ANALISANDO OS EFEITOS DA REFATORAÇÃO EM *BAD SMELLS*

CLEITON SILVA TAVARES

ANALISANDO OS EFEITOS DA REFATORAÇÃO EM *BAD SMELLS*

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha
Coorientador: Eduardo Magno Lages Figueiredo

Belo Horizonte
Março de 2021

CLEITON SILVA TAVARES

ANALYZING THE EFFECTS OF REFACTORINGS ON BAD SMELLS

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Mariza Andrade da Silva Bigonha
Co-Advisor: Eduardo Magno Lages Figueiredo

Belo Horizonte
March 2021

Tavares, Cleiton Silva.

T231a Analyzing the effects of refactorings on bad smells [manuscrito] / Cleiton Silva Tavares – 2021.
xviii, 78 f. il.

Orientadora: Mariza Andrade da Silva Bigonha.
Coorientador: Eduardo Magno Lages Figueiredo.
Dissertação (mestrado) – Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.67-75.

1. Computação – Teses. 2. Engenharia de Software – Teses. 3. Software - Refatoração– Teses. 4. Bad smell – Teses. I. Bigonha, Mariza Andrade da Silva. II. Figueiredo, Eduardo Magno Lages. III Universidade Federal de Minas Gerais; Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV. Título.

CDU 519.6*32.(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Analyzing the Effects of Refactorings on Bad Smells

CLEITON SILVA TAVARES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]
Profa. Mariza Andrade da Silva Bigonha - Orientadora
Departamento de Ciência da Computação - UFMG

Eduardo Figueiredo
Prof. Eduardo Magno Lages Figueiredo - Coorientador
Departamento de Ciência da Computação - UFMG

André Cavalcante Hora
Prof. André Cavalcante Hora
Departamento de Ciência da Computação - UFMG

[illegible]
Profa. Kécia Aline Marques Ferreira
Departamento de Computação - CEFET-MG

Belo Horizonte, 31 de Março de 2021.

Resumo

Refatoração visa aumentar a manutenibilidade de sistemas de software melhorando a sua estrutura sem alterar seu comportamento, podendo ser aplicada para remover *bad smells*. Mesmo com a existência de ferramentas para auxiliar a refatoração, muitos desenvolvedores não confiam em suas soluções, alegando que alguns estudos mostram que a refatoração pode introduzir novos *bad smells* no código-fonte. Contudo, não encontramos um catálogo completo que indique quando isso ocorre. Para investigar esse assunto em detalhe, o objetivo desta dissertação é avaliar os efeitos da refatoração em *bad smells*. Especificamente, investigamos se e qual refatoração remove *bad smells* ou os introduz. Para atingir esse objetivo, realizamos uma Revisão Sistemática da Literatura (RSL) para identificar a relação entre as refatorações e os *bad smells* propostos por Fowler. Conduzimos um estudo empírico com oito sistemas de *software*, aplicando cinco refatorações para analisar seus efeitos em dez *bad smells* com o auxílio de cinco ferramentas. Como resultado do estudo empírico, apresentamos via os dados estudados, quais *bad smells* são removidos ou introduzidos pelo processo de refatoração automatizada. A RSL resultou em 20 artigos mostrando a relação direta entre 31 refatorações e 16 *bad smells*. Produzimos um catálogo exibindo essas relações e também apresentamos um contraste com as relações discutidas por Fowler. Identificamos que a relação mais discutida na literatura se dá entre *Move Method* e *Feature Envy*. A RSL também revelou que existem estratégias de refatoração diferentes daquelas discutidas por Fowler para lidar com *bad smells*. No estudo empírico, observamos que os tipos de refatoração geraram diminuição, aumento e variações neutras no número de *bad smells*. Diferente da definição de Fowler, surpreendentemente descobrimos que a diminuição no número de *bad smells* foi a mais baixa em comparação com casos de aumento e variações neutras. Em uma análise adicional, contrastamos os resultados encontrados nos dois estudos realizados, classificando-os, validando-os e complementando-os.

Palavras-chave: Refatoração, *Bad Smells*, Impactos da Refatoração, Efeitos da Refatoração.

Abstract

Refactoring aims to increase software systems' maintainability by improving their structure without changing their behavior, may applied to remove bad smells. Even with tools to assist refactoring, many developers do not trust their solutions, claiming that some studies show that refactoring can introduce new bad smells into the source code. However, we have not found a complete catalog that states when this may occur. To investigate this subject deeply, the goal of this dissertation is to evaluate the effects of refactoring on bad smells. Specifically, we want to know if and what refactoring removes bad smells or introduces them. To achieve this goal, we conducted a Systematic Literature Review (SLR) to identify the relationship between refactorings and bad smells proposed by Fowler. We also conducted an empirical study with eight software systems applying five refactorings to analyze their effects on ten bad smells with the assist of five tools. As a result of the empirical study, we present, through the data studied, which bad smells tend to be removed or introduced by the automated refactoring process. In the SLR, we found 20 papers showing the direct relationship between 31 refactorings and 16 bad smells. We produced a catalog showing these relationships, and we also showed a contrast with relationships discussed by Fowler. We identified that the most discussed relationship in the literature is between Move Method and Feature Envy. The SLR also revealed different refactoring strategies than those discussed by Fowler for dealing with bad smells. In the empirical study, we observed that refactoring generated decrease, increase, and neutral variations in the number of bad smells. Unlike Fowler's definition, we surprisingly found that the number of bad smells decrease was the lowest compared to cases of increase and neutral variations. In an additional analysis, we contrast the results found in the two studies carried out, classifying, validating and complementing them.

Keywords: Refactoring, Bad Smell, Refactoring Impacts, Refactoring Effects.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Software systems must evolve to cope with new requirements from stakeholders. Hence, they require a great effort for developers to understand and make the necessary modifications in the source code. This effort is greatly affected by aspects of the source code's quality, such as comprehensibility, complexity, and maintainability. *Bad Smells* are indicators that there are code structure problems, which refactoring may solve [37]. *Refactoring* is a strategy used to increase the maintainability of the code by changing the source code's internal structure without changing the software system behavior. Refactoring is often recommended to solve bad smells [34].

We may find in the literature numerous tools for detecting bad smells [14, 26, 30, 40, 51, 64, 70] and for refactoring [35, 36, 42, 60, 72, 73, 74]. However, some studies show that the refactoring process often does not solve the source code's bad smells [7, 43]. Furthermore, other studies show evidence that the refactoring process may not only remove bad smells in the source code but also may introducing new ones [54, 65, 82]. Such a problem may lead the developer not to trust automated refactoring since there are high chances of not completely removing the bad smells present in the code.

1.1 Motivation

Refactoring is a tricky activity since developers have to analyze the source code (i) to identify the code fragment to refactor, (ii) the operation that will solve it, and (iii) where allocate the refactored code. These steps are challenging since they depend on the system context, although we may address them using automatic refactoring tools. However, some developers do not use these tools because they a) do not know when and how to refactor [49], b) do not fully trust the tool behavior [45], and c) because existing refactoring tools may introduce new bad smells after the refactoring process [84]. As a result, many developers prefer to refactor code manually, which leads them to face the mentioned challenges.

Studies that perform analysis based on the refactoring process argue that refactoring is effective in removing bad smells in less than 10% of the time [7, 15, 16]. Therefore, refactoring should be done in a disciplined manner to minimize the chances of introducing (i) another bad smell [15, 16, 54, 82] and defects [8, 59, 69]; and (ii) ensuring that the quality was enhanced [7, 34]. For instance, Xia et al. [87] have conducted an empirical study by surveying developers, and they have identified the wide use of refactoring. However, participants reported that this activity is often neglected in academia, focusing on writing code from scratch. Therefore, they suggest that educators emphasize in classes the process of identifying and eliminating bad smells present in the source code applying refactorings.

Even though some indications that the refactoring process may remove bad smells in the source code, there is no complete catalog that shows which refactoring process may even introduce bad smells. A catalog containing information about which refactoring process introduces bad smells may help developers perform the refactoring, whether manual or automated. With this information, developers may perform the most efficient and robust refactoring process to avoid introducing new bad smells in the source code.

1.2 Proposed Work

In this dissertation, we propose a research study to evaluate refactoring operations' impact on bad smells. We divided our study into two stages: the Systematic Literature Review (SLR) and the empirical study. To conduct both studies, we defined Research Questions (RQs) to guide us in each of them. The SLR is composed of two general RQs, **RQ1** and **RQ2**, and two specific **RQ1.1** and **RQ1.2**, defined as follows.

RQ1 - Which relationships between refactoring and bad smells are the literature explicitly discussing?

RQ1.1 - Which are the most mentioned relationships between bad smells and refactoring found in the literature?

RQ1.2 - Are the relationships we found different from those that Fowler presents?

RQ2 - Which tools found in papers perform refactoring from bad smell detection?

The empirical study is composed of one general RQ, **RQ3**, and two specific **RQ3.1** and **RQ3.2**. The RQs defined are as follows.

RQ3 - What are the impacts of automated refactoring on the detection of bad smells?

RQ3.1 - Does the automated refactoring process remove bad smells?

RQ3.2 - Does the automated refactoring process introduce bad smells?

With the SLR, we were able to identify the relationships between bad smell and refactoring discussed in the literature. In this study, we report only the situations described by the authors of the papers found, without entering into the merit of classification between positive or negative impacts. We found 20 papers that show the direct relationship between 31 refactoring types and 16 bad smells proposed by Fowler. We also found seven tools that apply refactoring after detecting bad smells. We identified that the most discussed relationship in the literature is between Move Method and Feature Envy. It also revealed that there are different refactoring strategies of those discussed by Fowler to address bad smells. The literature focuses mostly on strategies defined in Fowler's book [34] and shows that most refactoring tools do not detect bad smells.

After conducting the SLR, we choose some refactorings and bad smells available in the Fowler's catalog [34] and mentioned in the literature to conduct our empirical study. With the empirical study, we analyzed the impacts caused by the automatic refactoring process on bad smells. We applied five refactorings (see Section 4.2) and measured their effect on ten bad smells detected by five tools. We defined four perspectives to enable the analyses of the effects caused by the refactoring process. We observed that these perspectives had similar behavior. The refactorings types generate a decrease, increase, and neutral variations in the number of bad smells. Unlike Fowler's definition, we surprisingly found that the number of bad smells decrease was the lowest compared to the others. We also investigated which bad smells tend to be introduced and removed by automatic refactoring. Finally, after the results obtained and presented by four perspectives, we also present the aggregated data analysis.

With our SLR, we initially present only a relationship between bad smells and refactoring' while with our empirical study, we present which bad smells were introduced and removed by the refactoring process. Therefore, as one of the analyses, we contrast the results found in both studies conducted.

1.3 Publications

This dissertation generated the following publications:

- Silva, Cleiton; Santana, A.; Figueiredo, Eduardo; Bigonha, Mariza A. S., Revisiting the Bad Smell and Refactoring Relationship: A Systematic Literature Review. *23rd Iberoamerican Conference on Software Engineering (CIbSE), Experimental Software Engineering (ESELAW).* Curitiba (online), Brazil, 2020.

- Tavares, Cleiton; Bigonha, Mariza; Figueiredo, Eduardo, Analyzing the Impact of Refactoring on Bad Smells (short paper). *Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES)*, pages 97-101. Natal (online), Brazil, 2020. https://doi.org/10.1145/3422392.3422408.

- Tavares, Cleiton; Bigonha, Mariza A. S.; Figueiredo, Eduardo, Quantifying the Effects of Refactorings on Bad Smells. *10th Workshop on Theses and Dissertations - Master Students (WTDSOFT).* Natal (online), Brazil, 2020.

1.4 Dissertation Outline

We organized this Master dissertation into six chapters. This chapter introduced this dissertation.

Chapter 2 presents a theoretical reference to assist this dissertation's comprehension. It includes the main concepts related to the study, such as bad smell, refactoring, and tools to support these operations. It also presents some related works disposed of the literature. We discuss them in two aspects, (i) the bad smell and refactoring subject and (ii) the refactoring impacts.

Chapter 3 describes the Systematic Literature Review and presents the protocol used to conduct it, their execution, the analysis and, the results obtained after its application.

Chapter 4 defines the empirical study design to assess the automatic refactoring process's impacts. It also defines the refactoring strategies we analyzed and the bad smells used for its evaluation.

Chapter 5 presents the execution of the empirical study described in Chapter 4. It presents the results obtained and also additional analyses covering a broad vision of the data. It also contrasts the results obtained in the empirical study compared to the systematic literature review.

Chapter 6 concludes the dissertation with a discussion regarding the results of these studies and their importance for the software engineering area. We summarize the contributions of the study and give suggestions for future work.

Chapter 2

Background and Related Work

Software engineering brings with it different phases and processes for the development of a software system. One of its aspects is understanding what a bad smell is and when to apply the refactoring. Therefore, in this chapter, we present the definition used in this dissertation for the terms bad smell and refactoring in Section 2.1 and Section 2.2, respectively. Moreover, we present in Section 2.3 an overview of some tools used to (i) detect bad smells and (ii) support the refactoring process. To identify these tools, we used systematic literature reviews and a mapping study from the literature [22, 28, 47, 75, 79]. Moreover, we perform an ad-hoc search to identify tools that were recently published.

Based on what was presented by Fowler et al. [34] and discussions available in the literature, we may establish a direct relationship between the application of refactoring to solve the bad smells existing in the source code. In this way, we may identify different situations in which the literature discussed these subjects. There are several literature reviews in the context of bad smells [22, 28, 58, 61, 88] and refactoring [1, 2, 53, 79]. Although most studies deal with these subjects individually, focusing just on one theme, others deal with more than one theme, and still, others discuss these subjects together. Therefore, Section 2.4 presents some studies that discuss these subjects using different ways. The literature also presents several works discussing different approaches to evaluate the refactoring operation [7, 15, 23, 31], in Section 2.5 we discuss some of them. Finally, Section 2.6 presents the final remarks of this chapter.

2.1 Bad smell

Bad smell is an evidence of problems in the code structure that may use refactoring to solve it [37]. Fowler et al. [34] proposed one of the most complete lists containing 22 bad smells; besides that, they describe how we may identify them and what refactoring strategies we may apply in their solution. In recent book edition, Fowler [33] presents a new list containing 24 bad smells, where Fowler preserves much of the list presented in its first work, updates some terms, and adds new bad smells assuming that the context of bad smells is differentiated than presented in his first version.

Feature Envy is an example of a bad smell that occurs when a function in one module spends more time communicating with functions or data inside another module than it does within its module.

Algorithm 2.1 presents a Feature Envy example adapted from Elarning.[1] To allow the user to distinctly the numbers that make up a North American mobile phone number, we built the Phone class. In the Customer class, the method *getMobilePhoneNumber()* provides a North American-formatted mobile phone number. In this scenario, we can identify a Feature Envy. This fact is a piece of evidence in how customer reaches into phone's data to format the number, showing up a misplaced responsibility.

```java
public class Phone {
  private final String unformattedNumber;
  public Phone(String unformattedNumber) {
    this.unformattedNumber = unformattedNumber;
  }
  public String getAreaCode() {
    return unformattedNumber.substring(0,3);
  }
  public String getPrefix() {
    return unformattedNumber.substring(3,6);
  }
  public String getNumber() {
    return unformattedNumber.substring(6,10);
  }
}

public class Customer {
  private Phone mobilePhone;
  public String getMobilePhoneNumber() {
    return "(" +
    /* start feature envy code */
```

[1]https://elearning.industriallogic.com/gh/submit?Action=PageAction&album=recognizingSmells&path=recognizingSmells/featureEnvy/featureEnvyExample&devLanguage=Java

```
    mobilePhone.getAreaCode() + ") " +
    mobilePhone.getPrefix() + "-" +
    mobilePhone.getNumber();
    /* end feature envy code */
  }
}
```

Listing 2.1: Feature Envy Example

2.2 Refactoring

Refactoring is a process that improves the software system's internal structure without changing the code's external behavior. One of the most well-known and complete catalogs presents a list of 72 refactorings [34]. In his most recent book edition, Fowler [33] updates your refactorings catalog considering a more current context and following software development standards used today, assuming that the refactoring had become a standard tool for any skilled programmer. In this new version, the author catalogs 61 refactorings, although its *Web page*[2] displays a catalog with 66 refactorings, including those refactorings described in the book Refactoring 2nd Edition, together with the Ruby Edition.

Extract Method is an example of its refactoring, which consists of turning a code fragment into its method to improve your clarity. We may perform this refactoring when: (1) a method is too long, (2) it is difficult to understand the purpose of a code fragment, and (3) we want to finely grain a method, increasing the chances that other methods can use it.

Another refactoring example is the *Move Method*. This refactoring consists of moving to another class a method, which seems to reference another object more than the object it lives on. The classes may be made simpler by moving methods and end up being a more crisp implementation of a set of responsibilities. We may perform this refactoring when: (1) classes have too much behavior, and (2) besides high coupled, the classes collaborating too much.

As presented in Algorithm 2.1, which presents a Feature Envy bad smell example, a possible way to solve this bad smell would be the refactoring application process. Algorithm 2.2 presents a possible refactoring strategy that solves the bad smell jointly applying Extract Method and Move Method refactoring. This solution is an example

[2] `https://refactoring.com/catalog/`

adapted from Elarning.[3] We can observe that after the application of the refactoring process, the customer relies on Phone to do the formatting.

```java
public class Phone {
  private final String unformattedNumber;
  public Phone(String unformattedNumber) {
    this.unformattedNumber = unformattedNumber;
  }
  private String getAreaCode() {
    return unformattedNumber.substring(0,3);
  }
  private String getPrefix() {
    return unformattedNumber.substring(3,6);
  }
  private String getNumber() {
    return unformattedNumber.substring(6,10);
  }
  /* extract and move method applied */
  public String toFormattedString() {
    return "(" + getAreaCode() + ") " + getPrefix() + "-" +
        getNumber();
  }
}

public class Customer {
  private Phone mobilePhone;
  public String getMobilePhoneNumber() {
    /* extract and move method applied */
    return mobilePhone.toFormattedString();
  }
}
```

Listing 2.2: Extract and Move Method Example

[3]https://elearning.industriallogic.com/gh/submit?Action=PageAction&album=recognizingSmells&path=recognizingSmells/featureEnvy/featureEnvyExample&devLanguage=Java

2.3 Refactoring and Bad Smell Detection Tools

This section presents the tools chosen to assist the research study conducted in our dissertation.

DECOR/DETEX/PTIDEJ[4] [55]. Ptidej is a tool suite for evaluating and improving the quality of object-oriented programs, reverse-engineering (AOL, C/C++, Java), and promoting patterns. DECOR is a method of detection for code and design smells. DETEX, an instantiation of DECOR, uses a unified vocabulary and a dedicated language that allows the specification and the detection of code smells and antipatterns. Ptidej integrates DECOR, but in this dissertation we use the Eclipse Java project to identify code and design smells in the analyzed source code.

Designite[5] [71]. Designite is a tool used to assess the software system design quality. It analyzes C# code and identifies software quality issues to reduce technical debt and improve a software system's maintainability. DesigniteJava detects numerous design and implementation smells in its Java version and computes many commonly used object-oriented metrics. In this dissertation, we use Designite-Java Community, which is free and open-source. It also has an Enterprise edition, offering additional features than the Community edition.

JDeodorant[6] [29]. JDeodorant is an Eclipse plugin that employs various methods and techniques to identify code smells and solves them by suggesting the appropriate refactorings to be applied.

JSpIRIT[7] [84]. JSpIRIT (Java Smart Identification of Refactoring opportunITies) is a plugin for the Eclipse IDE that helps the developer to prioritize code smells and identify code smells and their possible agglomerations. It provides different rankings using different criteria, such as the application's history, the relevance of the code smell, or modifiability scenarios to prioritize the code smells.

Organic[8] [56]. Organic is a plugin for the Eclipse IDE designed to identify design problems in Java software systems' source code. It also identifies the relationships between code-anomaly agglomerations [57, 68] and software design degradation. However, it does not provide a user interface or user interaction. It collects code smells from Java projects using only command lines.

2.4 Bad Smell and Refactoring

Sousa et al. [77] present a systematic literature mapping of studies investigating the relationship between design patterns and bad smells. The authors focus on co-occurrence between design patterns and bad smells, providing a general analysis of the relationship between the GOF design patterns [25] and bad smells described by Fowler et al. [34].

Singh and Kaur [76] perform a systematic literature review of refactoring concerning code smells. Several data sets and tools for performing refactoring have been revealed and categorized depending on the detection approach: traditional method, visualization-based technique, automatic method, semi-automatic method, empirical studies, and metric-based method.

Kaur and Singh [44] conduct a systematic mapping study evaluating the effect of refactoring activities on software quality attributes of existing empirical studies. They considered and presented code smells with a small and general focus on the studies found. They select 142 primary studies following a multi-stage scrutinizing process and classified them along with different aspects, which enable them to provide various findings. One of their findings shows that the individual refactoring activities have variable effects on most quality attributes explored in primary studies, indicating that refactoring does not always improve quality attributes.

Lacerda et al. [47] present a tertiary systematic literature review of previous surveys, secondary systematic literature reviews, and systematic mappings on code smell and refactoring. They show that both subjects have a strong relationship with quality attributes, such as understandability, maintainability, testability, complexity, functionality, and reusability. Besides, they present a visualization that analyzes the relationship among quality attributes, which code smells affected them, which refactoring may be applying to these code smells, and its impact when using such refactoring.

Bafandeh Mayvan et al. [5] present an approach for bad smell detection in code based on a multi-step process using software quality metrics and refactoring opportunities. First, it obtained the bad smell formal specifications based on software metrics. After that, it uses them to achieve a set of candidates for each bad smell. Finally, each of the instances is examined and compared with the corresponding refactoring situations specified for that bad smell, striking out the false positives created in the previous stage. They evaluated and demonstrated the effectiveness of this approach.

Sousa et al. [77] studied the relationship between design patterns and bad smells. Regarding the works done by Singh and Kaur [76], Kaur and Singh [44], Lacerda et al. [47], and Bafandeh Mayvan et al. [5], even though these authors presented the context of refactoring and code smells, they do not explored enough the relationship between each other considering the Fowler's catalog [34]. Therefore, unlike these works, we want to focus on this subject profoundly, investigate how the literature discusses and construct a complete list of the relationship between bad smell and refactoring presented by Fowler et al. [34].

2.5 Refactoring Impacts

Bavota et al. [7] mined the evolution history of three Java open-source projects to investigate whether refactoring activities impact quality metrics or bad smells, suggesting a need for refactoring operations. According to their results, quality metrics usually do not show a clear relationship with refactoring; 42% performed refactoring operations on code entities affected by code smells and, only 7% of the performed operations remove the code smells.

Cedrim et al. [16] analyzed the version histories of 25 projects to find out how 2,635 refactorings distributed in 11 different types affect the density of five code smells. Their results show that 95.1% of refactorings did not reduce or introduce code smells, 2.24% of refactoring changes removed code smells, and 2.66% introduced new ones. In another work, Cedrim et al. [15] analyze how 16,566 refactorings distributed in ten different types affect the density of 13 types of code smells in the version histories of 23 projects. Their results reveal that 79.4% of the refactorings touched smelly elements, 57% did not reduce their occurrences, 9.7% of refactorings removed smells, and 33.3% induced new ones. They also characterized and quantified that 30% of the Move Method and Pull Up Method induced God Class's emergence, and Extract Superclass in 68% of the cases created Speculative Generality.

Fontana and Spinelli [31] analyze the impact of refactoring applied to remove code smells. They select four bad smells defined by Fowler, detected by three tools, and apply the refactoring automatically using two tools. They performed this process on the open-source, object-oriented system of about 400 classes. They analyzed and reported a summary of the impact according to six metrics proposed to evaluate the system's code and its design quality.

Du Bois and Mens [23] presented a formalism for describing the impact of refactorings on internal program quality metrics as indicators of quality factors. The authors described the formalist for three refactorings and six internal metrics. The abstract syntax tree performed the representation of the source code extended with cross-references. They elucidate how these metrics can be formally defined on top of this program structure representation and demonstrate how to project the impact of refactorings on their values in the form of potential drifts or improvements.

Chaparro et al. [17] present a technique that estimates the impact of refactoring operations on source code quality metrics, named RIPE (Refactoring Impact PrEdiction). It supports 12 Fowler refactoring operations, the ones dealing with generalization and composing methods, and 11 metrics that can be used together to refactoring recommendation tools. They evaluate this technique and estimate the impact on 8,103 metric values for 504 refactorings from 15 open-source Java systems. In general, 38% of the estimates are correct, whereas the median deviation of the actual values' estimates is 5% (with a 31% average).

Alshayeb [3] investigates whether refactoring to patterns improves software quality. This investigation was realized empirically by examining the metric values of five external quality attributes for three open-source Java systems before and after nine refactorings to patterns is applied. These refactorings were applied manually, but the author did not apply all refactorings to each system. He did not found consistent improvement in software quality attributes because each refactoring to patterns technique has a particular purpose and effect, affecting software quality attributes differently.

Bibiano et al. [11] analyzed 19 smell types and 13 transformation types in 4,607 batches, each applied by the same developer on the same code element. They computed the frequency in which five batch characteristics manifest, the probability of each batch characteristics to remove smells, and the frequency in which batches introduce and remove smells. According to their results, batches that apply on a single method are more prone to removing smells than batches affecting more than one method. Still, batches 51% of the time ended up introducing or 38% of the time not entirely removing smells.

Eposhi et al. [24] evaluated the impact of refactoring, focused on removing design problems, on the density and diversity of symptoms involving two C# systems. Their results show that refactorings caused almost no positive impact on the density and diversity of symptoms. However, the density and diversity of symptoms, such as the violation of object-oriented principles, were not predominantly higher.

Fernandes et al. [27] investigate if re-refactoring operations are more effective in improving attributes when compared to single operations. They analyzed 23 open software projects with 29,303 refactoring operations, from which nearly 50% constitute re-refactorings, assessed cohesion, complexity, coupling, inheritance, and size attributes. They combined descriptive analysis and statistical tests to deeply understand the effect of refactoring and re-refactoring on each attribute and revealed that 90% of refactoring operations, and 100% of re-refactoring operations, were applied to code elements with at least one critical attribute. 65% of the operations improve attributes presumably associated with the refactoring type applied and, 35% keep those attributes unaffected.

Unlike Bibiano et al. [11], we focus on studying the relationship between bad smells and refactoring. Different from Eposhi et al. [24], we decided to carry out our analysis based on Java systems. We investigate different Java systems than those of Bavota et al. [7]. Unlike Fontana and Spinelli [31] work, we intend to use many bad smells among those proposed by Fowler et al. [34].

Different from Fernandes et al. [27], we do not distinguish our analysis between refactoring and re-refactoring, even though we identified this situation. Unlike Chaparro et al. [17] and Du Bois and Mens [23], we evaluate the existing literature strategies assessing the impacts that the refactoring process has on the bad smells proposed by Fowler et al. [34]. Cedrim et al. [16], Cedrim et al. [15], and Alshayeb [3] detected refactorings carried out in the source code, we intend to apply refactorings using tools to assist these processes. Finally, as one of the results, we want to provide a catalog showing which bad smells we may remove and which ones, if they exist, we may introduce by refactoring.

2.6 Final Remarks

Bad smells are symptoms present in the source code that indicate any anomaly that refactoring operations may solve [20]. Refactoring aims to remove bad smells and increase software maintainability by improving the software structure without changing its behavior.

One of the precursors of the subject was Fowler, who, in 1999, presented one of the most complete and discussed catalogs in the literature [47]. He recently published a new edition of his work. He remained the definition of the concepts and updated his catalog of bad smells that may exist in a software system and refactorings strategies that a developer may use to improve their internal structure. As this dissertation's scope aimed at analyzing different bad smells and refactorings already consolidated in the literature, we decided to focus on Fowler's first catalog. Moreover, for the rest of this dissertation, we only present the bad smell and refactoring names proposed in the first edition of his work. Also, for the refactoring and bad smell detection tools chosen, we only use the operations provided by them according to the name established by Fowler et al. [34].

Refactoring and bad smells are two terms that may be work together. In the literature, several studies use this principle [5, 7, 15, 16, 31], but many do not explore this subject in detail. It is worth mentioning that both subjects are complete and complex enough to justify studies that focus on only one of them. Someone may find these terms directly linked to countless other software development contexts, and these relationships may also be studied. To identify the relationship between bad smell and refactoring, Chapter 3 conducts a systematic literature review to identify the specific refactorings mentioned to those presented' specific bad smells proposed by Fowler [34].

Chapter 3

Systematic Literature Review on the Relationship Between Bad Smell and Refactoring

Fowler et al. [34] presents a catalog of bad smells and refactoring consisting of 22 bad smells and 72 refactorings, widely discussed in the literature [22, 53, 77]. However, in many cases, the descriptions of bad smells and their relationships with refactorings are not precise and lack a consistent level of details [6]. Moreover, we need more studies in the area, not only in context-based evaluations of bad smells and how to eliminate them [78] but also in research exploring the relationship between refactoring and bad smell [13].

To better understand this relationship, this chapter presents a systematic literature review to find a direct relationship between the bad smell and the refactoring proposed in Fowler's catalog [34]. A Systematic Literature Review (SLR) is a study that provides identification, analysis, and interpretation of evidence related to a particular research topic issue or the phenomenon of interest [86]. We conducted the SLR protocol using the Kitchenham guidelines [46]. The SLR presented as part of this dissertation addresses studies focusing on the direct relationship between refactoring and bad smells. We carried out the SLR in three stages: (i) planning, (ii) execution, and (iii) analysis [46]. The results provide insights to researchers and developers of the existing relationships used to refactor code.

The remainder of this chapter has six sections. Section 3.1 presents the SLR planning. Section 3.2 explains the SLR execution. Section 3.3 discusses how we conducted the analysis. Section 3.4 reports the SLR results, answering the RQ1, RQ1.1, RQ1.2, and RQ2. Section 3.5 presents the threats to validity. Section 3.6 concludes this chapter.

3.1 Planning Stage

In this phase, we defined:

1. the topic investigated
2. the electronic databases used to search for papers
3. the search string to identify relevant studies
4. the inclusion and exclusion criteria to obtain primary studies
5. the timestamp to conduct this work.

Research Questions

To identify the relationships between refactoring and bad smells proposed by Fowler [34], we defined the following two general Research Questions (RQs), **RQ1** and **RQ2**, and two specific ones, **RQ1.1** and **RQ1.2**.

RQ1 - Which relationships between refactoring and bad smells are the literature explicitly discussing?

RQ1.1 - Which are the most mentioned relationships between bad smells and refactoring found in the literature?

RQ1.2 - Are the relationships we found different from those that Fowler presents?

RQ2 - Which tools found the papers perform refactoring from bad smell detection?

Electronic Databases

There are different electronic databases to be used in literature reviews to search for primary studies. Some studies use one database [88, 89]; others use three databases [39], six databases [4, 22, 28], seven databases [2], or even eight databases [10, 53, 83]. We used seven of the most used ones [22], as shown in Table 3.1. The first column in Table 3.1 shows the database name, while the second one presents their respective websites.

Search String

The search string identifies the relevant studies in the selected databases, allowing us to answer the proposed research questions. We conducted a pilot study with search strings composed of multiple terms and applied them to each database. We evaluated their results in each database to identify which search string has

Table 3.1: Eletronic Databases

Database	Address	Papers Returned
ACM Digital Library	`http://dl.acm.org/`	113
Engineering Village	`https://www.engineeringvillage.com`	83
IEEE Xplore	`http://ieeexplore.ieee.org/`	47
Science Direct	`http://www.sciencedirect.com/`	07
Scopus	`http://scopus.com/`	70
Springer	`http://link.springer.com/`	2,025
Web of Science	`http://apps.webofknowledge.com/`	44
Total		**2,389**

reached as many studies as possible in the literature. In the end, the search string was consolidated with three terms as follows.

```
(refactor OR refactoring) AND (relationship OR correlation OR
associate) AND (“code smell” OR “bad smell” OR bug OR “anti
pattern”)
```

Inclusion and Exclusion Criteria

We used four inclusion and two exclusion criteria to select the primary studies (see Table 3.2). These criteria allow classifying each study under review as a candidate to be included or excluded from the SLR. As an SLR may involve many studies, we limited the scope of selecting only complete papers in English and ignored short papers or documents classified as dissertations or thesis. We decided to do so because usually, they are published as full papers by the authors. Regarding short papers, they present emerging results.

Table 3.2: Inclusion and Exclusion Criteria

Inclusion Criteria	Exclusion Criteria
• Written in English • Published in conferences, journals, workshops and book chapters* • Available in electronic format • Present refactoring and bad smells defined by Fowler	• < 5 pages • Thesis, dissertations, tutorials, courses and magazines issues

*Papers published at conferences that appear as book chapters in digital libraries

Search Source

We searched all studies published up to 2018. We carried out the search process from February 6 to 9, 2019.

3.2 Execution Stage

This phase consists of (i) applying the search string in the selected databases, identifying primary studies, and (ii) selecting the relevant ones found through the exclusion/inclusion criteria. Figure 3.1 presents the resulting number of primary studies after each step, serving as input to the next step. *1.Remove Duplicates*, *2.Reading Title*, *3.Reading Abstract*, *4.Inclusion and Exclusion Criteria*, *5.Reading Introduction and Conclusion*, and *6.Complete Reading of Paper* represents these steps, discussed below. The check symbols indicate the studies remaining in the step, while numbers near cross symbols indicate the studies excluded. At the end of the process, we identified 20 papers that fit the scope of this work. These studies were analyzed and summarized in order to collect information to answer our RQs.

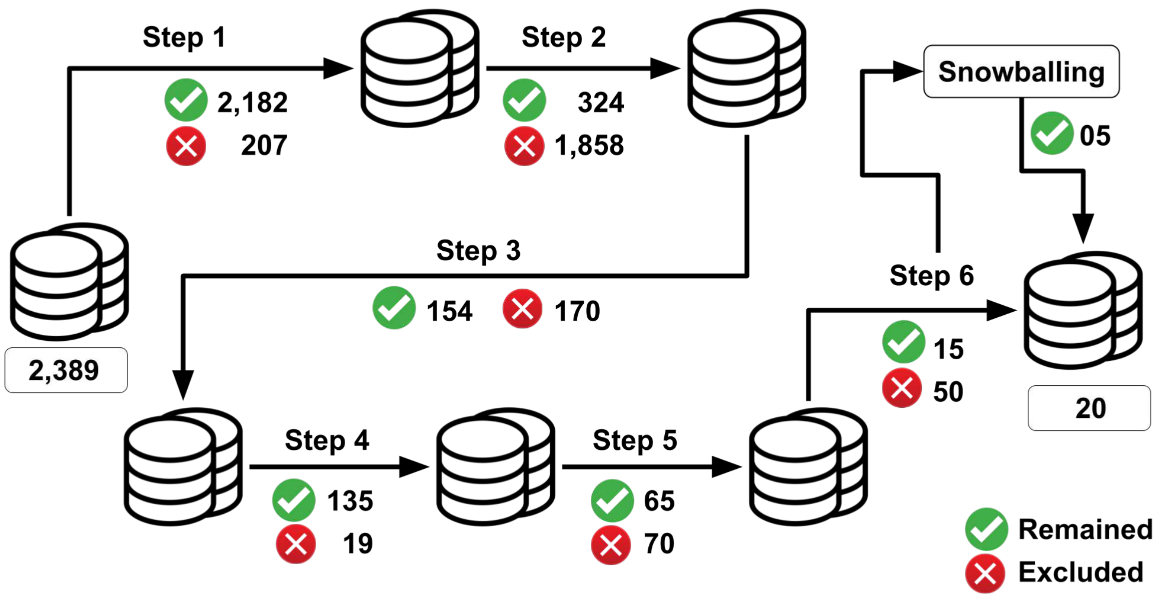

Figure 3.1: Number of Papers Filtered in Each Step

Search Process

In the search process, we did not define any constraint, and, therefore, we considered all studies returned by non-filter databases per publication year. Table 3.1 presents the primary studies returned by the search in each database, with a total of 2,389 studies, including duplicates.

Study Selection Process

Figure 3.1 presents the six steps focusing on selecting relevant papers according to their content. The result regarding each database is available on our website.[1] We discuss the six steps as follows.

Step 1 - *Remove Duplicates.* We merge the papers returned in all databases, which causes the removal of duplicated papers, and randomly eliminated the papers without favoring any database.

Step 2 - *Reading Title.* We retained only those papers with excerpts of the search string used or those that could be relevant for the study. In case of doubts, we kept the papers for the next filtering step.

Step 3 - *Reading Abstract.* Here, we selected those papers that show some evidence of being linked to the SLR context.

Step 4 - *Inclusion and Exclusion Criteria.* We applied the inclusion criteria defined in Section 3.1, if the paper fits in at least one of the exclusion criteria, we removed it from our SLR.

Step 5 - *Reading Introduction and Conclusion.* We selected those papers that present evidence to help answer our RQs and those relevant to the study.

Step 6 - *Complete Reading of Paper.* We included only those papers that are directly related to our RQs.

Snowballing - Searching in the electronic database does not guarantee that we will recuperate all relevant studies related to a particular topic. To mitigate this limitation, we did a snowballing procedure. Snowballing is a search approach that uses paper citations as a reference list to identify studies that are not found in the search process [85]. Snowballing allows performing backward or forward. Backward uses the reference list of the paper to find other studies. Forward refers to identifying new articles analyzing the studies that cite a given study. In our SLR, we execute backward snowballing. We performed this process with the 15 papers identified at the end of Step 6. Futhermore, for the papers identified, we applied again these six steps to find new candidates to be included. The snowballing has resulted in the inclusion of five new papers.

Data Extraction

Research questions are the main drivers of what information needs to be extracted. RQ's main topic was identified and summarized as tabular data. For each paper found, we documented: each bad smell mentioned in the study and

[1] `https://cleitonsilvat.github.io/dissertation/`

the refactoring operations related to it, and each tool mentioned in the paper and its characteristics.

This information allows us to identify the most studied relationship and compare the Fowler suggestions against the literature's proposed suggestions. It also provides insights into which tools are proper to study both bad smells and refactoring.

The quality assessment is an integral part of an SLR to assess the evidence's strength and consider when synthesizing the results [12]. However, we did not perform it in this SLR. The reason is that we consider that the Study Selection Process and Data Extraction were sufficient to reach the most relevant papers in our study. Besides, a good strategy to validate the relationships found would be the conduction of an empirical study. We carried out this study in our dissertation at Chapter 4 to validate, refute or complement the relationships found.

3.3 Analysis Stage

This section presents an overview of the primary studies, describing the papers' characterization found.

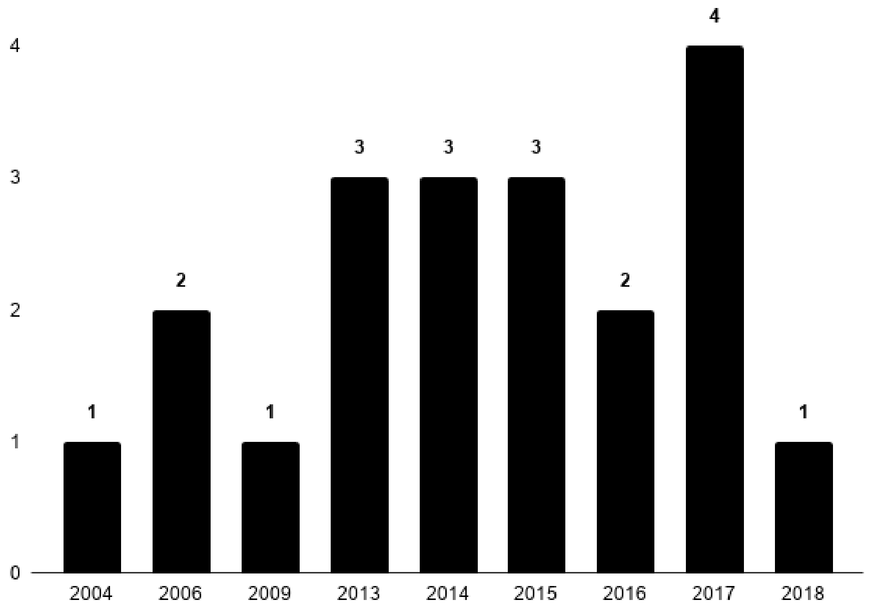

Figure 3.2: Publication Year

It is worth noting that the publication of all selected papers occurred between 2004 and 2018. Figure 3.2 shows the number of studies found per year. Observe that

2017 shows the largest number of relevant studies, and the years from 2013 to 2015 show three papers each. We did not find any study discussing the direct relationship between refactoring and bad smells in 2005, 2007, 2008, and 2010 to 2012. Furthermore, from the figure, we may argue that the topic has been recently researched.

Journals and conferences mainly published nine and eight studies from the selected ones, respectively. They published them in 17 different events, most of them in the IEEE Transactions on Software Engineering and the International Conference on Agile Software Development (XP), with 3 and 2 papers, respectively. The remaining events presented only one study, our Appendice A and website[2] exhibit them. The next section presents the SLR results and answers the research questions raised.

3.4 SLR Results

This section presents the results obtained in our SLR. Section 3.4.1 answers the RQ1 and Section 3.4.2 answers the RQ2 raised.

3.4.1 Relationships between Bad Smells and Refactoring

This section answers the research questions RQ1, RQ1.1, and RQ1.2.

RQ1 Which relationships between refactoring and bad smells are the literature explicitly discussing?

Tables 3.3, 3.4, and 3.5 present the relationships between refactoring and bad smells according to the analysis of the 20 papers found. These tables present, respectively:

(i) a perfect match between discussions presented by Fowler et al. [34] and the papers found in our SLR,
(ii) the relationships that were only discussed by Fowler et al. [34], and
(iii) other situations found about the relationships between bad smells and refactoring discussed by Fowler et al. [34] and by papers found in our SLR.

Each line presents a different relationship between a bad smell and refactoring. The first column presents the bad smell; the second column shows which refactoring related to it; columns 3 and 4 check whether the relationship is discussed by Fowler or

[2]`https://cleitonsilvat.github.io/dissertation/`

by the literature, respectively. A bullet indicates that refactoring was discussed. It is worth noticing that a relationship may be discussed by Fowler and by the literature. The last column presents a reference for the studies that discuss these relationships.

Table 3.3: Relationships Between Bad Smell and Refactoring Perfect Match

Bad Smell	**Refactoring**	Fowler	Literature	**References**
Alternative Classes with Different Interfaces	Extract Superclass	●	●	[19]
	Move Method	●	●	[19]
	Rename Method	●	●	[19]
Data Class	Encapsulate Collection	●	●	[21, 69]
	Encapsulate Field	●	●	[19, 21, 69]
	Extract Method	●	●	[21, 69]
	Hide Method	●	●	[19, 21, 69]
	Move Method	●	●	[19, 21, 69]
	Remove Setting Method	●	●	[69]
Refused Bequest	Push Down Field	●	●	[7, 19]
	Push Down Method	●	●	[7, 15, 19]
	Replace Inheritance with Delegation	●	●	[7]

Table 3.4: Relationships Between Bad Smell and Refactoring Discussed Only by Fowler

Bad Smell	**Refactoring**	Fowler	Literature	**References**
Comments	Extract Method	●		
	Introduce Assertion	●		
	Rename Method	●		
Data Clumps	Extract Class	●		
	Introduce Parameter Object	●		
	Preserve Whole Object	●		
Long Parameter List	Replace Parameter with Explicit Methods	●		
Middle Man	Inline Method	●		
	Remove Middle Man	●		
	Replace Delegation with Inheritance	●		
Primitive Obsession	Extract Class	●		
	Introduce Parameter Object	●		
	Replace Data Value with Object	●		
	Replace Type Code with Class	●		
	Replace Type Code with Subclass	●		
	Replace Type Code with State/Strategy	●		
	Replace Array with Object	●		
Temporary Field	Extract Class	●		
	Introduce Null Object	●		

We focused on bad smells and refactorings proposed by Fowler. However, we have merged different terminologies into a single bad smell for specific situations since their meaning and characteristics are similar, only changing their nomenclature. We classified Clone [50] or Code Clone [41, 52] as Duplicated Code, and Parallel Inheritance [63, 69] as Parallel Inheritance Hierarchies. There is a case where the authors explicitly indicated State Checking as the Switch Statements smell [18]. For the refactoring classification, we have the case of Polymorphism [18] named for Replace Conditional with Polymorphism.

Every paper we have identified discusses more than one relationship between bad smells and refactoring. Therefore, we individually documented each relationship found. Observe that there are cases where different refactoring may be applied to address the same bad smell. To answer RQ1, Tables 3.3, 3.4, and 3.5 present the 22 bad smells proposed by Fowler, which 16 of them (73%), exhibited in Tables 3.3 and 3.5, are also studied in the literature. They indicate the concern of researchers to address most bad smells present in the catalog. However, the situation for refactoring is different. From a total of 72 refactorings in the catalog, the literature cited only 31 (43%). Besides, this shows that not all refactoring operations mentioned by Fowler are discussed in the literature, relating it to bad smells.

RQ1.1 Which are the most mentioned relationships between bad smells and refactoring found in the literature?

Tables 3.3 and 3.5 show all relationships between refactoring and bad smells found in this SLR. We do not consider the results described in Table 3.4 because it does not present relationships discussed in the literature, only those presented by Fowler et al. [34]. To answer this RQ, we consider the relationship discussed by the largest number of different studies. We identified that the highest number of citations is related to Move Method (refactoring) and Feature Envy (bad smell), totaling 14 different papers targeting this relationship, as exhibited in Table 3.5.

According to the data presented in Tables 3.3 and 3.5, we also observed that the refactoring operation that relates to the most considerable amount of bad smells is Move Method, which relates to 11 types of smells, while Long Method is the bad smell related with the highest number of operations; 13 refactorings relate to it.

RQ1.2 Are the relationships we found different from those that Fowler presents?

To answer this question, we performed an analysis based on columns 3 and 4 of Tables 3.3, 3.4, and 3.5. We identified three cases:

Case 1: if Column 3 is marked and Column 4 is not, the relationship is discussed only by Fowler, meaning that the relationship needs to be researched, or maybe it is not discussed in the literature (see Table 3.4).

Table 3.5: Relationships Between Bad Smell and Refactoring

Bad Smell	**Refactoring**	Fowler	Liter.	**References**
Divergent Change	Extract Class	●	●	[16, 63]
	Extract Method		●	[15]
	Extract Superclass		●	[16]
	Move Field		●	[16]
	Move Method		●	[16]
	Pull Up Method		●	[16]
Duplicated Code	Extract Method	●	●	[32, 41, 50, 52]
	Form Template Method		●	[32]
	Pull Up Method	●	●	[19, 32, 41]
	Replace Met. with Met. Obj.		●	[32]
	Substitute Algorithm	●		
Feature Envy	Consolidate Dup. Cond. Frag.		●	[7]
	Extract Method	●	●	[7, 15, 21, 69, 81]
	Move Field		●	[18, 21, 81]
	Move Method	●	●	[7, 8, 9, 18, 19, 21, 48, 52] [63, 65, 66, 67, 69, 81]
	Pull Up Method		●	[21]
Inappropriate Intimacy	Change Bidir. Assoc. to Unidir.	●		
	Extract Class	●		
	Hide Method	●		
	Move Field	●	●	[19]
	Move Method	●	●	[19]
Incomplete Library Class	Introduce Foreign Method	●		
	Introduce Local Extension	●		
	Move Method	●	●	[19]
Large Class	Duplicate Observed Data	●		
	Extract Class	●		
	Extract Subclass	●	●	[19]
Lazy Class	Collapse Hierarchy	●		
	Inline Class	●	●	[7]
	Move Method		●	[65]
	Push Down Method		●	[15]
Long Method	Add Parameter		●	[7]
	Consolidate Cond. Expr.		●	[7, 21]
	Decompose Conditional	●	●	[21]
	Extract Method	●	●	[7, 16, 18, 21, 38, 50]
	Inline Method		●	[7]
	Introduce Explaining Variable		●	[7]
	Introduce Parameter Object	●	●	[21]
	Preserve Whole Object	●	●	[21]
	Remove Control Flag		●	[7]
	Remove Parameter		●	[7]
	Rename Method		●	[7]
	Replace Met. with Met. Obj.	●	●	[21]
	Replace Temp With Query	●	●	[21]
Message Chains	Extract Method	●		
	Hide Delegate	●		
	Move Method	●	●	[19]
Parallel Inheritance Hierarchies	Extract Subclass		●	[63]
	Move Field	●	●	[19]
	Move Method	●	●	[19, 69]
Shotgun Surgery	Move Class	●		
	Move Field		●	[16, 19, 63]
	Move Method	●	●	[16, 19, 63]
	Pull Up Method		●	[16]
Speculative Generality	Collapse Hierarchy	●	●	[7]
	Inline Class	●		
	Remove Parameter	●	●	[19]
	Rename Method		●	[19]
Switch Statement	Extract Method	●		
	Move Method	●	●	[19]
	Rep. Cond. with Polymorphism		●	[18]
	Rep. Type Code with Sta./Strat.	●		
	Rep. Type Code with Subclass	●		

Case 2: if the opposite situation occurs, where Column 4 is marked, and Column 3 is not, it means that we found a new relationship not addressed in Fowler's catalog. Some situations in Table 3.5 shows it.

Case 3: if columns 3 and 4 are marked, this represents that the relationship found in the literature agrees with what Fowler says. Table 3.3 exhibits it.

For example, in Table 3.5 we identified for Lazy Class all refactoring operations related to it. We found a refactoring that only Fowler presents a relationship: Collapse Hierarchy (Case 1). We also found that the literature proposes two new refactoring operations related to it: Move Method and Push Down Method (Case 2). Finally, we identified Inline Class in Fowler's work and in the literature (Case 3).

From Table 3.3 and by the reasoning explained above, we may conclude that only Alternative Classes with Different Interfaces, Data Class, and Refused Bequest have both columns 3 and 4 being a perfect match, which indicates that there is no new refactoring strategy related to them reported in the literature. Also, the literature did not address six bad smells from the Fowler's catalog (see Table 3.4), which may be due to their nature, since some bad smells do not significantly impact the source quality [62]. Observe that for seven of the 22 smells presented in Table 3.5, the literature has proposed more refactoring operations than Fowler. However, for these smells, Fowler has presented 18 refactoring strategies related to them, of which literature did not confirm only three. In total, we have identified 35 relationships in Case 1, 24 in Case 2, and 35 in Case 3.

3.4.2 Tools for Refactoring

This section answers the research question RQ2.

RQ2 Which tools found in papers perform refactoring from bad smell detection?

It is worth noticing that to answer the **RQ2** we do not search all proposed or mentioned tools present in the literature. This SLR focuses on identifying studies that explicitly cite the relationship between refactoring and bad smells, not identifying all refactoring tools. Therefore, we documented only tools that appear in the resulting studies, which detect bad smells, and propose refactoring to solve them. Out of 20 studies, 16 have mentioned the tools used to conduct them. Table 3.6 exhibits the tools composed of both concepts and information about each one, such as if it has a Graphical User Interface, or it is a Framework. The complete list of tools may be found on our Appendice A and website.[3] We observe that nine tools need more than one tool

[3]`https://cleitonsilvat.github.io/dissertation/`

to perform the process of detecting bad smells and applying the refactoring. As our study's focus is to tackle the refactoring and bad smell context, we only report seven tools that target both concepts in the same tool.

Table 3.6: Characteristics of Each Tool

Tool [Ref.]	**GUI**	**FRA**	**ONL**	**PLG**	**FRE**	**OPS**	**USG**	**SL**
Extract Method Detector [50]	Yes	-	No	Yes	Yes	Yes	No	Java
JDeodorant [8, 18, 69]	Yes	-	No	Yes	Yes	Yes	Yes	Java
JMove [67]	Yes	-	-	Yes	Yes	Yes	Yes	-
Liu's Approach [48]	-	-	-	-	-	-	-	-
Methodbook [9]	-	-	-	-	-	-	-	Java
MMRUC3 [66]	-	Yes	-	-	-	-	-	Java
Tsantalis's Methodology [81]	-	-	-	-	-	-	-	Java

GUI: graphical user interface; **FRA**: framework; **ONL**: online; **PLG**: plugin; **FRE**: free for use; **OPS**: open-source; **USG**: user guide available; **SL**: supported language; **"-"**: information not available.

3.5 Threats to Validity

Even with careful planning, definition, and application of our design, some threats to validity may invalidate our findings. We discuss the threats to validity listed by Wohlin et al. [86]: (i) construct validity, which concerns the relationship between which the experiment setting reflects the theory; (ii) internal validity, that may affect the independent variable concerning causality; (iii) conclusion validity, which may affect the ability to draw the correct conclusion; and (iv) external validity, that limits the ability to generalize the results beyond the experiment setting.

Construct Validity

The electronic database selected might not retrieve all relevant papers. To minimize this threat, we selected seven databases that aggregate papers from many publishers. Also, the search string used may not find all papers that are relevant to this SLR. To minimize this threat, we designed a search string that includes standard terms for *"refactoring"* and *"bad smell"*. Furthermore, we performed a pilot search in each database to select a subset of the most common terms used in the research field, to retrieve the highest number of relevant papers. However, we may not assume that this filtering strategy found all existing related works.

Internal Validity

This study's guaranteed reproducibility is due to the detailed specification of the search engines used, the search string, and the inclusion and exclusion criteria. Possible limitations of the search results were overcome by including different terms used by different authors but with similar concepts, staying in this SLR scope. Another internal validity threat may be related to the judgment of the information presented, which expresses only the researchers' point of view. The researchers have carried out the selection stage to minimize this threat, which in the initial stages made comparisons of the selected papers to avoid bias in selecting studies. Besides that, frequent meetings were held between three researchers to discuss the relevant papers.

Conclusion Validity

The data summaries found in the literature and the Fowler's catalog present the researchers' point of view, which may not present the papers' actual concept. To minimize this conclusion threat and maintain the information's integrity, we documented what was explicitly presented by the papers.

External Validity

This threat is related to the selected papers' representativeness published up to 2018 regarding the systematic literature review's primary goals. We used the systematic protocol to support a comprehensive representation of the selected papers, but some other papers may have been published after 2018 or indexed after applying the search string in the databases. Our findings of the relationship between refactoring and bad smells in the study period are accurate to the best of our knowledge. Furthermore, we focus on the presentation of the bad smells and refactorings presented in the first edition of Fowler's book [34]. As the second edition's book was presented in 2018 [33], we believe that a period to be sought after that date may be related to this second edition, which is not the focus of our study.

3.6 Final Remarks

Refactoring from bad smell detection is not profoundly discussed in the literature. This chapter presented the result of an SLR to identify the direct relationship between refactoring and bad smells. We have identified 20 papers that show the direct relationship between 31 different refactoring and 16 bad smells. Analyzing these papers, we have found that:

(i) the relationship between Move Method applied to Feature Envy appears as the most discussed one in these studies;

(ii) there are 24 different relationships between refactoring related to some bad smells than those proposed by Fowler;

(iii) seven tools refactor through the identification of the bad smells proposed by Fowler.

According to the results presented in this chapter, we identified some relationships that Fowler did not initially discuss. These new relationships found may represent situations that were not previously foreseen, patterns of certain development groups, or some other situation. To identify and validate these possible situations, we conducted a deep study in this context, via an empirical study aiming to analyze the impact of refactoring on bad smells, that may validate or refute the new relationships found in our SLR. We describe it in Chapter 4.

Chapter 4

Empirical Study Design

A software system requires much effort from developers to maintain its source code [22]. To continue meeting its requirements and evolve or adapt to new technologies, it must follow acceptable development practices to facilitate the team's work. Thus, being refactoring the code is expected to decrease the number of bad smells, which are poor code implementations that indicate the need to be refactored [37]. However, some studies show that the refactoring process often does not solve all source code's bad smells [7, 43]. Furthermore, other studies show evidence that the refactoring process may not only remove bad smells in the source code but also may introducing new ones [54, 65, 82]. Such a problem may lead the developer not to trust automated refactoring since there are high chances of not completely removing the bad smells present in the code.

This chapter defines an empirical research design to assess the automated refactoring process's impacts in detecting bad smells to address the mentioned problem. To assess this impact, we selected eight open-source Java systems available at Qualitas Corpus[80]. We analyzed the impacts with the assistance of five tools (see Section 4.2), using five refactorings, except for JDeodorant; this happens because refactorings of the type Replace Type Code with State/Strategy and Replace Conditional with Polymorphism [34] are treated together by JDeodorant. We refer to them for documentation criteria by Replace Refactoring. Therefore, given this union, from Section 4.1 to the end of this dissertation, we treat the refactorings operation analyzed as being of four types. We analyzed these refactoring impacts on ten bad smells proposed by Fowler et al. [34].

We organized the remainder of this chapter as follows. Section 4.1 describes this study's goal and the research questions used to conduct it. Section 4.2 discussess the research phases adopted to construct our datasets of systems, tools, bad smells, and refactorings to be analyzed. Section 4.3 describes the steps used to the assessment of the impacts. Finally, Section 4.4 concludes this chapter.

4.1 Goal and Research Questions

Our goal in this study is to evaluate the impact of refactoring operations on detecting bad smells in open-source Java systems. To conduct this study, we address four refactoring and ten bad smells proposed by Fowler et al. [34], discussed mainly in the literature [47]. The research questions (RQs) to be analyzed are defined as follows.

RQ3 - What are the impacts of automated refactoring on the detection of bad smells?

RQ3.1 - Does the automated refactoring process remove bad smells?

RQ3.2 - Does the automated refactoring process introduce bad smells?

To answer **RQ3**, we used an automated refactoring tool that created a new refactored version from each system's original version. We used bad smell detection tools in the original version and also in the refactored version. With the detection of bad smells carried out in the original and refactored versions, it was possible to assess the refactoring operation's impact on the bad smells detections, making it possible to answer **RQ3.1** and **RQ3.2.**

4.2 Research Phases

To assess the impacts caused by the refactoring operations, we initially defined three phases to compose the data to be analyzed. Figure 4.1 shows these phases followed by the description of each one.

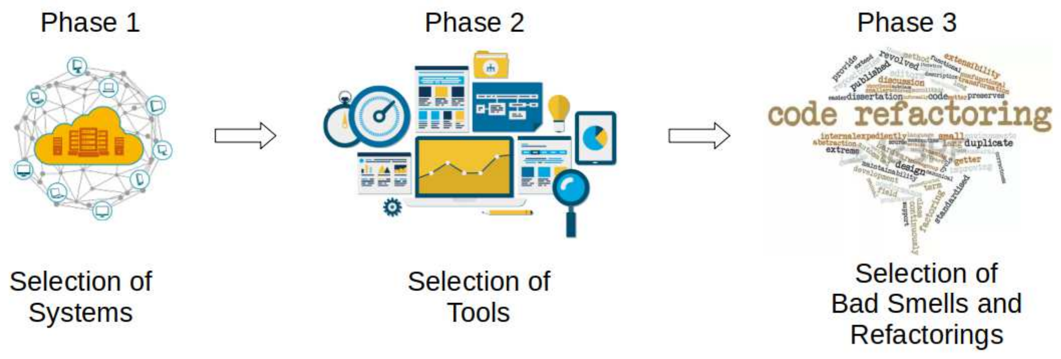

Figure 4.1: Preparation Phases

Phase 1 - Selection of Systems

To conduct our research, we initially selected a set of eight systems to compose the study sample. We prioritize systems available through the Qualitas Corpus[1] [80], which contains a curated collection of open-source Java software systems to be used for empirical studies. Table 4.1 presents the selected systems. The first column presents the abbreviation of each system. The second column presents the name of the systems. Finally, the last six columns present some metrics which demonstrate the heterogeneity of the selected systems. The calculated metrics were: Number of Classes *(NOC)*, Total Lines of Code *(TLOC)*, Weighted Method per Class *(WMC)*, Number of Methods *(NOM)*, Number of Packages *(NOP)*, and Method Lines of Code *(MLOC)*. These metrics were calculated with the assistance of Metrics[2], an Eclipse plugin used for static code analysis.

Table 4.1: Selected Systems

Abbreviation	System	NOC	TLOC	WMC	NOM	NOP	MLOC
S1	Checkstyle-5.6	492	36,641	5,164	2,665	42	21,070
S2	Commons-codec	136	20,400	2,757	1,262	13	12,314
S3	Commons-io	276	30,371	4,761	2,274	18	23,110
S4	Commons-lang	520	75,622	11,456	5,099	27	68,124
S5	Commons-logging	73	5,449	1,008	464	19	3,608
S6	JHotDraw-7.5.1	671	79,668	14,122	5,892	66	52,852
S7	Quartz-1.8.3	232	28,557	5,294	2,343	51	20,161
S8	Squirrel_sql-3.1.2	56	6,944	930	532	3	4,316

Phase 2 - Selection of Tools

This research uses some tools to automate the processes of refactoring and detecting bad smells. We selected five tools for bad smell detection: Decor, Designite, JDeodorant, SJpIRIT, and Organic. JDeodorant was the only one used to assist the refactoring process, besides performing bad smell detection, also supports automated refactoring.

Phase 3 - Selection of Bad Smells and Refactorings

To carry out our research, we focused on a sample composed of ten bad smells and four refactorings available in the Fowler's catalog [34]. These sample of bad smells and refactorings were chosen because there are already tools supporting them. We choose this catalog because it is the most completed one. Besides, the

[1]`http://qualitascorpus.com/`

[2]`http://metrics.sourceforge.net/`

literature widely discusses it. Table 4.2 presents the list of the ten types of bad smells evaluated and the tools we used to perform their detections.

Table 4.2: Bad Smell Detections by Tools

Abbreviation	Bad Smell	Decor	Designite	JDeodorant	JSpIRIT	Organic
DC	Data Class				●	●
FE	Feature Envy			●	●	
LC	Large Class	●				
ZC	Lazy Class	●				●
LM	Long Method	●	●	●		
LP	Long Parameter List	●	●			
MC	Message Chains	●				
RB	Refused Bequest	●			●	●
SS	Shotgun Surgery				●	
SG	Speculative Generality	●				●
	Total	**7**	**2**	**2**	**4**	**4**

It is worth mentioning that for Refused Bequest, we also consider Refused Parent Bequest detections provided by the tools. We have not used the Fowler's bad smell Duplicated Code offered for JDeodorant because it requires the addition of another tool, and we decided to use only the results of the tool itself. Moreover, the five tools used to detect bad smells do not focus only on making the bad smell detections proposed by Fowler. Considering this study focuses on the detections of bad smells proposed by Fowler, we documented only them.

As said in the introduction of this chapter, we refer to Replace Type Code with State/Strategy and Replace Conditional with Polymorphism by Replace Refactoring in JDeodorant. Therefore, given this union, we treat the refactorings operation analyzed as Extract Class, Extract Method, Move Method, and Replace Refactoring. JDeodorant supported all of them.

4.3 Impacts Assessment

We choose to create a refactored version of each system with the support of an automated refactoring tool. The evaluated systems represent a sample of systems from Qualitas Corpus. The tools used offer automated assistance to carry out the refactoring and bad smell detection operations. Figure 4.2 shows the seven steps taken to assess the automated refactoring impacts described as follows.

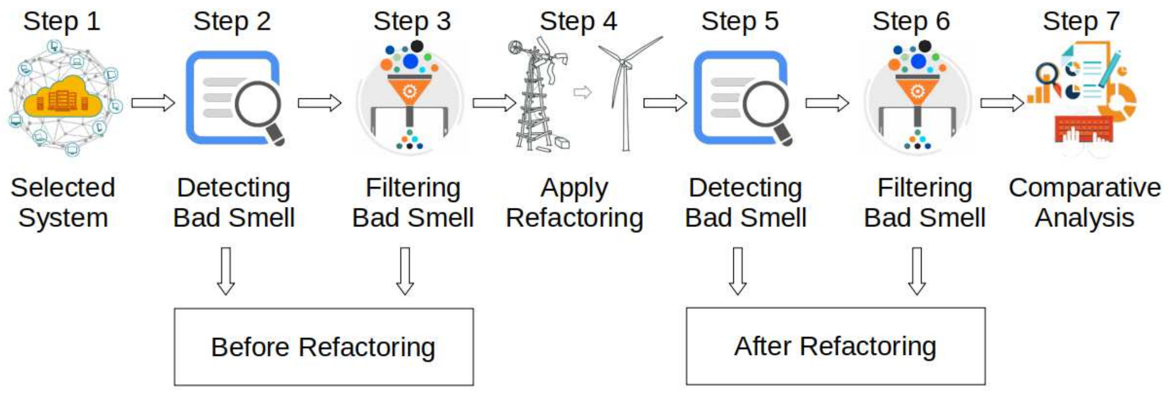

Figure 4.2: Steps of This Empirical Study

Step 1 selects the system for evaluation. Before Refactoring, in Step 2, we compute the results identified by the five bad smell detection tools in the original version of the system. In Step 3, we filter the results returned by these bad smell detection tools to find only the ones that are the focus of this study. It is worth remembering that none of the five tools used supports detecting the ten bad smells mentioned. Some tools detected only two bad smells, and others detected up to seven of the analyzed bad smells.

Step 4 selects a Refactoring Strategy to be applied. For this step, aiming to use the entirety tool, we carry out all the refactoring suggestions provided by JDeodorant. After Refactoring, Step 5 computes the results identified by the five bad smell detection tools in the refactored version of the system. In Step 6, we filter the results returned by these bad smell detection tools to find only the ten bad smells that occur in this study.

Step 7 is responsible for performing a comparative analysis of the results obtained from steps 3 and 6, where these two steps represent the ten bad smells detected in the original and refactored versions of the system, respectively. Comparing the results allows us to analyze the impacts caused by the automated refactoring operation, for instance, if this type of refactoring removes bad smells or introduces new ones. Moreover, it allows us to list the bad smells removed and show any new smells by a particular refactoring strategy.

Considering that the agreement rate between the bad smells detection tools is only high in situations of true negative [61], i.e., entities categorized as not containing any smell. According to the precision and recall of these tools, we may conclude that the false positive rate remains high. Thus, we decide to conduct our analyzes initially based on four perspectives (see Section 5.1) to enable us to answer our research questions. Besides, we also conduct two complementary analyzes: a) the aggregate analysis, and

b) the comparative analysis. In the first case, we conduct an analysis considering the total value of detections made by the tools, as will be described in Section 5.5. In the second case, we present a general overview considering initially the four perspectives used in our empirical study comparing with the findings presented in our SLR, as presented in Chapter 3 (see Section 5.6).

Algorithm 1: How steps work

```
Result: Comparative analysis
Data: 8 systems, 4 refactoring types, 5 bad smell tools
foreach system in database do
    list v1 = DetectBadSmell(system);
    refactored = ApplyRefactoring(system);
    list v2 = DetectBadSmell(refactored);
    perform comparative analysis with v1 and v2;
end
Function DetectBadSmell(system):
    foreach bad smell tool do
        detect list of bad smells;
        foreach list of bad smells do
            select only Fowlers' bad smells
        end
        return list of Fowlers' bad smells;
    end
Function ApplyRefactoring(system):
    foreach type of refactoring do
        while exist suggestion of refactoring do
            if refactoring dont generate error then
                apply refactoring;
            end
        end
        return system refactored;
    end
```

Algorithm 1 presents a pseudo-algorithm demonstration of the execution carried out. Step 1 performs eight times, which represents the eight systems used in this research. For each execution of Step 1, steps 2 and 3 performed five executions, representing the identification provided by the five tools used to detect the bad smells on the original system. For each execution of Step 1, Step 4 performed four executions, representing the four refactoring strategies adopted. In this step, the number of refactorings applied fluctuates according to the suggestions provided by JDeodorant, except for some exceptions mentioned in Chapter 5.

For each execution of Step 4, steps 5 and 6 performed five executions. Steps 5 and 6 represent the identification of the five tools used to detect the bad smells on the refactored system. For Step 7, we carry out the comparative analysis for each applied refactoring taking into account the detection of bad smells performed by each tool in the original and refactored version of the system.

4.4 Final Remarks

This chapter presented the empirical research design to analyze the impact of refactoring on bad smells. To conduct this research, we selected eight open-source Java systems available in the Qualitas Corpus. We selected four refactorings to be applied and measured their impact on ten different bad smells detected by five different tools. Chapter 5 presents the analyses and results of the empirical study described here in this chapter.

Chapter 5

Empirical Study Analyses and Results

Considering the experimental design presented in Chapter 4, we describe in this chapter the analyses and results found in our experiment. We answer the research questions based on our results and present some discussions about the analyses performed. We applied a total of 668 refactorings providing by JDeodorant. After analyzing the results, we found that the refactoring process impacts different types of bad smells. As expected, all types of refactoring identified cases in which there was a bad smell solution. Surprisingly, however, we also found cases where refactoring may introduce new bad smells.

We organize the remainder of this chapter as follows. Section 5.1 describes the detections made and used as a basis for conducting the analyzes. Section 5.2 presents the refactorings applied, distributed in the eight systems. Section 5.3 presents the comparative analyses conducted. Section 5.4 describes a summary of the results and presents the raised research questions' answers. Section 5.5 presents in higher-level the analyses of the found results. Section 5.6 discusses and compares our empirical study results with our systematic literature review results described in Chapter 3. Section 5.7 presents the threats to validity. Finally, Section 5.8 concludes this chapter.

5.1 Primary Result

We analyzed the impact caused by four types of refactoring on ten types of bad smells evaluated in eight Qualitas Corpus systems using four perspectives to carry out this comparative analysis. We describe the four perspectives analyzed in our empirical study as follows.

(i) Original Detection. We consider the detections made by the bad smells detection tools as valid without making any changes, union, intersection or merge in this analysis's returned data, e.g., the first detection.

(ii) Standardized Detection. In this analysis, we consider the results provided by the tools as valid, but we perform a standardization in return made by the tools, evaluating the common detections between them, e.g., for the same bad smell detected, one tool presented the detection up to the class level and another tool presented the detection up to the method level, for this situation the detections for this type of smell were standardized considering up to the class level for both tools.

Table 5.1: Bad Smells Detected Before Refactoring

(a) Original Detection

Bad Smell	S1	S2	S3	S4	S5	S6	S7	S8
DC[T4 | T5]	| 25	3 |	| 2	22 | 4	1 | 2	22 | 52	5 | 10	| 8
FE[T3 | T4]	18 | 275	15 | 102	3 |	1 | 708	| 38	13 | 528	7 | 195	7 | 47
LC[T1]								
ZC[T1 | T5]	3 | 260	4 | 36	| 63	| 153	1 | 18	30 | 176	4 | 67	1 | 17
LM[T1 | T2 | T3]	123 | 4 | 243	27 | 13 | 117	| 6 | 74	| 10 |	19 | 1 | 53	126 | 37 |	45 | 11 |	12 | 1 | 99
LP[T1 | T2]	12 | 32	6 | 9	| 29	| 94		70 | 155	17 | 56	1 | 10
MC[T1]		1				40		1
RB[T1 | T4 | T5]	52 | 51 | 18	| 11 | 4	| | 4	| 6 | 15	6 | 3 | 5	146 | 100 | 16	21 | 18 | 1	
SS[T4]	39	1		20	9	167	49	2
SG[T1 | T5]	2 | 39	| 5	| 7	| 13	1 | 2	9 | 24	3 | 6	

(b) Standardized Detection

Bad Smell	S1	S2	S3	S4	S5	S6	S7	S8
DC[T4 | T5]	| 25	3 |	| 2	22 | 4	1 | 2	22 | 52	5 | 10	| 8
FE[T3 | T4]	18 | 274	15 | 102	3 |	1 | 639	| 36	13 | 512	7 | 191	7 | 47
LC[T1]								
ZC[T1 | T5]	3 | 250	4 | 36	| 62	| 139	1 | 18	30 | 176	4 | 67	1 | 17
LM[T1 | T2 | T3]	123 | 4 | 104	27 | 13 | 59	| 6 | 33	| 10 |	19 | 1 | 42	126 | 36 |	45 | 11 |	12 | 1 | 42
LP[T1 | T2]	12 | 22	6 | 8	| 12	| 38		70 | 73	17 | 23	1 | 8
MC[T1]		1				40		1
RB[T1 | T4 | T5]	52 | 51 | 18	| 11 | 4	| | 4	| 6 | 15	6 | 2 | 5	145 | 100 | 16	21 | 18 | 1	
SS[T4]	37	1		12	9	164	48	2
SG[T1 | T5]	2 | 39	| 5	| 7	| 13	1 | 2	9 | 24	3 | 6	

S1: Checkstyle-5.6, S2: Commons-codec, S3: Commons-io, S4: Commons-lang, S5: Commons-logging, S6: JHotDraw-7.5.1, S7: Quartz-1.8.3, S8: Squirrel_sql-3.1.2, T1: Decor, T2: Designite, T3: JDeodorant, T4: JSpIRIT, and T5: Organic

For the last two perspectives, we developed a voting mechanism considering that each bad smell detected by a tool consists of a vote. To differentiate the detections made by tools, we consider two pieces of information. First, the vote is unique and represents each type of bad smell detected. Second, the level is regarding the tool that detected a type of bad smell. Therefore, we analyze it at two different levels.

- **(iii) Vote Level 1.** This level consists of analyzing the bad smells detected by at least one tool.
- **(iv) Vote Level 2.** This level consists of analyzing the bad smells that at least two tools detected.

Table 5.2: Bad Smells Detected Before Refactoring - Vote Level

(a) Vote Level 1 Detection

Bad Smell	S1	S2	S3	S4	S5	S6	S7	S8	**Total**
Data Class	25	3	2	26	3	72	15	8	**154**
Feature Envy	289	113	3	639	36	522	197	53	**1,852**
Large Class									
Lazy Class	250	37	62	139	18	189	67	18	**780**
Long Method	195	90	39	10	52	160	56	50	**652**
Long Parameter List	23	8	12	38		73	23	8	**185**
Message Chains		1				40		1	**42**
Refused Bequest	119	15	4	21	8	251	39		**457**
Shotgun Surgery	37	1		12	9	164	48	2	**273**
Speculative Generality	39	5	7	13	2	24	6		**96**
Total	**977**	**273**	**129**	**898**	**128**	**1,495**	**451**	**140**	**4,491**

(b) Vote Level 2 Detection

Bad Smell	S1	S2	S3	S4	S5	S6	S7	S8	**Total**
Data Class						2			**2**
Feature Envy	3	4		1		3	1	1	**13**
Large Class									
Lazy Class	3	3			1	17	4		**28**
Long Method	36	9			10	2		5	**62**
Long Parameter List	11	6				70	17	1	**105**
Message Chains									
Refused Bequest	2				5	10	1		**18**
Shotgun Surgery									
Speculative Generality	2				1	9	3		**15**
Total	**57**	**22**		**1**	**17**	**113**	**26**	**7**	**243**

S1: Checkstyle-5.6, S2: Commons-codec, S3: Commons-io, S4: Commons-lang, S5: Commons-logging, S6: JHotDraw-7.5.1, S7: Quartz-1.8.3, S8: Squirrel_sql-3.1.2, T1: Decor, T2: Designite, T3: JDeodorant, T4: JSpIRIT, and T5: Organic

Tables 5.1 and 5.2 present the ten bad smells and the five tools used to detect them in each system's original version, considering the four perspectives. The first column of the tables shows the names of the bad smells analyzed and for tables 5.1a and 5.1b the tools used to detect them. The next eight columns show the detection of bad smells for each system analyzed. Tables 5.2a and 5.2b present in the last column and line the total of bad smell detected for each type and system, respectively.

We may observe in tables 5.1 and 5.2 several entries with no values, which may represent three different cases presented by the five tools, discussed in the sequence. There is no case of those analyzed in which two or more tools detected the same number of bad smells, except in the non-detection of them.

1º Case: Four situations with no values.

In the first situation, represented by Decor, the detection is performed, and it did not present any bad smells in the analyzed systems. In the second situation, represented by JDeodorant, the tool's detection did not present results for a bad smell. In this research, we consider that this situation might represent no bad smell detected in the system or an internal error of the tool that failed to complete the search process. We documented both situations the same because the log system generated by JDeodorant is not easily accessible to the end-user; and we decided not to investigate the reason for this occurrence further. The third situation returned a complete list of all bad smells detected. Represent this situation: Designite, JSpIRIT, and Organic tools. We consider that the absence of a specific bad smell represents the non-existence of it. Finally, in the fourth situation, representing exclusively cases existing in Table 5.2b, exhibits the situation in which at least two tools did not detect the same bad smell.

2º Case: Same bad smell detection.

In the analysis of the original and standardized detection, as presented in Table 5.1, we decided not to compute the intersection of the detections made by the tools for two reasons. The first one was that two or more tools no detected the same number of bad smells, except in the non-detection. The second was due to the fluctuation in the number of bad smells of the same type detected by different tools. For this reason, we analyzed all data individually, not prioritizing any tool used or compromising the results provided by them.

3º Case: Non-detection of bad smell.

In some situations, the same tool does not detect a specific bad smell in more than one system. However, we kept bad smells in our analysis because the same tool detects smells in other systems. In the specific case of Large Class and Message Chains in which only Decor detects these bad smells, we decided to document it to identify if these bad smells were introduced after the refactoring operation performed.

We provide a website[1] containing: (i) all systems analyzed; (ii) all data detected by the refactoring tools; (iii) all versions of the refactored systems; (iv) notes of particular cases; and (v) all analyses carried out to assess the impacts caused by the applied refactorings.

5.2 Impacts of Refactorings

After the detection of the bad smells, our focus was on the refactoring process. The refactoring performed automatically was based on suggestions provided by JDeodorant. JDeodorant suggests six different refactorings, namely: Extract Clone, Extract Class, Extract Method, Move Method, Replace Type Code with State/Strategy, and Replace Conditional with Polymorphism refactoring.

We carried out these refactorings focusing on solving Duplicated Code, God Class, Long Method, Feature Envy, and the last two Type Checking. The tool's first type of refactoring was not analyzed in our research because it requires an additional tool or strategy to detect Duplicated Code. The last two refactorings were considered together and named Replace Refactoring.

Table 5.3: Refactorings Applied

System	Extract Class	Extract Method	Move Method	Replace Refactoring	**Total**
Checkstyle-5.6	56	149	18	5	**228**
Commons-codec	10	77	15		**102**
Commons-io	5	45	3	2	**55**
Commons-lang	31	*			**31**
Commons-logging	4	44	*	1	**49**
JHotDraw-7.5.1	58	*	13	*	**71**
Quartz-1.8.3	26	*	6	11	**43**
Squirrel_sql-3.1.2	18	65	6	*	**89**
Total	**208**	**380**	**61**	**19**	**668**

Table 5.3 shows the number of refactorings carried out for each system. The first column represents the systems used in our research. The next four columns represent

[1]`https://cleitonsilvat.github.io/dissertation/`

the four types of refactoring performed. Furthermore, the last column shows the total number of refactorings performed for each system. Each line represents the data from an analyzed system, and the last line represents the total number of refactorings performed for each type of bad smell.

Some situations in Table 5.3 represent the no refactoring application for a given type, the empty entry, and the * case. The empty entry represents situations where JDeodorant provided suggestions for refactoring, but it could not perform it. For instance, in the cases found when it tries to perform the refactoring, the Eclipse IDE returned an error, making it impossible to perform it. The other case was when the Eclipse IDE showed compilation errors after the refactoring was applied. We did not perform the refactoring when it was not trivial to solve the compilation errors through suggestions from the Eclipse IDE itself.

The * represents situations in which JDeodorant did not present any suggestions for refactoring the system. The possible cause of this situation is that there is no existing suggestion for refactoring the system for that type. Alternatively, there may have been an internal error in the tool that made it impossible to find refactoring possibilities. As the log system generated by the tool is not easily accessible to the end-user, we decided not to investigate the reason for this occurrence.

In a brief summarization of the data, we identify 668 refactorings carried out, where Replace Refactoring presented the lowest number of refactorings, and Extract Method was the most applied refactoring with 19 and 380 refactorings, respectively. The Commons-lang system had the lowest number of refactorings, and Checkstyle-5.6 had the highest, respectively, a total of 31 and 228 of the four types of refactorings that were applied.

The refactoring process was always performed from the original version of the system, thus generating the refactored version. We conducted all analyses separately and compared the data according to the detections made in the original version.

Tables 5.4, 5.5, 5.6 and 5.7 present the ten bad smells detected using the five tools to detect them after conducting the refactoring operation in each system. The first column shows the names of the bad smells analyzed and the tools used to detect them. The next eight columns show the systems analyzed, where each one may contain between three, two, one or none entry separated by "|" containing the percentage of increase or decrease bad smell detected. In a general analysis that includes the combination of the ten types of bad smells and the five detection tools performed in the eight systems used in this research, we have a total of 152 situations to be analyzed in tables 5.4 and 5.5. Tables 5.6 and 5.7 present the Vote Level 1 Bad Smells Detected and the Vote Level 2 Bad Smells Detected, respectively, with a total of 80 situations.

Table 5.4: Original Bad Smells Detected

(a) After Extract Class

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
DC[T4 | T5]	| +12		| +50	+4.5 | +50		+9.1 | +9.6	| +40	+? | +37.5
FE[T3 | T4]	+11.1 | +1.1	+6.7 | +11.8	-100 |	+1,500 | +3	| -2.6	+46.2 | +0.2	+14.3 | +1	+28.6 | +4.3
LC[T1]								
ZC[T1 | T5]		| +5.6	| +3.2	| +9.2	| +5.6	| +9.7	| +4.5	| +23.5
LM[T1 | T2 | T3]	+8.1 | | +2.1	+3.7 | | +1.7	| | -73		+10.5 | | +1.9	+7.9 | |	+13.3 | |	+25 | | +3
LP[T1 | T2]		-100 |		| +1.1		+4.3 | +4.5	+5.9 | +1.8	+700 |
MC[T1]						+5		
RB[T1 | T4 | T5]	+5.8 | -3.9 |					+2.8 | -1 |		
SS[T4]				+5			+2	
SG[T1 | T5]								

(b) After Extract Method

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
DC[T4 | T5]								
FE[T3 | T4]	+50 | +7.3	| +10.8			| -10.5			+100 | +31.9
LC[T1]								
ZC[T1 | T5]					+1,000 |			
LM[T1 | T2 | T3]	-1.6 | | -94.7	| | -90.6	| | -90.5		-10.5 | | -92.5			| | -98
LP[T1 | T2]	+8.3 | +12.5	+83.3 | +77.8	| +6.9					| +20
MC[T1]								
RB[T1 | T4 | T5]	| +19.6 |				| | -100			
SS[T4]	+2.6							
SG[T1 | T5]								

(c) After Move Method

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
DC[T4 | T5]			| -50			+4.5 | -1.9	+20 |	
FE[T3 | T4]	-100 | -0.4	-100 | -3.9	-100 |			-84.6 | -0.2	-85.7 | -0.5	-85.7 | -2.1
LC[T1]								
ZC[T1 | T5]		| -2.8						
LM[T1 | T2 | T3]	| | -1.2					-1.6 | |		+8.3 | |
LP[T1 | T2]								
MC[T1]						+2.5		
RB[T1 | T4 | T5]	| -2 |					+0.7 | |		
SS[T4]						-0.6		+50
SG[T1 | T5]								

(d) After Replace Refactoring

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
DC[T4 | T5]	| -8							
FE[T3 | T4]	| +0.7						| +2.1	
LC[T1]								
ZC[T1 | T5]	| +3.1		| +11.1		| +5.6		| +10.4	
LM[T1 | T2 | T3]	+1.6 | |		| | +2.7				+15.6 | |	
LP[T1 | T2]								
MC[T1]								
RB[T1 | T4 | T5]	+38.5 | | +111.1		| | +125		+100 | |		+190.5 | | +4,000	
SS[T4]							+4.1	
SG[T1 | T5]	| +2.6		| +14.3					

S1: Checkstyle-5.6, S2: Commons-codec, S3: Commons-io, S4: Commons-lang, S5: Commons-logging, S6: JHotDraw-7.5.1, S7: Quartz-1.8.3, S8: Squirrel_sql-3.1.2, T1: Decor, T2: Designite, T3: JDeodorant, T4: JSpIRIT, and T5: Organic

Table 5.5: Standardized Bad Smells Detected

(a) After Extract Class

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
DC[T4 | T5]	0 | +12		0 | +50	+4.5 | +50		+9.1 | +9.6	0 | +40	+? | +37.5
FE[T3 | T4]	+11.1 | +0.7	+6.7 | +11.8	-100 | 0	+600 | +2.5	0 | -2.8	+46.1 | +0.4	+14.3 | +1	+28.6 | +4.3
LC[T1]								
ZC[T1 | T5]		0 | +5.6	0 | +3.2	0 | +10.1	0 | +5.6	0 | +9.7	0 | +4.5	0 | +23.5
LM[T1 | T2 | T3]	+8.1 | 0 | +1.9	+3.7 | 0 | +3.4	0 | 0 | -72.7		+10.5 | 0 | +2.4	+7.9 | 0 | 0	+13.3 | 0 | 0	+25 | 0 | +7.1
LP[T1 | T2]		-100 | 0		0 | +2.6		+4.3 | +4.1	+5.9 | +4.3	+700 | 0
MC[T1]						+5		
RB[T1 | T4 | T5]	+5.8 | -3.9 | 0					+2.7 | -1 | 0		
SS[T4]				+8.3			+2.1	
SG[T1 | T5]								

(b) After Extract Method

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
DC[T4 | T5]								
FE[T3 | T4]	+50 | +6.9	| +10.8			| -11.1			+100 | +31.9
LC[T1]								
ZC[T1 | T5]					+1,000 |			
LM[T1 | T2 | T3]	-1.6 | | -91.3	| | -84.7	| | -93.9		-10.5 | | -92.9			| | -95.2
LP[T1 | T2]	+8.3 | +4.5	+83.3 | +87.5						
MC[T1]								
RB[T1 | T4 | T5]	| +19.6 |				| | -100			
SS[T4]	+2.7							
SG[T1 | T5]								

(c) After Move Method

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
DC[T4 | T5]			| -50			+4.5 | -1.9	+20 |	
FE[T3 | T4]	-100 | -0.4	-100 | -9.3	-100 |			-84.6 |	-85.7 | -0.5	-85.7 | -2.1
LC[T1]								
ZC[T1 | T5]		| -2.8						
LM[T1 | T2 | T3]						-1.6 | |		+8.3 | |
LP[T1 | T2]								
MC[T1]						+2.5		
RB[T1 | T4 | T5]	| +2 |					+0.7 | |		
SS[T4]						-0.6		+50
SG[T1 | T5]								

(d) After Replace Refactoring

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
DC[T4 | T5]	| -8							
FE[T3 | T4]	| +0.7						| +2.1	
LC[T1]								
ZC[T1 | T5]	| +3.2		| +11.3		| +5.6		| +10.4	
LM[T1 | T2 | T3]	+1.6 | |						+15.6 | |	
LP[T1 | T2]								|
MC[T1]								
RB[T1 | T4 | T5]	+38.5 | | +111.1		| | +125		+100 | |		+190.5 | | +4,000	
SS[T4]							+4.2	
SG[T1 | T5]	| +2.6		| +14.3					

S1: Checkstyle-5.6, S2: Commons-codec, S3: Commons-io, S4: Commons-lang, S5: Commons-logging, S6: JHotDraw-7.5.1, S7: Quartz-1.8.3, S8: Squirrel_sql-3.1.2, T1: Decor, T2: Designite, T3: JDeodorant, T4: JSpIRIT, and T5: Organic

Table 5.6: Vote Level 1 Bad Smells Detected

(a) After Extract Class

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
Data Class	+12		+50	+11.5		+9.7	+26.7	+50
Feature Envy	+1.7	+10.6	-100	+3.3	-2.8	+1.1	+1.5	+7.5
Large Class								
Lazy Class		+5.4	+3.2	+10.1	+5.5	+10.6	+4.5	+16.7
Long Method	+5.6	+3.3	-61.5		+3.8	+6.2	+10.7	+12
Long Parameter List				+2.6		+4.1	+4.3	
Message Chain						+5		
Refused Bequest	+0.8					+1.2		
Shotgun Surgery				+8.3			+2.1	
Speculative Generality								

(b) After Extract Method

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
Data Class								
Feature Envy	+9	+9.7			-11.1			+30.2
Large Class								
Lazy Class					+22.2			
Long Method	-31.8	-48.9	-79.5		-59.6			-72
Long Parameter List	+4.3	+87.5						
Message Chain								
Refused Bequest	+8.4							
Shotgun Surgery	+2.7							
Speculative Generality								

(c) After Move Method

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
Data Class			-50				+6.7	
Feature Envy	-5.5	-13.3				-1.7	-3	-11.3
Large Class								
Lazy Class		-2.7						
Long Method						-1.2		
Long Parameter List								
Message Chain						+2,5		
Refused Bequest	-0.8					+0.4		
Shotgun Surgery						-0.6		+50
Speculative Generality								

(d) After Replace Refactoring

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
Data Class	-8							
Feature Envy	+0.7						+2	
Large Class								
Lazy Class	+3.2		+11.3		+5.5		+10.4	
Long Method	+1						+12.5	
Long Parameter List								
Message Chain								
Refused Bequest	+16.8		+125		+75		+102.6	
Shotgun Surgery							+4.2	
Speculative Generality	+2.6		+14.3					

S1: Checkstyle-5.6, S2: Commons-codec, S3: Commons-io, S4: Commons-lang, S5: Commons-logging, S6: JHotDraw-7.5.1, S7: Quartz-1.8.3, S8: Squirrel_sql-3.1.2

Table 5.7: Vote Level 2 Bad Smells Detected

(a) After Extract Class

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
Data Class								
Feature Envy	-33.3	+25		+100		+66.7		
Large Class								
Lazy Class						-17.6		+?
Long Method	+2.8				+10			
Long Parameter List		-100				+4.3	+5.9	+700
Message Chain								
Refused Bequest								
Shotgun Surgery								
Speculative Generality								

(b) After Extract Method

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
Data Class								
Feature Envy	+66.7							+600
Large Class								
Lazy Class					+600			
Long Method	-97.2	-66.7			-100			-80
Long Parameter List	+9.1	+83.3						
Message Chain								
Refused Bequest					-100			
Shotgun Surgery								
Speculative Generality								

(c) After Move Method

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
Data Class								
Feature Envy	-100	-100				-66.7	-100	-100
Large Class								
Lazy Class								
Long Method								-20
Long Parameter List								
Message Chain								
Refused Bequest								
Shotgun Surgery								
Speculative Generality								

(d) After Replace Refactoring

Bad Smell	S1 (%)	S2 (%)	S3 (%)	S4 (%)	S5 (%)	S6 (%)	S7 (%)	S8 (%)
Data Class								
Feature Envy								
Large Class								
Lazy Class								
Long Method								
Long Parameter List								
Message Chain								
Refused Bequest	+1,000						+4,000	
Shotgun Surgery								
Speculative Generality								

S1: Checkstyle-5.6, S2: Commons-codec, S3: Commons-io, S4: Commons-lang, S5: Commons-logging, S6: JHotDraw-7.5.1, S7: Quartz-1.8.3, S8: Squirrel_sql-3.1.2

We classified the results obtained after a comparative analysis between the refactored version's data with the system's original version into three different types: decrease, increase, and neutral refactoring.

Decrease

This result type represents the decrease in the number of bad smells found in the system's refactored version compared to its original version. We may see this case easily in tables 5.4, 5.5, 5.6 and 5.7, with the minus sign ("-"), representing the percentage of decrease in the number of bad smells found.

Increase

This result type represents the increase in the number of bad smells found in the system's refactored version compared to its original version. We may observe this case easily in tables 5.4, 5.5, 5.6 and 5.7, represented by the plus sign ("+"), indicating the percentage of increase in the number of bad smells found. We may find special situations characterized by the appearance of bad smells in the refactored version that were not in the original version. These particular situations present the "?" after the "+" sign.

Neutral

This result type represents the non-existence or no-change in the number of bad smells found in the system's refactored version compared to its original version. Tables 5.4, 5.5, 5.6 and 5.7 represent this case by an empty entry where there was no change in the number of bad smells found. It is worth mentioning that cases in which the refactoring process did not carry out, as mentioned above, fall under the neutral cases.

5.3 Comparative Results

This section presents the comparative results after the application of (i) Extract Class, (ii) Extract Method, (iii) Move Method and (iv) Replace Refactorings.

Tables 5.4a, 5.5a, 5.6a and 5.7a present the results after the application of Extract Class; tables 5.4b, 5.5b, 5.6b and 5.7b presents the results after the application of the Extract Method; tables 5.4c 5.5c, 5.6c and 5.7c present the results after the application of Move Method, and tables 5.4d, 5.5d, 5.6d and 5.7d present the results after the application of Replace Refactoring.

We identified in tables 5.4a and 5.5a regarding Extract class; tables 5.4b and 5.5b for Extract method; tables 5.4c and 5.5c related Move method, and tables 5.4d and 5.5d, for Replace refactoring **152** situations according to our data.

Also, all the tables regarding the application of Extract Class, Extract Method, Move Method and Replace Refactorings, respectively represented in tables 5.6a and 5.7a; 5.6b and 5.7b; 5.6c and 5.7c, and 5.6d and 5.7d, we have **80** situations, according our data. In the next four subsections, we detail each one, separated by the type of refactoring performed.

5.3.1 Extract Class Refactoring

Tables 5.4a and 5.5a show that from the 152 situations found, the number of bad smells *decreased* by 3.95% (6), *increased* by 32.24% (49), and remained the same number of bad smells in 63.82% (97).

The Refused Bequest bad smell presented a divergence of identification in the same system by different tools. For the Checkstyle-5.6 and JHotDraw-7.5.1 systems, Decor computed an *increase* in the number of bad smells, and JSpIRIT computed a *decrease*. The Data Class bad smell presents a unique situation. In the original version of the Squirrel_sql-3.1.2 system, JSpIRIT did not detect this type of bad smell.

Table 5.4a tell us that Feature Envy and Long Parameter List, for Commons-lang and Squirrel_sql-3.1.2, present an *increase* of 1,500% and 700%. Their absolute values represent an *increase* from 1 to 16 and from 1 to 8, respectively.

Table 5.5a, regarding Feature Envy and Long Parameter List, for Commons-lang and Squirrel_sql-3.1.2, we observed that they present an *increase* of 600% and 700%, and their absolute values *increase* from 1 to 7 and from 1 to 8, respectively.

Table 5.6a shows that after extract class refactoring, the number of bad smells *decreased* by 3.75% (3), *increased* by 41.25% (33), and remained the same number of bad smells in 55% (44), according to the 80 situations found.

Tables 5.4a, 5.5a and 5.6a, observing these tables, we note that the refactoring process did not influence the Large Class and Speculative Generality bad smells in any evaluated systems. In both cases, the number of bad smells detected remained the same. Message Chains bad smell only *increased* in one system, while in the other systems, the number of this smell remained the same.

Table 5.7a shows that the number of bad smells *decreased* by 3.75% (3), *increased* by 11.25% (9), and remained the same number of bad smells in 23.75% (19), considering the 80 situations.

The refactoring process did not influence the Data Class, Refused Bequest, and Speculative Generality bad smells in any evaluated systems. In these cases, the number of bad smells detected remained the same. The Lazy Class bad smell presents a unique situation. The original version of the Squirrel_sql-3.1.2 system did not detect this type of bad smell. Its refactored version caused one bad smell. Long Parameter List for Squirrel_sql-3.1.2, which present an *increase* of 700%, their absolute values represent an *increase* from 1 to 8.

5.3.2 Extract Method Refactoring

Table 5.4b shows that in 5.92% (9) the number of bad smells *decreased*; in 9.21% (14) the number of bad smells *increased*, and in 84.87% (129) the number of bad smells remained the same.

Table 5.5b shows that in 5.26% (8), the number of bad smells *decreased*; in 7.89% (12), the number of bad smells *increased*, and in 86.84% (132), the number of bad smells remained the same.

Tables 5.4b and 5.5b show that the Lazy Class and Shotgun Surgery bad smells only *increased* in one system. In the other systems, the number remained the same. Lazy Class for the Commons-logging presents an *increase* of 1,000%, and their absolute value represents an *increase* from 1 to 11.

Table 5.6b tell us that in 7.50% (6) the number of bad smells *decreased*; in 10% (8) the number of bad smells *increased*, and in 45% (36) the number of bad smells remained the same.

The Lazy Class, Refused Bequest, and Shotgun Surgery bad smells only *increased* in one system. In the remainder of the systems, the number remains the same.

Tables 5.4b, 5.5b and 5.6b, observing these tables, we see that the refactoring operation did not influence the Data Class, Large Class, Message Chains, and Speculative Generality bad smells in any evaluated systems. In all these cases, the number of bad smells detected remained the same.

Table 5.7b shows that in 6.25% (5) the number of bad smells *decreased*; in 6.25% (5) the number of bad smells *increased*, and in 8.75% (7) the number of bad smells remained the same.

The refactoring operation did not influence the Speculative Generality bad smell in any evaluated systems. In this case, the number of bad smells detected remained the same. The Lazy Class and Refused Bequest bad smell only *increased* and *decreased* in one system, respectively. In the other systems, the number remained the same. Lazy Class for the Commons-logging, which presents an *increase* of 600%, their absolute value represents an *increase* from 1 to 7.

5.3.3 Move Method Refactoring

Table 5.4c says that the number of bad smells *decreased* by 11.84% (18); in 3.95% (6) the number of bad smells *increased*, and bad smells remained the same in 84.21% (128).

Table 5.5c, observing this table, we may see that the number of bad smells *decreased* by 9.87% (15); in 4.61% (7), the number of bad smells *increased*, and bad smells remained the same in 85.53% (130).

Tables 5.4c and 5.5c, we observe that the Message Chains bad smell *increased* in one system, while Lazy Class bad smell *decreased* in one system. In the remainder systems, both bad smells remain the same. The Data Class bad smell presented divergence of identification in the same system by different tools. For the JHotDraw-7.5.1 system, SpIRIT computed an *increase* in the number of bad smells, and Organic computed a *decrease*.

Table 5.6c shows that the number of bad smells *decreased* by 12.50% (10); while *increased* 5% (4). The bad smells remained the same in 70% (56).

The Message Chains bad smell only *increased* in one system, while Lazy Class and Long Method bad smell only *decreased* in one system. In other systems, both bad smells remained the same.

Tables 5.4c, 5.5c and 5.6c show that the refactoring operation did not influence the Large Class, Long Parameter List, and Speculative Generality bad smells in any of the evaluated systems; in these cases, the number of bad smells remained the same.

Table 5.7c says that the number of bad smells *decreased* by 7.50% (6); in 0% (0) the number of bad smells *increased*. The bad smells remained the same in 25% (20).

The refactoring operation did not influence the Data Class, Lazy Class, Long Parameter List, Refused Bequest, and Speculative Generality bad smells in any of the evaluated systems; in these cases, the number of bad smells remained the same. The Long Method bad smell only *decreased* in one system. In other systems, it remained the same.

5.3.4 Replace Refactoring

Table 5.4d, observing this table, we see that the number of bad smells *decreased* 0.66% (1), *increased* 11.84% (18), and in 87.50% (133) cases remained the same.

Table 5.5d, shows the number of bad smells *decreased* 0.66% (1), *increased* 11.18% (17), and in 88.16% (134) cases remained the same.

Tables 5.4d and 5.5d say that Refused Bequest for Quartz-1.8.3 presented an *increase* of 190.5% and 4,000%. Their absolute values represent an *increase* from 21 to 61 and from 1 to 41, respectively.

Table 5.6d. This table shows the number of bad smells *decreased* by 1.25% (1), *increased* by 18.75% (15), and remaining the same in 55% (44) cases.

Tables 5.4d, 5.5d and 5.6d, here, the Large Class, Long Parameter List, and Message Chains bad smells were not influenced by the refactoring process in any evaluated systems. In all cases, the number of bad smells detected remained the same. The Data Class and Shotgun Surgery bad smells only *decreased* and *increased*, respectively, in one system; they remained the same in the other systems.

Table 5.7d shows that the number of bad smells *decreased* 0% (0), *increased* 2.50% (2), and in 22.50% (18) cases remained the same.

The refactoring process did not influence the Feature Envy, Lazy Class, Long Method, Long Parameter List, and Speculative Generality bad smells in any evaluated systems. In all these cases, the number of bad smells detected remained the same. Refused Bequest for Checkstyle-5.6 and Quartz-1.8.3 presented an *increase* of 1,000% and 4,000%. Their absolute values represent an *increase* from 2 to 22 and from 1 to 41, respectively.

5.4 Results Summarization

The refactorings applied impacted a lot of the number of analyzed bad smells. For three out of the four types of refactorings applied, the number of different bad smells introduced was more significant than the number of different bad smells removed. For

Move Method, the number of different bad smells removed was higher than the number of different bad smells introduced.

In some cases, different bad smells detection tools computed the introduction of a bad smell type and removed them. For instance, the Extract Class refactoring was the one that impacted the highest number of different bad smells, eight in total, positively and negatively.

Table 5.8: Anwsering the Request Questions with Original Detections

	RQ3.1. Does the automated refactoring remove bad smells?		**RQ3.2. Does the automated refactoring introduce bad smells?**	
	Answer: Affirmative		**Answer: Affirmative**	
Refactoring	**Bad Smell**	**% of system**	**Bad Smell**	**% of system**
Extract Class	Feature Envy	25%	Data Class	75%
	Long Method	12.5%	Feature Envy	75%
	Long Parameter List	12.5%	Lazy Class	87.5%
	Refused Bequest	25%	Long Method	75%
			Long Parameter List	50%
			Message Chains	12.5%
			Refused Bequest	25%
			Shotgun Surgery	25%
Extract Method	Feature Envy	12.5%	Feature Envy	37.5%
	Long Method	62.5%	Lazy Class	12.5%
	Refused Bequest	12.5%	Long Parameter List	50%
			Refused Bequest	12.5%
			Shotgun Surgery	12.5%
Move Method	Data Class	25%	Data Class	25%
	Feature Envy	75%	Long Method	12.5%
	Lazy Class	12.5%	Message Chains	12.5%
	Long Method	25%	Refused Bequest	12.5%
	Refused Bequest	12.5%	Shotgun Surgery	12.5%
	Shotgun Surgery	12.5%		
Replace Refactoring	Data Class	12.5%	Feature Envy	25%
			Lazy Class	50%
			Long Method	37.5%
			Refused Bequest	50%
			Shotgun Surgery	12.5%
			Speculative Generality	25%

Using JDeodorant, we apply the Extract Method refactoring from detecting Long Method bad smell. We know that for JDeodorant, the Long Method detection after the Extract Method applied may have high positive effects if compared to the other tools. Therefore, we decided to document the tool's results to determine if the refactoring suggestion proposed by JDeodorant may introduce another type of bad smell that it may detect. According to tables 5.4 and 5.5 this situation occurred for the Checkstyle-5.6 and Squirrel_sql-3.1.2 systems, where the Extract Method refactoring introduces the Feature Envy bad smell. These also may occur with Move Method that performs refactoring from detecting the Feature Envy bad smell. Nevertheless, we observe that JDeodorant did not introduce another type of bad smell that may be detected for the systems analyzed.

Table 5.9: Anwsering the Request Questions with Standardized Detections

	RQ3.1. Does the automated refactoring remove bad smells?		**RQ3.2. Does the automated refactoring introduce bad smells?**	
	Answer: Affirmative		**Answer: Affirmative**	
Refactoring	**Bad Smell**	**% of system**	**Bad Smell**	**% of system**
Extract Class	Feature Envy	25%	Data Class	75%
	Long Method	12.5%	Feature Envy	75%
	Long Parameter List	12.5%	Lazy Class	87.5%
	Refused Bequest	25%	Long Method	75%
			Long Parameter List	50%
			Message Chains	12.5%
			Refused Bequest	25%
			Shotgun Surgery	25%
Extract Method	Feature Envy	12.5%	Feature Envy	37.5%
	Long Method	62.5%	Lazy Class	12.5%
	Refused Bequest	12.5%	Long Parameter List	25%
			Refused Bequest	12.5%
			Shotgun Surgery	12.5%
Move Method	Data Class	25%	Data Class	25%
	Feature Envy	75%	Long Method	12.5%
	Lazy Class	12.5%	Message Chains	12.5%
	Long Method	12.5%	Refused Bequest	25%
	Shotgun Surgery	12.5%	Shotgun Surgery	12.5%
Replace Refactoring	Data Class	12.5%	Feature Envy	25%
			Lazy Class	50%
			Long Method	25%
			Refused Bequest	50%
			Shotgun Surgery	12.5%
			Speculative Generality	25%

For the specific situations of Extract Method to Long Method and Move Method to Feature Envy for JDeodorant, we consider that it may cause an introduction of the same type of smell. We observed these situations where re-refactorings were applied, considering that all the tool's suggestions have been applied for its entire use. However, we decided not to focus the analyses in this aspect. Our website[2] provide sufficient data to find these cases.

Taking into account the results obtained in tables 5.4, 5.5, 5.6 and 5.7 according to the analysis performed on the eight systems, tables 5.8, 5.9, 5.10 and 5.11, respectively,

[2] `https://cleitonsilvat.github.io/dissertation/`

exhibits the bad smells removed and the ones introduced after applying the automatic refactoring process. Therefore, we use Tables 5.8, 5.9, 5.10 and 5.11 to answer the research questions **RQ3.1** and **RQ3.2**.

Table 5.10: Anwsering the Request Questions with Vote Level 1

	RQ3.1. Does the automated refactoring remove bad smells?		**RQ3.2. Does the automated refactoring introduce bad smells?**	
	Answer: Affirmative		**Answer: Affirmative**	
Refactoring	**Bad Smell**	**% of system**	**Bad Smell**	**% of system**
Extract Class	Feature Envy	25%	Data Class	75%
	Long Method	12.5%	Feature Envy	75%
			Lazy Class	87.5%
			Long Method	75%
			Long Parameter List	37.5%
			Message Chains	12.5%
			Refused Bequest	25%
			Shotgun Surgery	25%
Extract Method	Feature Envy	12.5%	Feature Envy	37.5%
	Long Method	62.5%	Lazy Class	12.5%
			Long Parameter List	25%
			Refused Bequest	12.5%
			Shotgun Surgery	12.5%
Move Method	Data Class	12.5%	Data Class	12.5%
	Feature Envy	62.5%	Message Chains	12.5%
	Lazy Class	12.5%	Refused Bequest	12.5%
	Long Method	12.5%	Shotgun Surgery	12.5%
	Refused Bequest	12.5%		
	Shotgun Surgery	12.5%		
Replace Refactoring	Data Class	12.5%	Feature Envy	25%
			Lazy Class	50%
			Long Method	25%
			Refused Bequest	50%
			Shotgun Surgery	12.5%
			Speculative Generality	25%

The first column of tables 5.8, 5.9, 5.10 and 5.11 shows the name of the refactoring applied. The second column presents the name of the bad smells removed, and the third column the ones introduced. We show the percentage of systems where the tools detected the bad smells were removed or introduced.

With the results found, exhibited in tables 5.4, 5.5, 5.6, 5.7 and tables 5.8, 5.9, 5.10, 5.11, we answer the RQs raised as follows.

RQ3. What are the impacts of automated refactoring on the detection of bad smells? The impacts caused by the automated refactoring process may be positive and negative. As we evaluate different refactoring and bad smell detection tools, we may analyze them from different perspectives.

RQ3.1. Does the automated refactoring process remove bad smells? The answer to **RQ3.1** is AFFIRMATIVE. A single refactoring strategy may remove more than one bad smell.

RQ3.2. Does the automated refactoring process introduce bad smells? The answer to **RQ3.2** is AFFIRMATIVE. A single refactoring strategy may introduce more than one bad smell.

Table 5.11: Anwsering the Request Questions with Vote Level 2

	RQ3.1. Does the automated refactoring remove bad smells?		**RQ3.2. Does the automated refactoring introduce bad smells?**	
	Answer: Affirmative		**Answer: Affirmative**	
Refactoring	**Bad Smell**	**% of system**	**Bad Smell**	**% of system**
Extract Class	Feature Envy	12.5%	Feature Envy	37.5%
	Lazy Class	12.5%	Lazy Class	12.5%
	Long Parameter List	12.5%	Long Method	25%
			Long Parameter List	37.5%
Extract Method	Long Method	50%	Feature Envy	25%
	Refused Bequest	12.5%	Lazy Class	12.5%
			Long Parameter List	25%
Move Method	Feature Envy	62.5%		
	Long Method	12.5%		
Replace Refactoring			Refused Bequest	25%

5.5 Analyzes

To obtain a general overview of the analyzes carried out, we performed an aggregated analyzes computing the total number of bad smells identified in each version of the systems analyzed. To compute this data, we use two of the perspectives used in our research, vote levels 1 and 2, considering that these were the only ones which it is possible to guarantee the non-existence of detection carried out for the same bad smell, i.e., duplicate detections made by the same tool or between different tools. Tables 5.12 and 5.13 show our data. The first column of these tables shows the name of the systems analyzed. The next five columns show the total value of bad smells identified in each version of the system.

Table 5.12: Agglutinate Total Smells Vote Level 1

System	Original Version	Move Method	Replace Refactoring	Extract Method	Extract Class
Checkstyle-5.6	977	960	1008	953	997
Commons-codec	273	257	273	247	290
Commons-io	129	128	142	98	105
Commons-lang	898	898	898		938
Commons-logging	128		135	97	130
JHotDraw-7.5.1	1,495	1,485			1,546
Quartz-1.8.3	451	446	511		469
Squirrel_sql-3.1.2	140	135		120	157

Table 5.13: Agglutinate Total Smells Vote Level 2

System	Original Version	Move Method	Replace Refactoring	Extract Method	Extract Class
Checkstyle-5.6	57	54	77	25	57
Commons-codec	22	18	22	21	17
Commons-io					
Commons-lang	1	1	1		2
Commons-logging	17		17	8	18
JHotDraw-7.5.1	113	111			115
Quartz-1.8.3	26	25	66		27
Squirrel_sql-3.1.2	7	7		9	15

Figures 5.1 and 5.2 present a clearly visualization of our aggregated data, making it possible to identify an increase or decrease in the number of bad smells after each type of performed refactoring. The X-axis represents the name of the systems analyzed, and the Y-axis represents each version of the system created from the application of a refactoring type. We presented the scale with the percentage value and represented the cases where there is a decrease in the number of bad smells by the negative values with green coloring. Moreover, we represent the cases in which there is an increase in the number of bad smells by the positive values with red coloring, and the cases in which there is no change in the number of bad smells, we represented with white coloring, with zero value on it.

The cases where (i) refactoring strategies were not applied, (ii) such bad smells were not detected in this version and also for (iii) the specific case of Commons-io which at least two tools detected no bad smells, as presented in Table 5.13, we represented with grey coloring, with no value on it.

According to our data, we have 26 valid situations in Figure 5.1 and 22 valid situations in Figure 5.2. In Figure 5.1, the number of bad smells decreased by 46.15%; in 42.31% the number of bad smells increased, and bad smells remained the same in

11.54%. For Figure 5.2, the number of bad smells decreased by 36.36%; in 36.36% the number of bad smells increased, and bad smells remained the same in 27.27%.

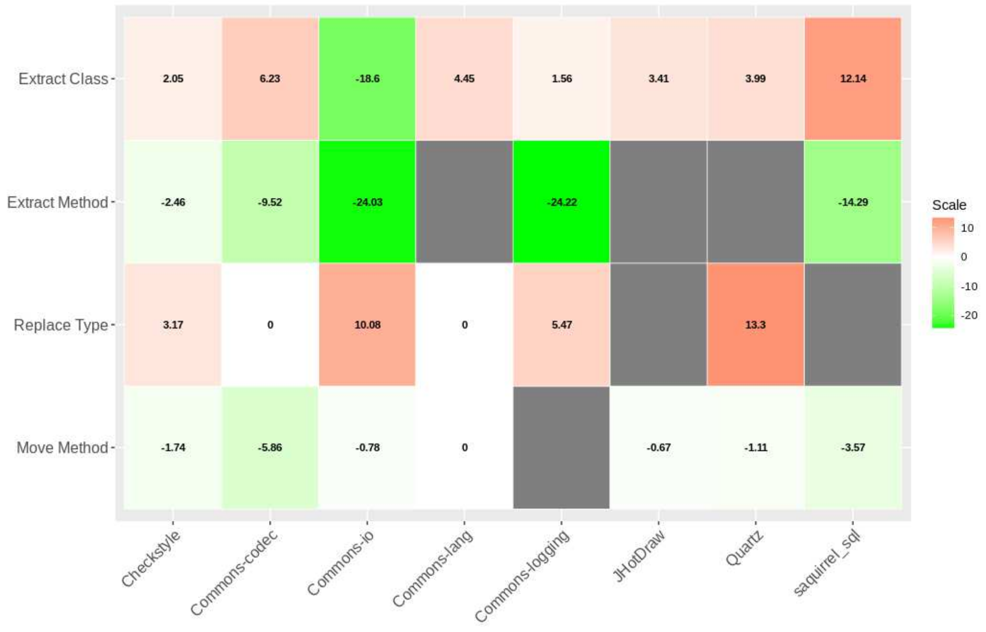

Figure 5.1: Agglutinate Analysis Vote Level 1

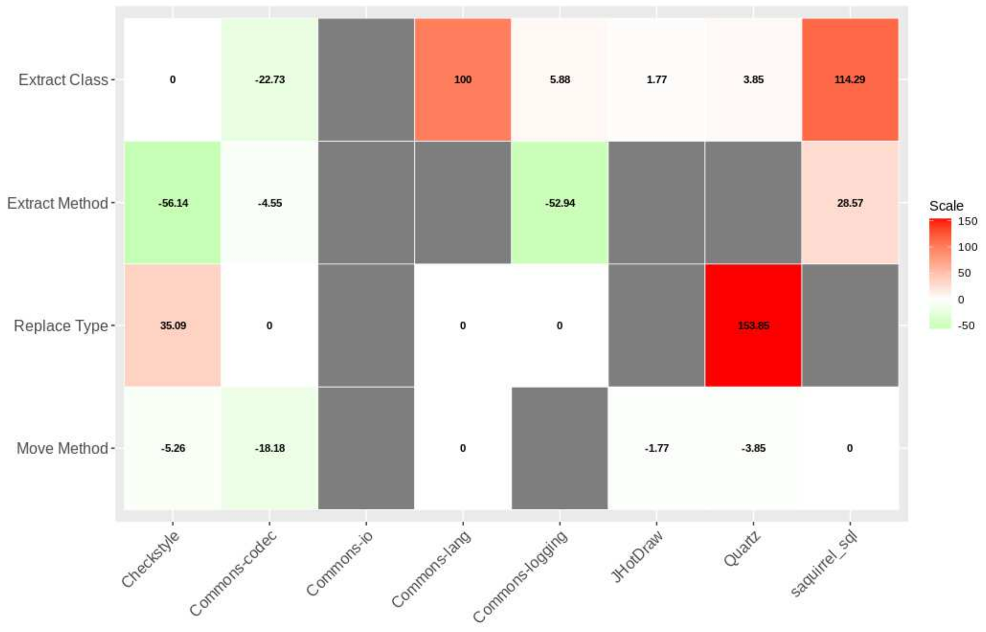

Figure 5.2: Agglutinate Analysis Vote Level 2

5.6 Discussions

Observing our analyses, we may see that the refactoring process oscillates the value of the bad smells present in the analyzed codes.

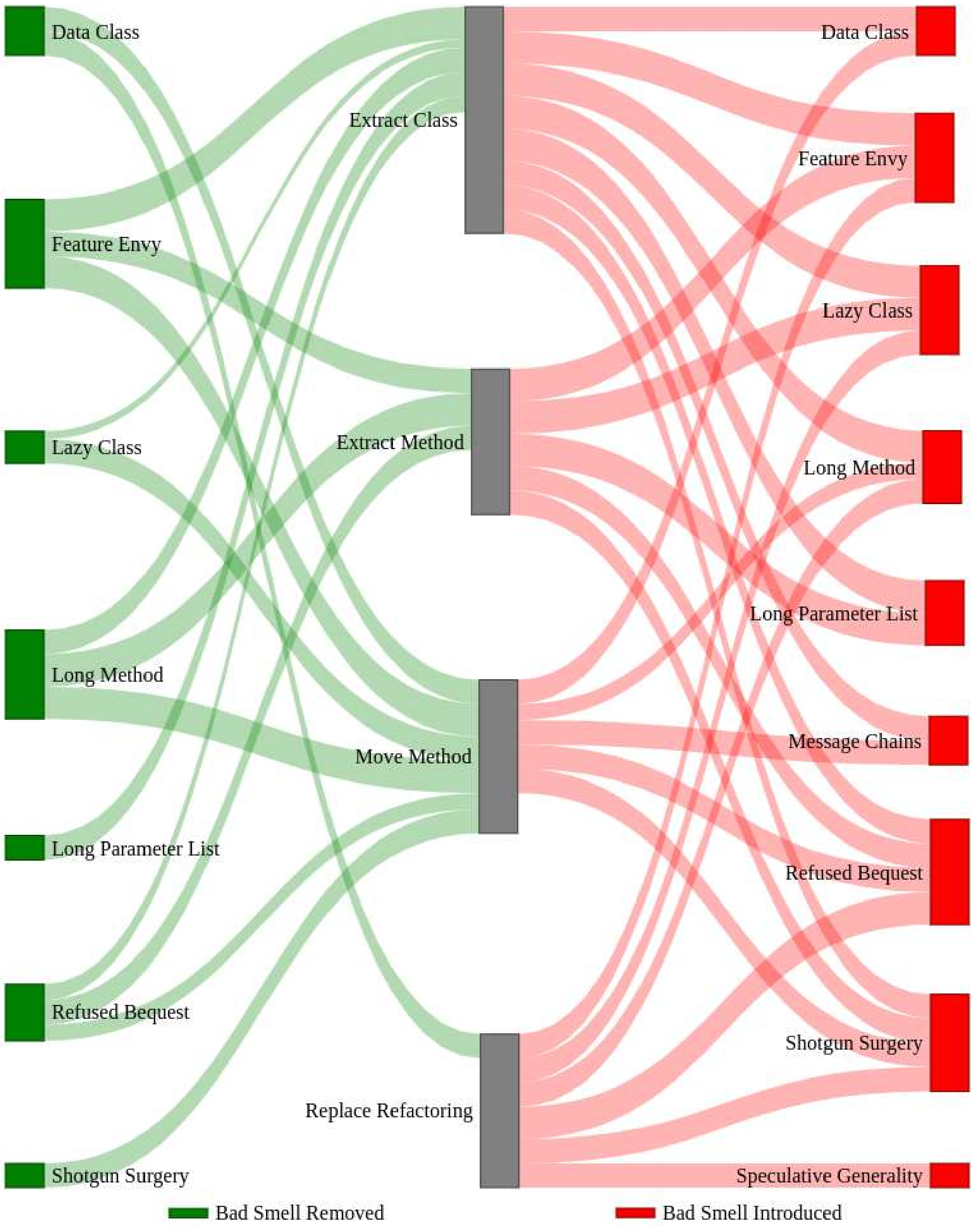

Figure 5.3: Bad Smells Introduced and Removed by Refactoring

With the detailed analyses presented in this chapter, we may accurately identify which bad smells are being introduced or removed by the refactoring process. To demonstrate these situations, Figure 5.3 presents which bad smells may be introduced or removed by the refactoring process according to our four perspectives addressed in our empirical study.

Each connection in Figure 5.3 represents the sum of the cases found in the four perspectives analyzed. Taking into account, the maximum value between a connection, which may represent a bad smell introduced or removed by the refactoring process, will be four. The first column of Figure 5.3 shows the list of bad smells removed by the refactoring process with green coloring. The second column represents the name of the refactorings analyzed. Finally, the last column lists the bad smells introduced by the refactoring process with red coloring.

Given the results obtained in the studies carried out in this dissertation, the Systematic Literature Review (SLR), Chapter 3, and Empirical Study, the latter presented in this chapter, and considering that the Empirical Study identified which bad smells are introduced or removed by the refactoring process, we decide to make a comparison between their results. Although we conducted our empirical study analyzing four different refactorings, we focused only on presenting insights about them. Figure 5.4 shows the comparison made.

Figure 5.4 exhibits which bad smells may be introduced or removed by the refactoring process considering the empirical study and SLR. The first column of the figure shows the bad smells that the refactoring process may remove. The second column shows the refactorings that were applied. The third column shows the bad smells that the refactoring process may introduce. Finally, in the fourth column, we present the bad smells related to refactorings, but we do not have enough data to characterize their behavior according to our Empirical Study.

Columns one, two, and three clearly show the data obtained in our empirical study. Columns two and four show the relationships mentioned in our SLR. There are data found in our empirical study that prove the relationships found in our SLR. As mentioned, columns one and three represent bad smells that have been removed, with green color, or introduced, with red color, by the refactoring process according to our empirical study data. To exhibits this classification under the relationships found in our SLR, we used the grey color. Therefore, the grey color denotes the relationship mentioned in our SLR and has now been classified in our empirical study. We presented the cases in which we may not classify the type of relationship in column four with blue color.

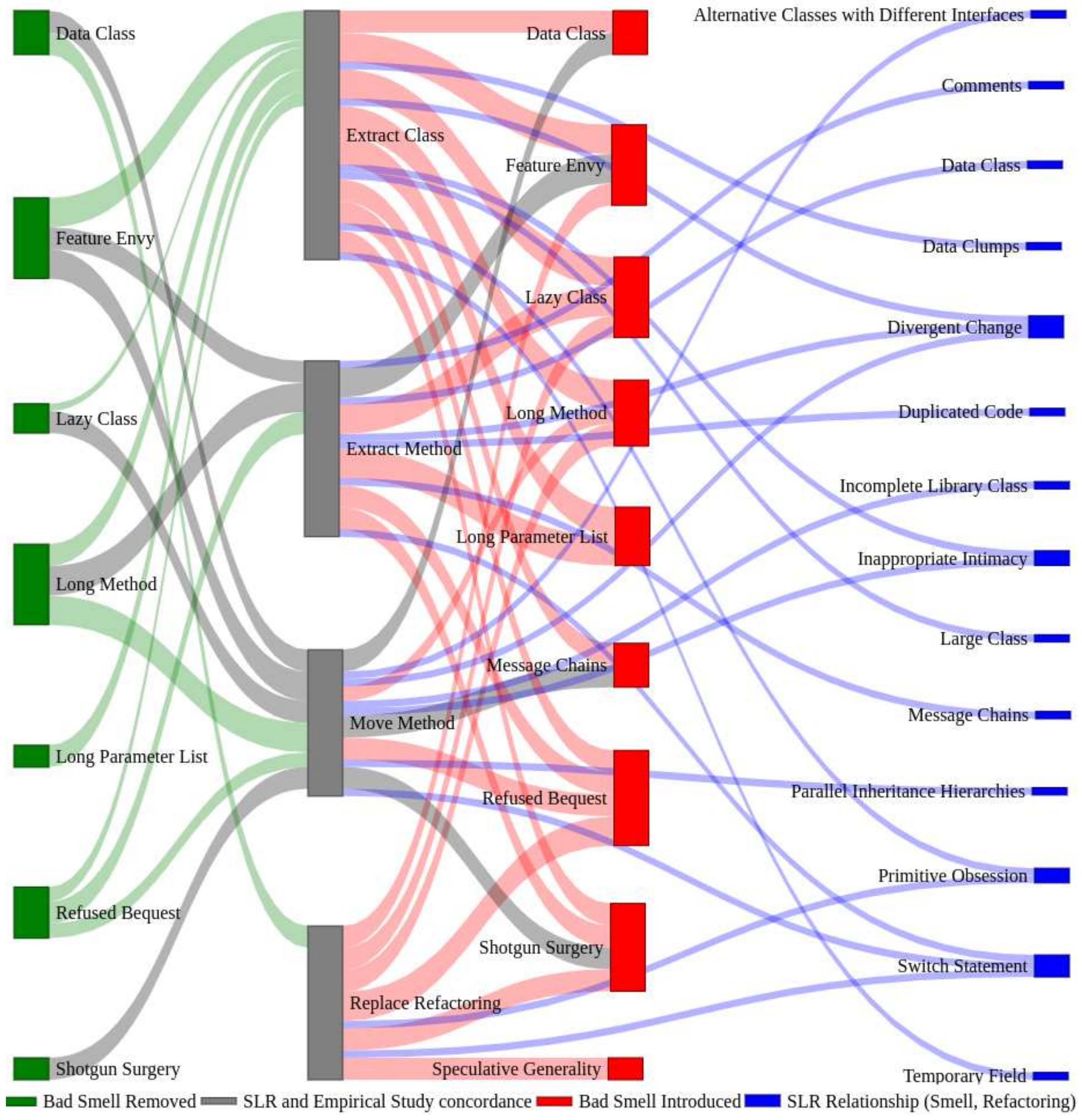

Figure 5.4: Bad Smells and Refactoring Contrast

In some cases, our SLR finds works that mention a relationship between bad smell and refactoring which we have analyzed in our empirical study, but we could not classify them with our data. These cases happen for the Data Class, Large Class, and Message Chains bad smells presents in column four.

With the analysis of this information, we have data for:

(i) validate, in cases where our empirical study complies with the SLR

(ii) contrast, for cases in which the data found in our empirical study, the SLR did not mention it, and

(iii) complementary, considering that our empirical study results are valid and contribute to presenting new relationships, that should be considered in other works.

5.7 Threats to Validity

We discuss the threats to validity listed by Wohlin et al. [86]: construct, internal, conclusion, and external. These threats show possible limitations found in the experimental paradigm: (a) theory, involving the direct relationship between cause and effect; and (b) observations, transforming into treatment and output. Discussing these raised threats, we present the strategies adopted to mitigate them.

Construct Validity

This threat concerns the relationship between which the experiment setting reflects the theory. The bad smell detection tool may not detect all bad smells present in the system. To minimize this threat, we used five different bad smell detection tools and consider all their detections as valid. The data analyses may present cherry-pick, inducing the reader to focus only on the part of the data. We present perspectives from different dimensions to represent the detection provide by all the tools to minimize this threat.

Internal Validity

This threat may affect the independent variable concerning causality. The detailed specification of the process performed and the tools used, guaranteed the reproducibility of this study. Possible limitations may cause changes made to the tools or the operations provided by them. For the refactoring tool used, we performed only the refactorings suggested by it. The Eclipse IDE itself performed and documented all the adjustments made to solve trivial errors to some refactorings that caused compilation errors. For the bad smell detection tools, only the standard configuration of all of them was used, without any modification in their process to run or change the results provided by them.

Conclusion Validity

This threat may affect the ability to draw the correct conclusion. The data's comparative analysis may induce the reader to focus only on what the researcher wants. We performed the comparative analysis using four different perspectives to minimize this threat, taking into account only the original version and applying a single refactoring strategy. The results found were individually documented, without favoring any information or tool.

External Validity

This threat limits the ability to generalize the results beyond the experiment setting; for instance, the analyzed systems' representativeness to generalize to any other system. To reduce this threat's impact, we chose to use a known database focused on conducting empirical studies, the Qualitas Corpus. The number and systems evaluated were selected randomly, and the eight systems were sufficient to bring us insights into how the refactoring process can impact certain bad smells.

5.8 Final Remarks

This chapter presented empirical research analyzing the impact of refactoring on bad smells. To conduct this research, we selected eight open-source Java systems available in Qualitas Corpus. We applied four refactorings, measured their impact on ten different bad smells detected by five different tools, and performed the analysis in four different perspectives.

We have observed that the four types of refactorings generate a decrease, increase, and neutral alterations on the number of bad smells. Surprisingly, the amount of decrease in the number of bad smells was the lowest compared to the incease and neutral variations, except in the analysis after the Move Method refactoring was applied. We investigated which bad smells tend to be introduced and removed by the automatic refactoring operation and compute our results in four perspectives.

Regardless of the perspectives analyzed, we found that a refactoring strategy may introduce more bad smells types then remove them, except in the Move Method case. We observed that in some cases a bad smell type might be introduced and removed for the same refactoring strategy. For instance, this situation happened with the Feature Envy bad smell, which may be removed and introduced by Extract Class.

According to the results found and discussed in this chapter, we were able to achieve the goal of our dissertation. Therefore, in Chapter 6 we conclude our dissertation.

Chapter 6

Conclusion

In this dissertation, we presented a research study to assess the effects of refactoring on bad smells. First, we conduct a Systematic Literature Review (SLR) to identify the relationships on the subject discussed in the literature. In our SLR, we decided to demonstrate the existence of a relationship and not report which relationships represent positive points in the case of removal; and negative in the case of introducing a new bad smell. Second, to validate the SLR findings, we decided to conduct an Empirical Study to provide enough data to confirm and complement what the literature has presented.

Our SLR found 20 papers that show the direct relationship between 31 refactoring types and 16 bad smells proposed by Fowler. We also found seven tools that apply refactoring after detecting bad smells. We identified that the most discussed relationship in the literature is between Move Method and Feature Envy. It identified that:

1. there are different refactoring strategies than those discussed by Fowler to address bad smells;
2. the literature addresses most strategies defined in the Fowler's book, and
3. most refactoring tools found may not detect bad smells.

Therefore, in conducting our empirical study as it is a subjective matter, the definition of threshold to define what is a bad smell and what code fragment to refactor, we decided to use the tooling support. We attribute this subjectivity to the tools used, considering the thresholds used by them as being valid when defining code fragments to refactoring and identifying bad smells present in the source code.

Our analyses consider different perspectives to provide sufficient data for different analyses, affections, and readers. For those who trust the result and the definition of only one of the tools used, it is possible to perform the data's isolation and use as requested. For the more conservative in which they seek to define concepts through different sources, exploring their union or intersection of their results and definitions, we also provide sufficient data for this type of analysis.

In our Empirical Study, we analyzed the effects of automatic refactoring on bad smells. To conduct this study, we selected eight open-source Java systems available in the Qualitas Corpus. We applied four refactorings and measured their effects on ten different bad smells detected by five different tools. We investigated which bad smells tend to be introduced and removed by the refactoring operation. We have observed that the four types of refactorings generate a decrease, increase, and neutral alterations on the number of bad smells. Surprisingly the number of decrease cases was the lowest compared to the others. We found that a refactoring strategy may introduce more bad smells types than remove them, except in the Move Method case. We observed that in some cases a bad smell type might be introduced and removed for the same refactoring strategy. This situation happened with the Feature Envy bad smell, which may be removed and introduced by Extract Class refactoring operation.

We organized the remainder of this chapter as follows. Section 6.1 summarizes the contributions of our research. Finally, Section 6.2 suggests future work.

6.1 Contributions

With the results achieved in our dissertation, we present the contributions with our research as follows.

- a catalog presenting the relationship between bad smells and refactoring discussed in the literature and a contrast with Fowler's catalog;
- a catalog showing which bad smells tend to be introduced and removed by the automatic refactoring strategy;
- evidence to confirm or deny, or complement the findings discussed in the literature providing a contrast between the literature and Fowler's catalog with our empirical study, composing a more refined catalog;
- individual analyses using four perspectives to understand what the effects of automatic refactoring operation cause on bad smells of each one;
- identification of some tools that refactor through the detection of bad smells proposed by Fowler;

- a catalog showing a joint analysis with the data obtained in our empirical study and SLR presenting which bad smells tend to be introduced and removed by the refactoring strategy;
- relationships found between bad smells and refactorings not mentioned in the literature.

Besides that, the findings mentioned may assist developers in different ways, for instance:

- developers who want to apply refactoring strategies, automatically or manually, are better informed now of which bad smells may be introduced or removed by the refactoring operation;
- assist developers to take care of and perform the most efficient and robust refactoring tools so as not to introduce new bad smells in the source code.

6.2 Future Work

As future work related to SLR, we suggest investigate bad smells and refactoring strategies that Fowler did not propose, and the investigation of the new Fowler's catalog [33] in comparison with the SLR findings. We also suggest extend the SLR RQ2, focusing on identifying all refactoring tools present in the literature, that apply refactoring after detecting bad smells.

Related to the empirical study, first, in the construct of the data, we suggest extending this research to more systems and refactoring tools, different bad smells, and refactoring. Second, one may conduct this research method manually refactoring to evaluate the automated and manual refactoring operations' differences. Moreover finally, the development of a tool that considers the catalog created for an effective refactoring operation.

Bibliography

[1] Abid, C., Alizadeh, V., Kessentini, M., Ferreira, T. d. N., and Dig, D. (2020). 30 years of software refactoring research: A systematic literature review. *arXiv preprint arXiv:2007.02194*.

[2] Al Dallal, J. (2015). Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology*, 58:231–249.

[3] Alshayeb, M. (2011). The impact of refactoring to patterns on software quality attributes. *Arabian Journal for Science and Engineering*, pages 1241–1251.

[4] Azeem, M. I., Palomba, F., Shi, L., and Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *IST*, 108:115–138.

[5] Bafandeh Mayvan, B., Rasoolzadegan, A., and Javan Jafari, A. (2020). Bad smell detection using quality metrics and refactoring opportunities. *Journal of Software: Evolution and Process*, page e2255.

[6] Basit, W., Lodhi, F., and Bhatti, U. (2010). Extending refactoring guidelines to perform client and test code adaptation. In *Int. Conf. on Agile Soft. Development*, pages 1–13.

[7] Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., and Palomba, F. (2015). An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, pages 1–14.

[8] Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R. (2014). Recommending refactoring operations in large software systems. In *RSSE*, pages 387–419.

[9] Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., and De Lucia, A. (2013). Methodbook: Recommending move method refactorings via relational topic models. *TSE*, 40(7):671–694.

[10] Beecham, S., Baddoo, N., Hall, T., Robinson, H., and Sharp, H. (2008). Motivation in software engineering: A systematic literature review. *IST*, 50(9-10):860–878.

[11] 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 *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11.

[12] bin Ali, N. and Usman, M. (2019). A critical appraisal tool for systematic literature reviews in software engineering. *IST*, 112:48–50.

[13] Boshnakoska, D. and Mišev, A. (2010). Correlation between object-oriented metrics and refactoring. In *Int. Conf. on ICT Innovations*, pages 226–235.

[14] Bulychev, P. and Minea, M. (2008). Duplicate code detection using anti-unification. In *Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering*.

[15] 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 on Foundations of Software Engineering*, pages 465–475.

[16] Cedrim, D., Sousa, L., Garcia, A., and Gheyi, R. (2016). Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, pages 73–82.

[17] Chaparro, O., Bavota, G., Marcus, A., and Di Penta, M. (2014). On the impact of refactoring operations on code quality metrics. In *IEEE International Conference on Software Maintenance and Evolution*, pages 456–460.

[18] Chatzigeorgiou, A. and Manakos, A. (2014). Investigating the evolution of code smells in object-oriented systems. *Innovations in Syst. and Soft. Engineering*, 10(1):3–18.

[19] Counsell, S., Hassoun, Y., Loizou, G., and Najjar, R. (2006). Common refactorings, a dependency graph and some code smells: an empirical study of java oss. In *Proc. of the ISESE*, pages 288–296.

[20] Cruz, D., Figueiredo, E., and Santana, A. (2020). Detecting bad smells with machine learning algorithms: an empirical study. In *Proc. of Int. Conf. on Technical Debt (TechDebt).*

[21] da Silva Carvalho, L. P., Novais, R. L., do Nascimento Salvador, L., and de Mendonça Neto, M. G. (2017). An approach for semantically-enriched recommendation of refactorings based on the incidence of code smells. In *ICEIS*, pages 313–335.

[22] de Paulo Sobrinho, E. V., De Lucia, A., and de Almeida Maia, M. (2018). A systematic literature review on bad smells — 5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering.* ISSN 2326-3881.

[23] Du Bois, B. and Mens, T. (2003). Describing the impact of refactoring on internal program quality. In *International Workshop on Evolution of Large-scale Industrial Software Applications*, pages 37–48.

[24] Eposhi, A., Oizumi, W., Garcia, A., Sousa, L., Oliveira, R., and Oliveira, A. (2019). Removal of design problems through refactorings: are we looking at the right symptoms? In *IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 148–153.

[25] Erich Gamma, Richard Helm, R. J. and Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. *Addison Wesley Longman Publishing.*

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

[27] Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L., and Oizumi, W. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology.*

[28] Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A review-based comparative study of bad smell detection tools. In *Proc. of the 20th EASE*, pages 1–12.

[29] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. (2012). Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, pages 2241–2260.

[30] Fontana, F. A., Ferme, V., Zanoni, M., and Roveda, R. (2015a). Towards a prioritization of code debt: A code smell intensity index. In *IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 16–24.

[31] Fontana, F. A. and Spinelli, S. (2011). Impact of refactoring on quality code evaluation. In *Proceedings of the 4th Workshop on Refactoring Tools*, pages 37–40.

[32] Fontana, F. A., Zanoni, M., and Zanoni, F. (2015b). A duplicated code refactoring advisor. In *Int. Conf. on Agile Soft. Development*, pages 3–14.

[33] Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional. 2nd Edition.

[34] Fowler, M., Beck, K., Brant, J., and Opdyke, W. (1999). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[35] Ghannem, A., El Boussaidi, G., and Kessentini, M. (2014). Model refactoring using examples: a search-based approach. *JSEP*, pages 692–713.

[36] Gligoric, M., Behrang, F., Li, Y., Overbey, J., Hafiz, M., and Marinov, D. (2013). Systematic testing of refactoring engines on real software projects. In *ECOOP*.

[37] Gupta, V., Kapur, P. K., and Kumar, D. (2016). Modelling and measuring code smells in enterprise applications using tism and two-way assessment. *Int. Journal of Syst. Assurance Eng. and Management*, 7(3):332–340.

[38] Haas, R. and Hummel, B. (2017). Learning to rank extract method refactoring suggestions for long methods. In *Int. Conf. on Soft. Quality*, pages 45–56.

[39] Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. (2011). A systematic literature review on fault prediction performance in software engineering. *TSE*, 38(6):1276–1304.

[40] Hecht, G., Benomar, O., Rouvoy, R., Moha, N., and Duchien, L. (2015). Tracking the software quality of android applications along their evolution (t). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 236–247.

[41] Higo, Y., Kamiya, T., Kusumoto, S., and Inoue, K. (2004). Refactoring support based on code clone analysis. In *Int. Conf. on Product Focused Soft. Proc. Improvement*, pages 220–233.

[42] Jiau, H. C., Mar, L. W., and Chen, J. C. (2013). Obey: Optimal batched refactoring plan execution for class responsibility redistribution. *TSE*, pages 1245–1263.

[43] Kádár, I., Hegedűs, P., Ferenc, R., and Gyimóthy, T. (2016). Assessment of the code refactoring dataset regarding the maintainability of methods. In *International Conference on Computational Science and Its Applications*, pages 610–624.

[44] Kaur, S. and Singh, P. (2019). How does object-oriented code refactoring influence software quality? research landscape and challenges. *Journal of Systems and Software*, 157:110394.

[45] Kim, M., Zimmermann, T., and Nagappan, N. (2014). An empirical study of refactoring challenges and benefits at microsoft. *TSE*, 40(7):633–649. ISSN 2326-3881.

[46] Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. *In Technical Report, Ver. 2.3 (EBSE).*

[47] Lacerda, G., Petrillo, F., Pimenta, M., and Guéhéneuc, Y. G. (2020). Code smells and refactoring: a tertiary systematic review of challenges and observations. *Journal of Systems and Software*, page 110610.

[48] Liu, H., Wu, Y., Liu, W., Liu, Q., and Li, C. (2016). Domino effect: Move more methods once a method is moved. In *23rd SANER*, volume 1, pages 1–12.

[49] Liu, W., Hu, Z.-g., Liu, H.-t., and Yang, L. (2014). Automated pattern-directed refactoring for complex conditional statements. *Journal of Central South University*, 21(5):1935–1945.

[50] Liu, W. and Liu, H. (2016). Major motivations for extract method refactorings: analysis based on interviews and change histories. *Front. of Com. Science*, 10(4):644–656.

[51] Lozano, A. and Wermelinger, M. (2010). Tracking clones' imprint. In *Proceedings of the 4th International Workshop on Software Clones*, pages 65–72.

[52] Mahmoud, A. and Niu, N. (2014). Supporting requirements to code traceability through refactoring. *RE*, 19(3):309–329.

[53] Misbhauddin, M. and Alshayeb, M. (2015). Uml model refactoring: a systematic literature review. *ESE*, 20(1):206–251.

[54] Mkaouer, M. W., Kessentini, M., Bechikh, S., Cinnéide, M. Ó., and Deb, K. (2016). On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545.

[55] Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2009). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, pages 20–36.

[56] Oizumi, W., Sousa, L., Oliveira, A., Garcia, A., Agbachi, A. B., Oliveira, R., and Lucena, C. (2018). On the identification of design problems in stinky code: experiences and tool support. *Journal of the Brazilian Computer Society*, page 13.

[57] Oizumi, W. N., Garcia, A. F., Colanzi, T. E., Ferreira, M., and Staa, A. V. (2015). On the relationship of code-anomaly agglomerations and architectural problems. *Journal of Software Engineering Research and Development*, 3(1):1–22.

[58] Oliveira, J., Viggiato, M., Santos, M. F., Figueiredo, E., and Marques-Neto, H. (2018). An empirical study on the impact of android code smells on resource usage. In *SEKE*, pages 314–313.

[59] Ouni, A., Kessentini, M., Sahraoui, H., and Boukadoum, M. (2013). Maintainability defects detection and correction: a multi-objective approach. *ASE*, 20(1):47–79.

[60] Ouni, A., Kessentini, M., Ó Cinnéide, M., Sahraoui, H., Deb, K., and Inoue, K. (2017). More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *JSEP*, pages 1843–1863.

[61] Paiva, T., Damasceno, A., Figueiredo, E., and Sant'Anna, C. (2017). On the evaluation of code smells and detection tools. *JSERD*, 5(1):7.

[62] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., and De Lucia, A. (2018). On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. In *Proc. of the 40th ICSE*, pages 482–482.

[63] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. (2014). Mining version histories for detecting code smells. *TSE*, 41(5):462–489.

[64] Peldszus, S., Kulcsár, G., Lochau, M., and Schulze, S. (2016). Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 578–589.

[65] Pietrzak, B. and Walter, B. (2006). Leveraging code smell detection with inter-smell relations. In *Int. Conf. on Extreme Programming and Agile Proc. in Soft. Engineering*, pages 75–84.

[66] Rahman, M. M., Riyadh, R. R., Khaled, S. M., Satter, A., and Rahman, M. R. (2018). Mmruc3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design. *SPE*, 48(9):1560–1587.

[67] Sales, V., Terra, R., Miranda, L. F., and Valente, M. T. (2013). Recommending move method refactorings using dependency sets. In *20th WCRE*, pages 232–241.

[68] Santana, A., Cruz, D., and Figueiredo, E. (2021). An exploratory study on the identification and evaluation of bad smell agglomerations. In *Proc. of the 36th Symposium On Applied Computing (ACM SAC)*.

[69] Sehgal, R., Mehrotra, D., and Bala, M. (2017). Analysis of code smell to quantify the refactoring. *Int. Journal of Syst. Assurance Eng. and Management*, 8(2):1750–1761.

[70] Sharma, T., Fragkoulis, M., and Spinellis, D. (2016). Does your configuration code smell? In *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 189–200.

[71] Sharma, T., Singh, P., and Spinellis, D. (2020). An empirical investigation on the relationship between design and architecture smells. *EMSE*.

[72] Shomrat, M. and Feldman, Y. A. (2013). Detecting refactored clones. In Castagna, G., editor, ECOOP, pages 502–526.

[73] Silva, D., Terra, R., and Valente, M. T. (2014). Recommending automated extract method refactorings. In *ICPC*, pages 146–156.

[74] Silva, D. and Valente, M. T. (2017). Refdiff: Detecting refactorings in version histories. In *MSR*, pages 269–279.

[75] Singh, S. and Kaur, S. (2018a). A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*, pages 2129–2151.

[76] Singh, S. and Kaur, S. (2018b). A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*.

[77] Sousa, B. L., Bigonha, M. A., and Ferreira, K. A. (2018). A systematic literature mapping on the relationship between design patterns and bad smells. In *Proc. of the 33rd Annual ACM SAC*, pages 1528–1535.

[78] Tahir, A., Yamashita, A., Licorish, S., Dietrich, J., and Counsell, S. (2018). Can you tell me if it smells?: A study on how developers discuss code smells and anti-patterns in stack overflow. In *Proc. of the 22nd EASE*, pages 68–78.

[79] Tavares, C. S., Ferreira, F., and Figueiredo, E. (2018). A systematic mapping of literature on software refactoring tools. In *Proc. of the XIV SBSI*, page 11.

[80] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). Qualitas corpus: A curated collection of java code for empirical studies. In *Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345.

[81] Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *TSE*, 35(3):347–367.

[82] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering*, pages 403–414.

[83] Vale, G., Figueiredo, E., Abílio, R., and Costa, H. (2014). Bad smells in software product lines: A systematic review. In *Eighth SBCARS*, pages 84–94.

[84] Vidal, S. A., Marcos, C., and Díaz-Pace, J. A. (2016). An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23(3):501–532.

[85] Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 38.

[86] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

[87] Xia, X., Wan, Z., Kochhar, P. S., and Lo, D. (2019). How practitioners perceive coding proficiency. In *Proc. of the 41st ICSE*, pages 924–935.

[88] Zhang, M., Hall, T., and Baddoo, N. (2011). Code bad smells: a review of current knowledge. *Journal of Soft. Maint. and Evol.: Research and Practice*, 23(3):179–202.

[89] Zhang, M., Hall, T., Baddoo, N., and Wernick, P. (2008). Do bad smells indicate "trouble" in code? In *Proc. of Workshop on Defects in Large Software Systems*, pages 43–44.

Appendix A

Complementary Data of the SLR

In this appendix, we present complementary data that were documented in our Systematic Literature Review (SLR). Table A.1 presents the events where the papers found in our SLR were published. The first column shows the name of the publication event and the second column shows the number of papers that were found.

Table A.1: Published Events

Event Name	**Papers Returned**
Frontiers of Computer Science	01
IEEE Transactions on Software Engineering	03
Innovations Syst Softw Eng	01
International Conference on Agile Software Development (XP)	02
International Conference on Enterprise Information Systems (ICEIS)	01
International Conference on Product Focused Software Process Improvement (PROFES)	01
International Conference on Software Analysis, Evolution, and Reengineering	01
International Conference on Software Quality (SWQD)	01
International Journal of System Assurance Engineering and Management	01
International Symposium on Empirical Software Engineering	01
Joint Meeting on Foundations of Software Engineering	01
Journal of Software: Practice and Experience	01
Journal of Systems and Software	01
Recommendation Systems in Software Engineering	01
Requirements Eng	01
Simpósio Brasileiro de Engenharia de Software (SBES)	01
Working Conference on Reverse Engineering (WCRE)	01

Table A.2 presents the name of the tools mentioned in the papers found in our SLR. The first column shows the name of the presented tool. The second and third columns show the used refactoring and bad smell detection tools, respectively. Finally, in the last column, we present the reference of the original paper.

Table A.2: Tools

Tool	**Refactoring Tool Used**	**Bad Smell Tool Used**	**Reference**
	Refactoring Miner	Own Metric Developed	[15]
	JDeodorant	JDeodorant and DECOR	[69]
	Ref-Finder	Own Metric Developed	[7]
	JDeodorant	JDeodorant	[18]
	ARIES and JDeodorant	JDeodorant	[8]
	Ref-Detector	Metrics to Detect Smell	[16]
CCShaper	Integrated	CCFinder	[41]
DCRA	Integrated	NiCad	[32]
Extract Method Detector	Integrated	Integrated	[50]
HIST		Integrated	[63]
JMove	Integrated	Integrated	[67]
Liu's Approach	Integrated	Integrated	[48]
Methodbook	Integrated	Integrated	[9]
MMRUC3	Integrated	Integrated	[66]
RESYS	OSORE	OCEAN	[21]
Tsantalis's Methodology	Integrated	Integrated	[81]