push Down, Extraction	11 (0.2%)
G17	Push Down, Pull Up, Inline, Extraction, Motion, Rename	11 (0.2%)
G18	Pull Up, Inline	8 (0.2%)
G19	Pull Up, Rename	7 (0%)
G20	Push Down, Pull Up	6 (0%)
G21	Push Down, Inline	3 (0%)
G22	Push Down, Rename	2 (0%)
	Total	**4,607 (100%)**

Contribution 4: 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.

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 [Kim et al. 2014, Murphy-Hill et al. 2012]. In this context, a previous study [Cedrim 2018] evaluated empirically the batch effect on code smell introduction and removal. Unfortunately, that study did not evaluate systematically the batch effect and they 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 large data set of 57 software projects, plus a set of batch characteristics extracted from our literature review (Section 3), we impact the current knowledge of refactoring and revealed additional aspects of the batch effect on code smells. Similarly to the previous work [Cedrim 2018], 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 [Cedrim et al. 2017, Fowler 1999] more likely 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. In this vein, our results revealed some patterns of batch refactorings that can be recommended to developers, which were not yet reported or exploited by the literature. Many of these patterns are not ad-

dressed either in Fowler's refactoring recommendations for removing certain code smell stypes.

5. Insights to Enhance Current Refactoring Support

The empirical studies of this work [Bibiano 2019] have also enabled us to reveal limitations and some opportunities for refining current tools and methods to support batch refactoring. We discuss these improvements as follows.

Extending Refactoring Tools to Recommend Batches for Code Smell Removal. Previous studies (e.g., [Bavota et al. 2011, Tsantalis and Chatzigeorgiou 2011] and even more recent ones) introduced tooling support, heuristics or recommendation systems for guiding the application of isolated code transformations along code refactoring aiming at supporting 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 interrelations 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 [Cedrim 2018, Kim et al. 2014, Murphy-Hill et al. 2012], the automated guidance of batch refactoring is desired.

Improving the Refactoring Recommendation for Long Method Removal. An existing robust tool, called JDeodorant [Tsantalis and Chatzigeorgiou 2011], recommends the Extract Method application aiming to remove a Long Method. However, our results show that mere application of a single Extract Method quite often leads to an extracted method that is also a Long Method or clearly becomes a Feature Envy. 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. Moreover, the applied refactoring might have highlighted a more severe problem, i.e., the presence of envy code. Robust tools, such as JDeodorant, could be extended to progressively (or in one go) recommend batches with the proper amount of Extract Methods (and Move Methods) to fully resolve the entire smelly structure affecting the host class.

Improving the Refactoring Recommendation for Feature Envy Removal. The existing catalogs recommend batches composed by Extract Methods and Move Methods to remove a Feature Envy [Fowler 1999]. In our results, surprisingly, we have found that batches composed by these transformations have also been introducing 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 are the proper methods to 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. 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. [Lin et al. 2016] 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. However, Refactoring Navigator departs from the assumption that the developer often knows the intended design. An analysis of several examples of batches in project repositories in our study, however, revealed that developers more often do not have a fixed, target design. Our patterns of smell-removal batches can clearly help in those cases.

6. Conclusion

In this paper, 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 [Cedrim 2018]. Each set of interrelated code transformations is called a *batch* [Cedrim 2018].

Unfortunately, the current knowledge about batch refactoring is quite scarce, which makes hard to support developers in their daily refactoring practices. Therefore, little is known about the characteristics that constitute a batch in practice. Moreover, there is limited empirical evidence about the effect of batches on code maintenance [Cedrim 2018]. As an implication, developers may still feel reluctant in applying batches on their projects [Kim et al. 2014], due to the fear of worsening rather than enhancing code structures.

Aimed to address the literature gaps mentioned above, this work 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 batch characteristics and batches affect types. 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. We expect that some of the outcomes of this paper can guide developers in their daily work while revealing opportunities for future research work on the topic.

References

Bavota, G., Lucia, A. D., and Oliveto, R. (2011). Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software (JSS)*, 84(3):397–414.

Bibiano, A. C., Fernandes, E., Oliveira, D., Garcia, A., Kalinowski, M., Fonseca, B., Oliveira, R., Oliveira, A., and Cedrim, D. (2019). A quantitative study on characteristics and effect of batch refactoring on code smells. In *13th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11.

Bibiano, A. C. G. (2019). *Understanding Characteristics and Structural Effects of Batch Refactorings in Practice*. PhD thesis, PUC-Rio.

Cedrim, D. (2018). *Understanding and Improving Batch Refactoring in Software Systems*. PhD thesis, Informatics Department (DI), Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil.

Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M., and Chávez, A. (2017). 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)*, pages 465–475.

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., and Garcia, A. (2017). How does refactoring affect internal quality attributes? A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES)*, pages 74–83.

Ferreira, I., Fernandes, E., Cedrim, D., Uchôa, A., Bibiano, A. C., Garcia, A., Correia, J. L., Santos, F., Nunes, G., Barbosa, C., et al. (2018). 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*, pages 406–407.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1st edition.

Jalali, S. and Wohlin, C. (2012). Systematic literature studies: Database searches vs. backward snowballing. In *Proceedings of the 6th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 29–38.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU-SEI-90-TR-21 and ESD-90-TR-222, Software Engineering Insttitute (SEI), Carnegie Mellon University (CMU).

Kim, J., Batory, D., Dig, D., and Azanza, M. (2016). Improving refactoring speed by 10x. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1145–1156.

Kim, M., Zimmermann, T., and Nagappan, N. (2014). An empirical study of refactoring: Challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering (TSE)*, 40(7):633–649.

Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical report, EBSE 2007-001, Version 2.3, Keele University and University of Durham.

Kuhlemann, M., Liang, L., and Saake, G. (2010). 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)*, pages 37–48.

Lin, Y., Peng, X., Cai, Y., Dig, D., Zheng, D., and Zhao, W. (2016). Interactive and guided architectural refactoring with search-based recommendation. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE)*, pages 535–546.

Meananeatra, P. (2012). Identifying refactoring sequences for improving software maintainability. In *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*, pages 406–409.

Mkaouer, M. W., Kessentini, M., Bechikh, S., Deb, K., and Ó Cinnéide, M. (2014). Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th International Conference on Automated Software Engineering (ASE)*, pages 331–336.

Murphy-Hill, E., Parnin, C., and Black, A. (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering (TSE)*, 38(1):5–18.

Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S., and Hemati Moghadam, I. (2012). Experimental assessment of software metrics using automated refactoring. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 49–58.

Ouni, A., Kessentini, M., and Sahraoui, H. (2013a). Search-based refactoring using recorded code changes. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 221–230.

Ouni, A., Kessentini, M., Sahraoui, H., and Hamdi, M. S. (2013b). 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)*, pages 1461–1468.

Paixão, M., Uchôa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J., and Arvonio, E. (2020). Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *17th Mining Software Repositories (MSR)*.

Qayum, F., Heckel, R., Corradini, A., Margaria, T., Padberg, J., and Taentzer, G. (2010). Search-based refactoring based on unfolding of graph transformation systems. In *Proceedings of the 5th International Conference on Graph Transformation (ICGT): Doctoral Symposium (DS)*, pages 1–14.

Sousa, L., Cedrim, D., Garcia, A., Oizumi, W., Bibiano, A. C., Tenorio, D., Kim, M., and Oliveira, A. (2020). Characterizing and identifying composite refactorings: Concepts, heuristics and patterns. In *17th Mining Software Repositories (MSR)*.

Stol, K.-J., Ralph, P., and Fitzgerald, B. (2016). Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 120–131.

Tsantalis, N. and Chatzigeorgiou, A. (2011). Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software (JSS)*, 84(10):1757–1782.

Yamashita, A. and Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*, pages 306–315.