Pontifícia Universidade Católica
do Rio de Janeiro

Daniel Tenorio Martins de Oliveira

Towards customizing smell detection and refactorings

Dissertação de Mestrado

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

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
May 2020

Pontifícia Universidade Católica
do Rio de Janeiro

Daniel Tenorio Martins de Oliveira

Towards customizing smell detection and refactorings

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

Prof. Alessandro Fabricio Garcia
Advisor
Departamento de Informática – PUC-Rio

Prof. Marcos Kalinowski
Departamento de Informática – PUC-Rio

Prof.ª Simone Diniz Junqueira Barbosa
Departamento de Informática – PUC-Rio

Rio de Janeiro, May the 4th, 2020

Daniel Tenorio Martins de Oliveira

I am an MSc student in Computer Science at Pontifical Catholic University of Rio de Janeiro (PUC-Rio, Brazil) and a part-time software engineer for Tecgraf Institute/PUC-Rio. I have a bachelor's degree in Computer Science from the Federal University of Alagoas (UFAL). Since my graduation, I always aimed to collaborate in international projects on Software Engineering, in special with relevant European Universities such as University of Coimbra, University of Florence, and University of College London. I also actively participated as researcher for projects in partnership with large companies, as BlackBerry. My current research is focuses on providing developers with better automated support for code smell detection and batch refactorings. As a result of my dedication to research, I submitted accepted papers for relevant vehicles, such as International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE) and International Workshop on Refactoring (IWoR).

Bibliographic data

Tenorio Martins de Oliveira, Daniel

Towards customizing smell detection and refactorings / Daniel Tenorio Martins de Oliveira; advisor: Alessandro Fabricio Garcia. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2020.

v., 128 f: il. color. ; 30 cm

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

Inclui bibliografia

1. Informática – Teses. 2. Engenharia de software – Teses. 3. Refatoração;. 4. Manutenibilidade;. 5. Customização de refatoração;. I. Fabricio Garcia, Alessandro. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I thank all the professors of the course, who were important for my academic life and in the development of this dissertation. In especial to Prof. Dr. Alessandro Garcia, for his advice and for allowing me to have the opportunity to advance in this scientific world. I also thank my parents and brothers and a special cousin who, with great care and support, worked hard to take me to this stage of my life. I also thank my girlfriend, for always staying by my side when writing this dissertation. I thank my friends from both university and childhood, who have always been by my side.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Tenorio Martins de Oliveira, Daniel; Fabricio Garcia, Alessandro. **Towards customizing smell detection and refactorings**. Rio de Janeiro, 2020. 128p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Code smells are poor structures that harm software maintenance. Therefore, code smells should be detected and removed, through refactoring, early in the software lifecycle. Refactoring consists of a sequence of code modifications that aim to improve software maintenance by removing or mitigating poor code structures. However, the strategies for detecting and refactoring smells are subjective. Even developers working on the same software may diverge on their opinions about the existence of a smell. In fact, this divergence is mostly influenced by the developer's knowledge, including the system's design and the analyzed source code. As a consequence, the same divergence affects the application of the corresponding refactorings. Therefore, there is a need to support the customization of smell detection and refactoring based on the developer's knowledge. The developer is who, after all, becomes the decision maker on confirming the harmfulness of a smelly structure and how to refactor it out. In order to address this issue, we split our research in 3 steps: (i) how to customize smell detection strategies? (ii) whether and how often developers customize their refactorings? and (iii) how to support refactoring customization? In the first step, we evaluated the use of machine learning techniques for properly customizing smell detection for each developer. Second, we investigated how developers customize refactorings by analyzing their code modifications while applying certain refactoring types. Besides, we also discussed how these customizations are related to the introduction, removal or mitigation of smells, and whether they are currently supported by Eclipse, a popular IDE. Third, we proposed an approach that allows the application of custom refactoring. Our results indicated that machine learning techniques are able to efficiently capture the developer's knowledge and achieve high smell detection accuracy. Also, even though developers frequently customize refactorings, their customizations are often not supported by Eclipse. To make it worse, complex customizations, which are manually performed, tend to reduce the positive effect of the refactoring. Therefore, our contributions serve as a basis for improving tool support for: (i) customized detection of smells considering the developer's knowledge, and (ii) application of customized refactoring.

Keywords

Refactoring; Maintainability; Refactoring Customization.

Resumo

Tenorio Martins de Oliveira, Daniel; Fabricio Garcia, Alessandro. **Rumo a customização na detecção de smell e na refatoração**. Rio de Janeiro, 2020. 128p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Code smells são estruturas pobres que prejudicam a manutenção do sistema. Sendo assim, code smells devem ser detectados e removidos, através de refatoração, no começo do ciclo de vida do software. Refatoração consiste em modificações no código que visam melhorar a manutenção do software, removendo ou mitigando estruturas pobres. Contudo, as estratégias de detecção e refatoração de smells são subjetivas. Isto é, desenvolvedores trabalhando no mesmo sistema podem divergir acerca da existência de um smell. Essa divergência é influenciada pelo conhecimento do desenvolvedor, incluindo o design do sistema e o código analisado. Como consequência, essa divergência afeta também a aplicação das refatorações. Assim, é preciso customizar a detecção de smell e refatoração a partir do conhecimento dos desenvolvedores. Afinal, o desenvolvedor é quem confirma a nocividade de um smell e define como refatorá-lo. Para isso, decompomos nossa pesquisa em 3 passos: (i) como customizar estratégias de detecção de smells?, (ii) se e com que frequência os desenvolvedores customizam suas refatorações? e (iii) como dar suporte a customização de refatoração?. No primeiro passo avaliamos as técnicas de aprendizagem de máquina quanto a capacidade de customizar sua detecção para cada desenvolvedor. Segundo, nós investigamos como desenvolvedores customizam refatorações, analisando suas modificações de código enquanto aplicam certos tipos de refatoração. Além disso, nós também discutimos como essas customizações estão relacionadas com a inserção, remoção ou mitigação de smells e se são apoiados pelo Eclipse. Terceiro, nós propusemos uma abordagem que permite a aplicação de refatorações customizadas. Nossos resultados indicaram que as técnicas de aprendizagem de máquina são capazes de capturar o conhecimento do desenvolvedor e obter alta acurácia detectando smells. Além disso, desenvolvedores frequentemente customizam refatorações que não são totalmente suportadas pelo Eclipse. Para piorar, customizações complexas, geralmente manuais, tendem a reduzir o efeito positivo da refatoração. Portanto, nossos resultados servem como base para melhorar o suporte de ferramentas: a (i) detecção customizada de smells, levando em consideração o conhecimento do desenvolvedor e (ii) a aplicação de refatoração customizada.

Palavras-chave

Refatoração; Manutenibilidade; Customização de refatoração;

Table of contents

List of figures

List of tables

1
Overall Introduction

Software developers need to properly read and understand the code structures for proper maintenance (1). Therefore, developers often try to keep the code clear and easy to understand. However, developers make decisions that (un)intentionally introduce poor structures making the code maintenance harder (2). These poor structures, usually called code smells, are harmful to software quality. A code smell is a surface indication that usually corresponds to a deeper problem in the system (1)). Some code smells are catalogued in previous work(e.g Fowler's catalog (1)) and their detection is supported by industrial and academic tools. Due to their harmfulness, code smells should ideally be removed as soon as they are detected (3, 4, 5).

Although code smell removal is necessary, the strategies to detect and remove code smells are quite subjective. This subjectivity makes these strategies hard to be defined. Indeed, different developers working on the same software may have different knowledge about code smells (6). This knowledge varies according to the developer's experience, individual skills and awareness regarding the source code being analyzed. Developers believe that the decision on removing code smells should be thoroughly made, avoiding side effects for the maintenance of the source code (7). In fact, developers are who has the knowledge to make those decisions and often this knowledge prevails on confirming the harmfulness of a smelly structure. Code smells are often detected as soon as developers conclude the program needs refactoring. A refactoring consists of a set of code modifications often aimed at removing poor structure and improving software maintainability (1, 8, 9). Before applying refactoring to remove a code smell, developers need to make subjective decisions such as the ones described below.

(i) *Decisions on the existence of code smells.* Consider the code smell Long Method, *i.e.*, a method that is too long and tries to do too much. Even though it is easy to understand the concept of this smell, it is hard to define how long the method needs to be or how many responsibilities the method needs to fulfill to be considered a Long Method. Developers are in charge of inspecting a possible Long Method and, based on their knowledge, to judge whether the method is indeed a Long Method or not. Only the

developers know what factors should be considered when detecting smells in their program. Indeed, developers has always the final word when confirming and removing code smells. The literature proposes automated metric-based strategies aiming to distinguish code more likely to have smells (10, 11). However, these strategies do not take into account the knowledge of each developer even though developers usually disagree regarding the existence of a smell (6).

Due to the importance of developers on code smell detection, it is necessary to customize the detection considering their knowledge. Customization of smell detection consists of adapting an existing strategy used in such detection. Rigid (i.e., not customized) strategies may suggest code smells that are not interesting to a specific developer through frequent irrelevant warnings. These warnings can hinder his concentration or camouflage smells that are considered more harmful according to this developer. However, there is little understanding in the literature concerning customization of smell detection. Even worse, existing strategies to support code smell detection provide limited support for its customization. Thus, the detection of code smells in specific contexts, inevitably, require the participation of a developer to confirm the existence of a smell (12, 13, 14).

(ii) *Decisions on the application of program refactoring.* Once a relevant code smell is detected, developers usually apply refactorings to remove it (1). Each type of smell needs different strategies to be removed depending on how it manifests in a program and the code elements affected. For instance, let us consider an *Extract Method* (EM) refactoring, which is the most common refactoring according to previous studies (2, 15). Fowler states an EM consists on creating a new method based on statements extracted from an existing method (16). Developers can apply an EM to remove Long Methods, once this refactoring extracts statements from a method. This extraction reduces the method's size and possibly the number of responsibilities fulfilled by the method.

Also, EM should be customized to be applicable in different contexts. A refactoring customization consists of adapting an existing refactoring to tailor it to the context where it will be applied. For instance, an EM can be customized to extract statements from different methods once: (i) these methods may have a Duplicated Code smell or (ii) a responsibility is scattered in two methods and, as a consequence, this responsibility should be modularized in a single method. In this context, more than one method will have the code extracted, but only one method needs to be created. Thus, the developer's knowledge about the software context is fundamental to decide how to best

customize a particular refactoring to remove a specific smell.

Once the empirical evidence of how developers customize refactoring in practice is scarce, these tools may not be adequate to proper support these refactorings (17, 18, 19, 20, 21). Indeed, they do not allow developers to properly create their own refactoring customization. In this way, developers are instead restricted to tailor modifications only through very basic configurations provided by these tools, which often do not satisfy their needs (17, 20). Due to this limitation, developers frequently apply these refactorings manually (17, 20) based on their knowledge. For instance, using refactoring tools, developers are not enable to always choose when to extract code from more than one method when applying *Extract Method*. In some tools, such as the provided by Eclipse[1], developers can choose to remove statements from more than one method only if Eclipse recognize that these methods have the same code statements.

1.1 Problem Statement and Related Work

This section discusses related work and provides statements of our three research problems.

Limited understanding about smell detection customization taking into account developer knowledge. As mentioned earlier, the previous work proposes strategies to detect code smells, *e.g.* (2, 10, 11, 22). The use of software metrics in strategies to detect code smells is quite common (23, 24). A detection strategy consists in firstly compute some measures from the source code. Each smell requires the computation of different measures. If a predefined set of measures exceed their established thresholds (10), then there is possibly a code smell. Using this strategy is possible to customize the smell detection by changing the set of metrics and the threshold values. The developer is usually responsible for adjusting such metrics and thresholds. The adjustment is based on the previous developer's knowledge on detecting the same smells and the affected code. This manual adjustment is time-consuming (25) and makes the customization of these strategies difficult and error-prone, especially for less experienced developers.

The use of machine learning techniques (ML techniques) (26) is a strategy to address this issue and automate the customization of code smell detection. Due to their ability to learn by examples, some machine learning techniques can be considered as a promising way to detect smells. These techniques can use a set of examples to customize the definition of such detection strategies. In other words, these techniques define the set of metrics and thresholds to

[1]https://www.eclipse.org/

classify whether a piece of code has smell or not. Recent studies (27, 28, 29, 30) have analyzed the use of ML techniques aiming to detect code smells.

However, these studies usually assess a single ML technique. Certain ML techniques may outperform others in the detection of each smell type. More importantly, there is still little understanding about how the machine learning techniques can customize the detection of code smells based on developers' knowledge. This customization is of paramount importance; otherwise, developers are likely to reject the smell candidates. The ability to customize smell detection also allows the adaptation of strategies to different quality standards required by companies and their developers. Besides, customizable detection can identify and report to developers only smells they are interested in.

> **Problem 1:** There is still little understanding about whether ML techniques can properly customize code smell detection taking into account the developer's knowledge.

Empirical evidence of how developers customize refactoring is quite scarce. The refactoring practice is often studied, especially its impact on the software structure (31). There is also a few studies investigating the refactorings required in certain software projects (32, 33, 34). However, there is little understanding about how developers customize refactorings. That is, the literature usually treats each refactoring as a fixed set of code modifications that is already cataloged, such as those modifications described for certain refactoring types presented by Fowler (1).

In order to understand customized refactorings, we need to study the refactoring as a mutable set of code modifications. This set of modifications likely varies across the context where a refactoring type is applied. A recent study (35) observed the code modifications performed in the project when applying certain refactoring types. However, the study does not investigate how developers customize the modifications of a refactoring type. The lack understanding about typical patterns of modifications for certain refactoring types impairs the advancement of refactoring tooling support. This limitation is also observed in other studies that relate refactorings with code modifications (36, 37).

It is important to understand how the refactorings are customized by developers. Otherwise, refactoring tools will be misaligned with developers practices and will not provide adequate support to custom refactorings. In particular, existing tools could provide developers with previously-defined

custom refactorings to be selected, further adapted or extended and applied by developers in their contexts.

> **Problem 2:** There is limited empirical understanding regarding refactoring customization in the literature.

Current refactoring tools do not provide suitable support for refactoring customization. As mentioned above, there is a lack of empirical understanding about refactoring customization. This lack of understanding turns the creation of suitable tools to support refactoring customization a hard task. Despite some IDEs, such as Eclipse[2], allowing developers to create custom refactorings, the customization process requires developers to know a specific description language. Alternatively, these IDEs also provide automated tools for applying refactorings. However, these automated tools only apply a limited number of simple refactorings. Besides, these automated tools restrict to tailor these refactorings only through very basic configurations. As consequence, developers are not allowed to properly create their own custom refactoring.

Indeed, developers prefer to apply refactorings manually instead of using automated refactoring tools despite all the advantages that automated refactoring has over manual refactoring (17, 19). One of the possible reasons is that the use of automated refactoring tools overly constrain the possibilities for customization. In addition, developers are not in the control of all modifications made by these tools (9, 17). Also, developers do not consider these tools flexible enough to use in specific contexts. (17).

> **Problem 3:** Automated refactoring tools do not allow developers to properly customize refactorings.

1.2 Main Contributions

In order to address the research problems (Section 1.1), this research focused on performing retrospective studies using data from several open-source software projects. Such studies aimed at improving the empirical understanding of how developers perform smell detection and refactoring. Our investigation was divided into the three following steps. For each step, we presented the corresponding contribution emerging at that research stage.

[2]https://www.eclipse.org/

Step 1: Customization of smell detection. We performed two complementary studies to evaluate machine learning techniques for customizing smell detection. Our studies relied on hundred of smells manifesting in popular open-source software projects. In particular, we compared several machine learning techniques considering their ability to customize the detection of smells taking into account the developer's knowledge. Developers are often responsible for performing the detection of code smells in their projects. In this way, customization is of paramount importance; otherwise, developers are likely to reject the smell candidates detected by the machine learning techniques. We analyzed which ML technique correctly detected well-know smell types (effectiveness) and with the fewest examples needed for training (efficiency). Then, we observed which ML technique is more effective and efficient to detect each smell type. This observation took into account different developer's opinions about the existence of a smell in the affected program.

We observed that all the analyzed techniques are sensitive to the developers' knowledge and the number of training examples. Besides, Random Forest was the technique with the highest effectiveness and efficiency. This technique needed fewer examples than any other ML technique for most types of smell types. Due to the importance of the developer's knowledge on the detection of code smells, our results shed light on the design of future studies aimed at revealing further knowledge involving developers' expertise and smell detection.

Contribution 1: Evaluation of the use of machine learning techniques to customize smell detection based on the developer's knowledge.

Step 2: Customization of refactoring. We investigated how developers customize refactorings in practice. We focused the investigation on four of the most frequent refactoring types, namely *Extract Method*, *Inline Method*, *Pull Up Method*, and *Move Method*. We identified refactoring occurrences of these refactoring types in open-source software projects using the Refactoring Miner tool (15, 38). For each identified refactoring, we compared the source code before and after the occurrence of such refactoring. We observed the code modifications performed along with such refactoring. Then, we catalogued the most frequently observed code modifications for each refactoring type. In this way, we investigated whether developers often have to customize such refactorings by including or discarding modifications that are tailored to their program's needs

> **Contribution 2:** A Catalog of modifications applied by developers when applying specific refactoring types.

Then, we identified the most frequent modification patterns in customized refactorings. We also analyzed the impact of these patterns on removing code smells. For this analysis, we investigated whether there are interesting customized modification patterns that tend to (i) reduce the occurrence of a particular type of smell, (ii) reduce the intensity of the smell, that is, improve the code metrics making the smell less harmful, and adversely (iii) increase the occurrence of a particular type of smell. Our methodology to detect customized refactoring is more rigorous than the methodology used elsewhere (39). For instance, we also analyzed all fine-grained modifications affecting both the refactored elements and their clients.

> **Contribution 3:** A catalog of customized refactorings applied by developers in practice and the impact of these customizations on code smells.

Our results revealed that developers often apply recurring modifications in their customized refactorings that are not covered by Fowler's catalog. Finally, our study also provides recurring patterns of customized refactorings that can help one: (i) to better understand the refactoring customization needs, and (ii) to further improve tooling support for customized refactoring

Step 3: Refactoring tool support. Finally, we investigated whether the tool for applying refactoring provided by Eclipse properly supports the application of customized refactorings. We chose Eclipse because it is a very popular environment for Java development. Besides, Eclipse is frequently used in literature on automated software refactoring *e.g.* (40, 41). For that, we observed the source code associated with the most frequent patterns of each refactoring type. We minimally adapted the code to be reproducible in our Eclipse environment. Then, we manually invoked the automated tool in order to reproduce the refactoring applied by the developer in the software project. Finally, we listed which code modifications could not be reproduced using the Eclipse's refactoring tool. Our results revealed that developers often apply recurring modifications in their customized refactorings that are not fully supported by the existing refactoring tool provided by Eclipse. Based on this investigation, we listed which customizations applied by developers in practice do not have adequate support provided by Eclipse's refactoring tool. Due

to this limitation, developers would have to apply part of the customization manually, which is a time-consuming and error-prone activity.

> **Contribution 4:** A catalog of recurring modifications present in customized refactorings that are not supported by the automated refactoring tool provided by Eclipse.

Finally, based on empirical evidence derived from our studies, we presented the prototype of a more flexible tool, which aims to satisfy the customizations applied by developers in practice. The approach allows a developer to: (i) compose an individual refactoring according to his context's needs, and (ii) reuse the custom refactorings in similar contexts. For that, we split each refactoring into a set of code modifications. Then, the developer can add or remove other code modifications that compose each refactoring. In this way, developers will able to change the formerly defined behavior of a refactoring,

> **Contribution 5:** A prototype of an automated refactoring tool that provides support for refactoring customization.

1.3 Dissertation Outline

The remainder of this dissertation, which is a compilation of technical papers (accepted or under submission), is organized as follows.

Chapter 2 presents a study to evaluate machine learning techniques for customizing smell detection. In this study, we analyzed which ML technique correctly detected well-know smell types and with the fewest examples needed for training when taking into account developers' knowledge. This study consists of the paper *"On the Sensitivity of Machine Learning Techniques to Detect Developer-sensitive Smells"* (42), under submission to a top international journal in Software Engineering.

Chapter 3 presents a complementary study to evaluate machine learning techniques for customizing smell detection. Similar to the study in Chapter 1, in this study we also evaluated the use of ML techniques to detect code smells taking into account developers' knowledge. However, we used a different dataset. In this dataset, we considered only smells that was refactored by developers associated with the analyzed projects. This study consists of the paper *"Assessing Machine Learning techniques on Code Smell Detection"* (43) being submitted to Brazilian Symposium on Software Engineering (SBES) in May 2020.

Chapter 4 presents a retrospective study to understand how developers customize refactorings in practice. In this study, we investigated 13 projects, from which we identified and analyzed 1,162 refactoring instances. We focused our analysis on four of the most frequent refactoring types, namely *Extract Method*, *Inline Method*,*Pull Up Method*, and *Move Method*. This study consists of the paper *"How Do Developers Customize Refactoring in Practice?"* (44) being submitted to the 35th International Conference on Automated Software Engineering (ASE) in May 2020.

Chapter 5 presents a flexible approach for refactorings tools. This approach allows a developer to: (i) compose a custom refactoring according to his context's needs, and (ii) reuse the custom refactorings in similar contexts. This approach consists of the paper *"On the Customization of Batch Refactoring"* (45) published in the 3th International Workshop on Refactoring in 2019, co-located with the International Conference on Software Engineering (ICSE 2019), which brings together the experts on software refactoring around the world.

Chapter 6 summarizes the conclusions of our work, presenting the main contributions to the state-of-art and the state-of-practice as well as future work.

2
Customization of Code Smell Detection

Code smells are considered symptoms of poor design and implementation choices, which make the software system hard to maintain and evolve (46). Developers are often responsible to detect smells in their software projects. However, due to the abstract nature of the code smell definitions, their detection becomes a challenging task. To make worse, different developers may have different opinions about the existence of a smell in a code fragment. The developer's opinion about the existence of a smell is influenced by the experience in software development and the knowledge of the analyzed project. Thus, code smell detection techniques should be customized taking into account developers' knowledge, otherwise, developers may not agree with the detected code smell existence by the techniques. Besides, the code smell detection can be customized to adhere to projects specific quality standards.

The use of ML techniques to customize smell detection is considered a promising way to achieve accurate results (27, 47). However, even there are many studies used machine learning techniques to perform their detection (27, 28, 29, 30, 47, 48, 49), there is little knowledge of how these ML techniques are sensitive to developers' knowledge. This sensitivity comes from the fact that the smells used to train the ML techniques are usually detected by developers. Thus, the training set can be influenced based on the knowledge of code smells that these developers have. Based on this influence, we decided to study how sensitive the ML techniques are to the developer's code smell knowledge. In this way, we can also observe how these ML techniques customize their detection strategy as they analyze smells detected by different developers.

This chapter presents the papers *"On the Sensitivity of Machine Learning Techniques to Detect Developer-sensitive Smells"* (42), under submission to a top international journal in Software Engineering. In this paper, we evaluated the use of ML techniques to detect code smells taking into account developers' knowledge. For that, we performed a broader study aimed at investigating the ability of ML techniques on detecting developer-sensitive smells. That is, we observed how the ML techniques customize their detection strategies based on different developers' knowledge about code smells.

On the Sensitivity of Machine Learning Techniques to Detect Developer-sensitive Smells

Daniel Oliveira[b], Filipe Falcão[a], Caio Barbosa[a], Baldoino Fonseca[a], Leonardo Souza[b], Alessandro Garcia[b], Márcio Ribeiro[a], Tiago Vieira[a], Evandro Costa[a]

[a]*Computing Institute, Federal University of Alagoas, Brazil*
[b]*Opus Research Group – LES, Informatics Dept., PUC-Rio, Brazil*

2.1 Introduction

Code smells are considered symptoms of poor design and implementation choices, which make the software system hard to evolve and maintain (46). Due to their harmfulness to software quality (3, 4, 5), code smell should be detected as early as possible to enable its removal. Unfortunately, several reasons make the code smell detection a challenging task. For instance, code smells are subjective in nature and, inevitably, require the participation of developers to detect them (12, 13, 14). However, a recent study (50) indicates a high divergence among developers about the existence of a same code smell into a code snippet. Hence, detecting code smells in practice is much harder than related studies usually discuss (51, 52, 53, 54, 55).

To make matters worse, definitions for some code smells are informal, ambiguous or insufficient to describe them precisely. For instance, let us consider the definition of *Long Method* (LM) code smell, which it is *a method that is too long and tries to do too much* (46). Although such definition states what a *Long Method* is, it does not describe what should be considered as "too long" neither what "to do too much" is. As a consequence, when a developer is focused on detecting *Long Methods*, he may face subjective questions, such as:

- How to define whether a method is long?
- How to define whether a method is doing too much?
- Is it possible to identify a *Long Method* solely based on the lines of code of a method?
- How many lines of code are required to characterize a method as *Long Method*?

Different developers working on the same code base may have different answers to these questions. As a consequence, while a developer may confirm a

code snippet as the host of a *Long Method*, other developers may not necessarily agree.Indeed, the literature discusses this divergence among developers (47). Thus, detecting smells taking into account the individual perception of each developer remains as a prevailing challenge.

Several studies (27, 28, 29, 30, 47, 48, 49, 56, 57) have analyzed the use of machine learning techniques (ML techniques) to identify smells. In a nutshell, the ML techniques require a training set containing code examples annotated by developers as smell or non-smell. From these training examples, the ML techniques generate smell detection models. Even though such studies indicate that ML techniques are a promising way to detect smells, there is little knowledge of how sensitive these ML techniques are on detecting smells. The sensitivity represents how the ML techniques' behavior varies to slight changes regarding the different perceptions of developers about the existence of code smells. The ML techniques' behavior change is assessed through two factors: (I) effectiveness: the capability that ML techniques have when detecting code smells for different developers, *i.e.*, high effectiveness indicates the ML techniques are able to detect correctly code smells from different perspectives; and (II) efficiency: the effectiveness variation as new code snippets validated by different developers are progressively considered by the ML techniques. High efficiency indicates the ML techniques need a low number of training example to reach high efficiency.

In this context, this paper reports a broader study aiming at investigating the sensitivity of seven ML techniques on detecting developer-sensitive smells. We assess these techniques based on their capability of detecting 10 different smell types in accordance with the individual perception of 63 developers. We performed our study through four main steps: (*Dataset*) for each smell type, developers evaluated the presence (or not) of code smells into 15 code snippets from industry-scale projects. Altogether, we collected and stored in a dataset 1,800 evaluations, which were used to evaluate the sensitivity of the ML techniques; (*Overall Effectiveness*) we evaluated the overall effectiveness of the ML techniques on detecting each one of the 10 smell types. We performed such evaluation without taking into account the developers' perceptions; (*ML techniques Dispersion*) we investigated the variation of the ML techniques effectiveness when they should identify developer-sensitive smells, *i.e.*, when they have to detect smells taking into account the individual perception of each developer. In addition, we analyzed the most disperse and effective ML techniques on detecting developer-sensitive smells; and, finally, (*ML techniques Efficiency*) we assessed the efficiency of the ML techniques by evaluating the effectiveness of each technique on detecting developer-sensitive smells whereas

we gradually increase the number of examples used to perform its training.

Our study led to five main findings:

- The *Random Forest* (RF) (58) and *Naive Bayes* (NB) (58) reached the highest overall effectiveness on detecting smells (seven out of 10). On the other hand, the *Support-vector Machine* (SVM) (58) presented the lowest overall effectiveness in six smell types (Section 2.3.1);
- The effectiveness of all the analyzed ML techniques were influenced by the developers' perceptions. As a consequence, the techniques could not reach an overall effectiveness above 0.8 in the vast majority of the cases analyzed (Section 2.3.2);
- While the SVM (58) is the most disperse technique, the *Random Forest* is less disperse one (Section 2.3.2);
- The *Random Forest* was the most effective on detecting developer-sensitive smells and the *JRip* (58) was the less effective one (Section 2.3.2);
- The *Random Forest* was able to detect developer-sensitive smells more effectively and with lower number of examples than any other analyzed technique in the cases of the *Data Class*, *Inappropriate Intimacy*, *Refused Bequest* and *God Class* (Section 2.3.3).

These findings suggest the increasing need for improving smell detection techniques by taking into account the individual perception of each developer. The remaining of this document is structured as follows. Section 2.2 describes the design of our study and the research questions. In Section 2.3 we present the results of the study and we answer the research questions. Section 2.4 details the threats of the study. Next, Section 2.5 presents the related work. Finally, Section 2.6 presents the conclusions observed in our study.

2.2 Study Design

Previous studies (*e.g*, (27, 47)) suggest that ML techniques are a promising way to identify code smells. However, the code smell detection involves the participation of developers who present a high divergence on defining if a code snippet is a smell or not (12, 13, 50, 59, 60). As a consequence, divergence among developers may considerably influence the set of code smells detected. In particular, such divergence may impact the effectiveness of code smell detection techniques based on machine learning, which depend of code smell examples previously annotated by developers to perform their training.

In this context, our study aims at investigating the sensitivity of ML techniques to detect code smells, *i.e.*, to investigate the effectiveness variation of each technique in terms of two factors: (i) the individual perceptions of developers about the presence of code smells; and (ii) the number of examples used to perform the training of the ML techniques.

Initially, we defined the research question **RQ1** aiming at investigating the overall effectiveness of ML techniques in detecting 10 smell types without considering none of these factors. Even though several studies (*e.g*, (27, 30, 47)) have investigated the overall effectiveness of ML techniques to detect code smells, each study evaluated a reduced number of smell types on a training set containing a large number of code smell examples annotated by few developers. In our study, for each one of the ten smell types analyzed, we performed the training of the ML techniques on a dataset containing only 15 smell examples annotated by 12 developers with different perceptions about the smell type analyzed.

> **RQ1.** *How effective are the ML techniques on detecting smells?*

Next, we defined the **RQ2** that investigates the effectiveness variation of a ML technique on detecting smells in accordance with the individual perceptions of each developer. The main motivation to this research question is the fact that developers have different backgrounds, experience and skills. These and other factors naturally lead developers to have different perceptions about the occurrence of a same code. As a consequence of this divergence among developers, ML techniques may present a variation in their effectiveness on detecting developer-sensitive smells (12, 13, 50, 59, 60).

> **RQ2**: *How disperse is the ML technique effectiveness on detecting developer-sensitive smells?*

Finally, we investigated the **RQ3** aiming at analyzing the efficiency of the ML techniques on detecting developer-sensitive smells, *i.e.*, how effective a ML technique detects developer-sensitive smells whereas we gradually increase the number of examples used to perform its training. Although ML techniques have been considered a promising way to detect code smells (*e.g*, (27, 30, 47)), these techniques require code smell examples annotated by developers to perform their training. However, the annotation of a large amount of examples may introduce an unfeasible additional effort to the developers. Hence, it is important to analyze the effectiveness variation of the ML techniques whereas we vary the number of examples used to perform the training of these techniques.

RQ3: *How efficient are the ML techniques on detecting developer-sensitive smells?*

2.2.1 Subjects and Projects

To perform our study, we recruited 63 developers from different companies and universities. These developers have different levels of experience in software development in Java. Besides, these developers also have previous experience on code smell detection in software projects, which had an emphasis on structural software quality.

After recruiting the developers, our next step was to select the types of smell to be analyzed. Table 2.1 describes all the selected smell types. We have chosen these smell types for two main reasons. First, they affect different scopes of a program, *i.e.*, classes, methods or parameters. Second, they have been investigated in previous studies about code smell detection (28, 29, 30, 48, 49).

Table 2.1: Types of Code Smells Investigated in this Study

Name	Description
God Class (GC)	Classes that tend to centralize the intelligence of the system.
Data Class (DCL)	Classes that have fields, getting and setting methods for the fields, and nothing else.
Long Method (LM)	Methods that are too long and try to do too much.
Feature Envy (FE)	Methods that use more attributes from other classes than from its own class, and use many attributes from few classes.
Message Chains (MC)	A object that calls another object, that requests yet another one, and so on.
Inappropriate Intimacy (II)	Classes that use internal fields and methods that don't belong to them.
Middle Main (MM)	Classes that delegate too much work to another classes and do nothing by herself.
Primitive Obsession (PO)	Using a lot of primitives as substitute for small objects.
Refused Bequest (RB)	Classes inherit from a superclass and don't use any of inherited functionality.
Speculative Generality (SG)	Unused classes, methods, fields or parameters created to future features that never get implemented.

Finally, we analyzed the source code of five open source Java projects: GanttProject[1] (2.0.10), Apache Xerces[2] (2.11.0), ArgoUML[3] (0.34), jEdit[4] (4.5.1) and Eclipse[5] (3.6.1). We selected such projects because they have been evaluated by existing smell detection techniques (23, 24, 48, 51, 54, 61, 62) and their source code contains a variety of suspicious code smells (47, 50) that enable the execution of our study.

[1]http://www.ganttproject.biz
[2]http://xerces.apache.org
[3]http://argouml.tigris.org
[4]http://www.jedit.org
[5]http://eclipse.org

2.2.2 Machine Learning Techniques

The seven chosen ML techniques to be evaluated are described below:

NaiveBayes: Probabilistic classifier based on the application of Bayes' theorem (63). This technique is highly scalable and is completely disregards the correlation between the variables in the training set. Its main idea describes the probability of an event based on prior knowledge of conditions that might be related to this event.

Support Vector Machine (SVM): Implementation of integrated software for the classification of support vectors (64) that analyzes the data used for classification and regression analysis. SVM assigns new examples to one of the two categories introduced in the training set, making it a non-probabilistic binary linear classifier. In order to make this classification, SVM creates classification models that are a representation of examples as points in space.These points are mapped in such a way that the examples in each category are divided by a clear space that is as broad as possible. Each new instance is mapped in the same space and predicted as belonging to a category based on which side of space they are placed.

Sequential Minimal Optimization (SMO): An implementation of John Platt's minimal sequential optimization algorithm to train a support vector classifier (65). In other words, SMO is a technique for optimizing the SVM training turning it faster and less complex than the previous methods. For that, SMO breaks the problem to be solved into a series of smallest possible sub-problems, which are solved analytically.

OneRule (OneR): Classification technique that generates a rule for each predictor in the data, then selects the rule with the lowest total error as its "single rule" (66). In order to create this rule, this technical analysis the training set associating a single data to a specific category based on its frequency. For instance, if a specific data is usually classified as *category A*, then a rule is created linking them. After the rules creation, the technique choose the one with the lowest total error.

Random Forest: A classifier responsible for building numerous classification trees representing a forest with random decision trees (67). The RF technique adds extra randomness to the model when during the tree's creation. Instead of looking for the best feature when partitioning nodes, it looks for the best feature in a random subset of features. This process creates a great diversity, which generally leads to the generation of better models, besides that this diversity also reduces the overfitting effect.

JRip: An implementation of an apprentice of propositional rules (68). It

is based in association rules with reduced error pruning, a very common and effective technique found in decision tree algorithms. Different from the other algorithms, JRip splits its training stage into two steps, a growing phase, and a pruning phase. The first phase grows a rule by greedily adding antecedents (or conditions) to the rule until the rule is perfect, *i.e.*, 100% effectiveness; The second phase incrementally prune each rule and allow the pruning of any final sequences of the antecedents.

J48: A Java implementation of the C4.5 decision tree technique (69). J48 builds decision trees from a training data set, at each node of the tree, this technique chooses the data attribute that most effectively partitions its set of samples into subsets tending to one category or another. The partitioning criterion is the information gain. The attribute with the highest gain of information is chosen to make the decision. This process is repeated on the smaller partitions.

We chose these techniques because of their comprehensiveness, they involve different data analysis approach, *i.e.*, decisions trees, regression analysis and based-rule analysis that are responsible to create the classifier models. This divergence of the approach allows us to compare the effectiveness and efficiency of them on detecting each studied smell, this comparison lead us to understand the scenarios that each approach can be better applied. Another reason is regarding that they also are widely evaluated in previous studies related to code smell detection (*e.g.*, (27, 47)). We used the Weka package (70) of the R plataform[6] in order to implement these techniques.

2.2.3 Data Collection

To support our study, we extracted 15 potentially-smelly code snippets from the projects analyzed for each type of code smell studied. A potentially-smelly code snippet is a set of statements where its behavior indicates a possible existence of a code smell. We used approached and heuristics provide by previous studies to identify a potentially-smelly code snippet (11, 23, 24, 55). Once the extraction was finished, the 63 developers involved in our study validated the extracted snippets by classifying them either as a *smelly* or *smell-free*. Finally, to turn this classification feasible and to avoid developers' fatigue during classification, we grouped the 63 developers into 10 groups. Each group, composed of 12 developers, was responsible to classify the same 15 code snippets concerning the existence (or not) of only a specific smell type. This repetitive classification for the same code snippet lets us ensure that the same

[6]https://www.r-project.org

code snippet was classified by different developers with different perspectives. As a result of this process, developers classified a total of 1,800 code snippets.

The classification process concerns on the analysis of a potentially-smelly code snippet by looking for the specific smell type reported. The code snippets comprehend the scope of the smell type analyzed, *i.e.*, method, class, package or project. Developers must conclude if a code snippet contains a smell type or not by providing the following answers: **YES**, if the developer agrees that a given code snippet presents the reported smell type; or **NO**, otherwise.

Following the classification of code snippets, we used the Understand[7] to extract software metrics that are used to characterize each code snippet in terms of features to be used during the training process of the ML techniques. Figure 2.1 presents the schema of the dataset containing the features, *i.e.*, metrics (M1...Mn), and classifications (True or False) associated with the code snippets. Aiming to improve the ML techniques' training, we filtered the extracted metrics before the creation of our dataset. We used only metrics already discussed in previous works and related with each smell scope (11, 23, 24, 55). In other words, whether a smell affects an entire class, we preferred to use metrics at the same level. In this way, each code smell has a different set of metrics used for training the ML techniques to its detection. Finally, we created one dataset for each code smell type analyzed in our study.

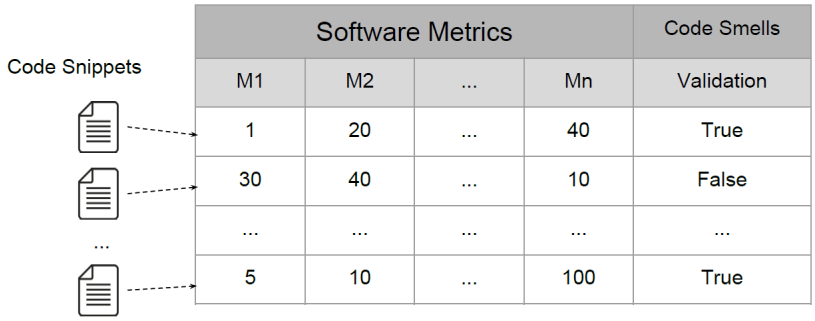


Figure 2.1: Schema of the Dataset.

2.2.4 Effectiveness Metrics

To assess the performance of the ML techniques, we used the *F-measure* that considers both the *recall* and *precision* to compute a score. For our study, the *true positive* **(TP)** elements represent the code snippet classified by the ML techniques as a code smell that are, actually, a real code smell, as well as the *false positive* **(FP)** elements refer to the code snippets wrongly classified

[7]https://scitools.com/features/

as code smell. Similarly, the *true negative* **(TN)** represents the code snippets correctly classified as not-smell and the *false negative* **(FN)** represents the wrong ones. In this context, we can define the *recall* and *precision* as:

- **Recall (R)** : Number of code snippets correctly classified as code smells among the total of code smell instances in the data collection.

$$R = \frac{TP}{TP + FN} \tag{2-1}$$

- **Precision (P)** : Number of code snippets correctly classified as code smell among the total of code snippet classified as code smell by the ML technique.

$$P = \frac{TP}{TP + FP} \tag{2-2}$$

- **F-Measure**: Harmonic mean of precision and recall.

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{2-3}$$

2.2.5 Operation

Using the datasets containing the developers' evaluations and the software metrics for each analyzed code snippet, we performed three different experiments. Each experiment aims at answering a research question.

(Overall Effectiveness) To answer **RQ1**, we used the datasets to analyze the effectiveness (in terms of *f-measure*) of the ML techniques on detecting a specific smell type, without considering the individual perceptions of each developer. For each smell type, we calculated the overall effectiveness of each technique by applying a 10-fold cross validation procedure on the 180 classifications performed by the 12 developers.

(ML techniques Dispersion) To answer **RQ2**, we used the datasets to evaluate the effectiveness variation of each technique on detecting code smells according to the individual perceptions of the developers. For each developer, we evaluated the effectiveness of each technique by applying a five-fold cross validation on the 15 code snippets analyzed by him. After obtaining the effectiveness of the techniques for each developer, we analyzed the effectiveness variation obtained by each technique on detecting smells to the different developers. Finally, we also analyzed the most effective techniques by counting the number of developers in which each technique obtained the highest effectiveness.

(ML techniques Efficiency) Aiming to answer **RQ3**, we evaluate the efficiency of the ML techniques, *i.e.*, the effectiveness of each ML technique whereas we vary the number of code smells examples used to perform the training of these techniques. We evaluated the effectiveness by considering the 15 classifications performed by each developer. However, we ranged the number of examples from three (*i.e.*, 20% of the examples) to 12 (*i.e.*, 80% of the examples), aiming to guarantee that both, the training and test sets, were composed of snippets classified as *smelly* and *smell-free* by the developer.

2.3 Results and Discussion

This section presents and discusses the main results of the study. The results are organized in terms of the three research questions presented in Section 2.2.

2.3.1 Overall Effectiveness

To answer **RQ1**, for each smell type studied, we analyze the effectiveness of the ML technique to detect smells for the developers responsible to evaluate the smell type in analysis. Figures 2.2 to 2.11 present the overall effectiveness of the ML techniques on detecting the 10 smell types analyzed. In each figure, the *x-axis* presents the ML technique used to detect a type of code smell, while the *y-axis* describes the values of the effectiveness (in terms of *f-measure*) obtained by the ML technique on detecting the smell types analyzed. To improve readability, we attach the median value of the *f-measure* to the top of the bars associated with each ML technique.

(**Highest Overall Effectiveness**) The *Random Forest* reached the highest effectiveness in four smell types: *Data Class*, *Middle Man*, *Refused Bequest* and *Speculative Generality*. Note that only in the *Data Class*, the *Random Forest* obtained an effectiveness above 0.8. Regarding the *Naive Bayes*, it reached the highest effectiveness in three smell types (*Feature Envy*, *Long Method* and *Primtive Obsession*), obtaining values above 0.8 only in the *Long Method* and *Primitive Obsession*. Finally, the *J48*, *OneR* and *SMO* obtained the highest effectiveness only one smell type. While the OneR and SMO reached the highest effectiveness in the *Message Chains* and *Inappropriate Intimacy*, respectively, the J48 presented the highest one in the *God Class*. Only the *J48* was able to reach an effectiveness above 0.8. The *SVM* and *JRip* could not obtain the highest effectiveness in none of the smell types.

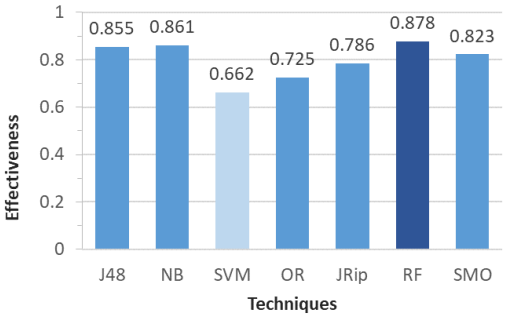


Figure 2.2: Effectiveness on DCL

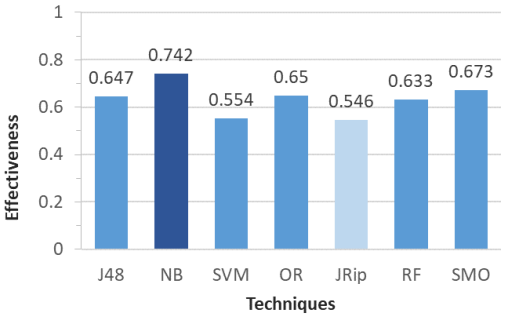


Figure 2.3: Effectiveness on FE

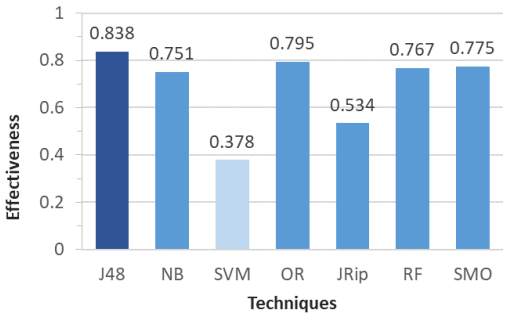


Figure 2.4: Effectiveness on GC

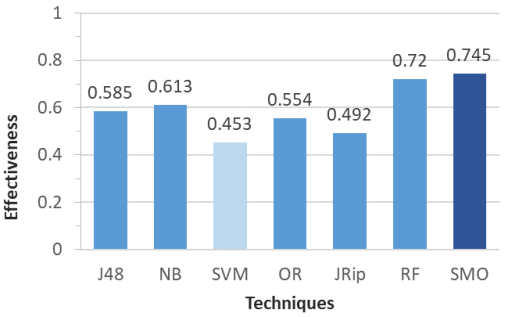


Figure 2.5: Effectiveness on II

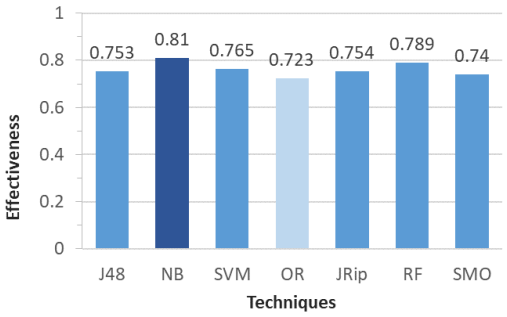


Figure 2.6: Effectiveness on LM

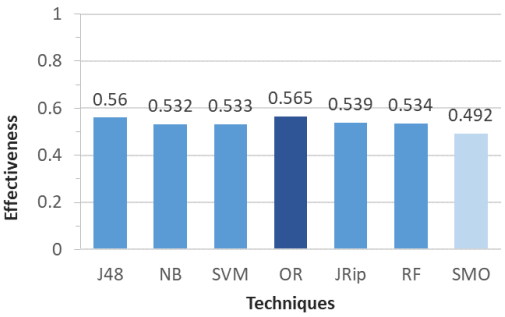


Figure 2.7: Effectiveness on MC

(**Lowest Overall Effectiveness**) The SVM and JRip obtained the lowest overall effectiveness in most of the smell types analyzed. While the SVM presented the lowest effectiveness in 6 smell types (*Data Class*, *God Class*, *Inappropriate Intimacy*, *Primitive Obsession*, *Middle Man* and *Speculative Generality*), the JRip obtained the lowest one in the *Feature Envy* and *Refused Bequest*. The SMO and OneR presented the lowest effectiveness in the *Message Chains* and *Long Method*, respectively.

The results indicate that **the *Random Forest* and *Naive Bayes* were able to reach the highest overall effectiveness on detecting 7 of the 10 smell types analyzed. On the other hand, the *SVM* obtained the lowest effectiveness in 6 of the 10 smell types.** Such results reinforce a previous study (30) that indicates a high effectiveness of the

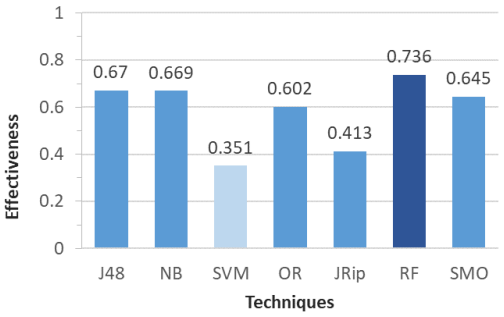


Figure 2.8: Effectiveness on MM

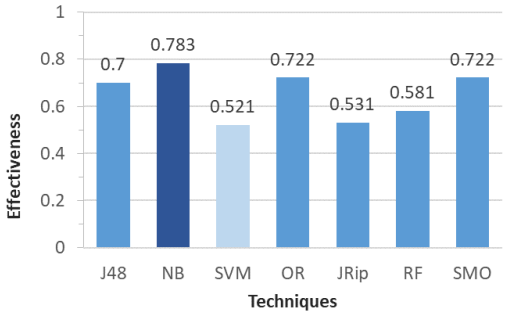


Figure 2.9: Effectiveness on PO

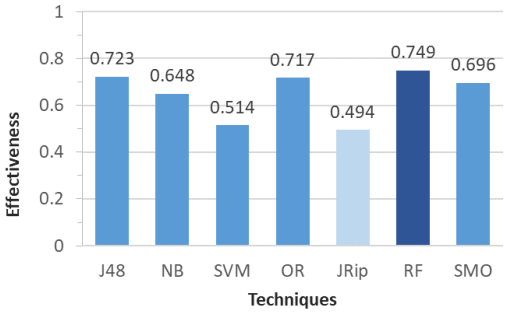


Figure 2.10: Effectiveness on RB

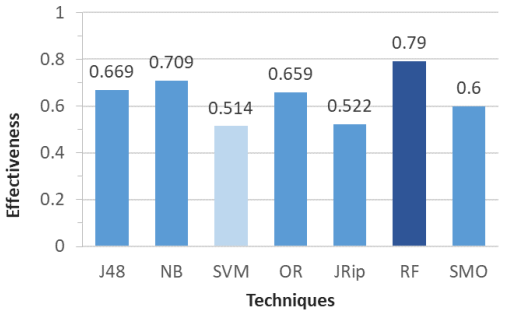


Figure 2.11: Effectiveness on SG

Random Forest and low effectiveness of the *SVM* on detecting four smell types: *Data Class*, *God Class*, *Feature Envy* and *Long Method*. Besides analyzing these smell types, our study also evaluated the *Inappropriate Intimacy*, *Primitive Obsession*, *Message Chains*, *Middle Main*, *Refused Bequest* and *Speculative Generality*. Such analysis enables us to identify four minor findings. First, the *Random Forest* is able to obtain the highest effectiveness on detecting *Middle Man*, *Refused Bequest* and *Speculative Generality*. Second, the *Naive Bayes* is an effective ML technique to detect smells since it obtained the highest effectiveness in 3 of the 10 smell types (*Feature Envy*, *Long Method* and *Primitive Obsession*). Third, we observe a low effectiveness of the *SVM* on detecting smell types, such as: *Inappropriate Intimacy*, *Primitive Obsession*, *Middle Man* and *Speculative Generality*. Finally, note that even though the *Random Forest* and *Naive Bayes* have reached the highest effectiveness in the most smell types analyzed, they were able to obtain values above 0.8 only in few cases.

2.3.2 ML Techniques Dispersion

As discussed in the previous section, the techniques could not reach an overall effectiveness above 0.8 in the vast majority of the cases analyzed. Our intuition is that the effectiveness of the techniques present a dispersion on

detecting code smells for each developer. As a consequence, the techniques obtained a low overall effectiveness. Hence, we investigate the **RQ2** aiming at analyzing the effectiveness variation of each technique on detecting developer-sensitive smells. Figures 2.12 to 2.11 present *beanplot* graphics (71) that support the discussions about this research question. In each figure, the *x-axis* describes the ML technique evaluated. Associated to each technique, we present a *beanplot* graphic representing the density of the effectiveness values obtained by the technique on detecting smells for each developer. In our study, we use the *standard deviation SD* to quantify the variation or dispersion of the effectiveness values obtained by each technique on detecting smells for the developers. We attach the SD to the top of the bars associated with each ML technique.

(**Highly Disperse ML techniques**) We observe that all the ML techniques present a variation in their effectiveness on detecting smells for different developers. By analyzing the SD obtained by each technique, we observe that the SVM technique presented the highest dispersion in seven out of 10 smell types analyzed. In the previous section, we observed that the SVM presented the lowest overall effectiveness. Such results suggest that a high dispersion may lead ML techniques to reach a low overall effectiveness.

(**Lowly Disperse ML techniques**) Both the *Random Forest* and *Naive Bayes* obtained the lowest dispersion in the highest number of cases analyzed. While the *Random Forest* presented the lowest SD in the *Speculative Generality*, *Refused Bequest* and *Message Chain*, the *Naive Bayes* presented the lowest one in the *Feature Envy*, *Long Method* and *Middle Man*. Note that both techniques obtained the highest overall effectiveness, as described in Section 2.3.1. Such results suggest that a low dispersion may lead ML techniques to reach a high overall effectiveness.

Although the ML techniques have presented a variation of their effectiveness on detecting code smells for different developers, these techniques were able to reach high effectiveness for specific developers. Hence, we analyze the most effective techniques to detect smells for each developer. Figures 2.22 to 2.31 present the main results that support the discussions about this analysis. In each figure, the *x-axis* describes the *id* of a developer that evaluated code snippets related to a smell type, and, the ML techniques that reached the highest effectiveness (in terms of *f-measure*) on detecting smells for the developer. The *y-axis* presents the highest effectiveness reached by the ML techniques for a specific developer. We attach the highest *f-measure* value to the top of the bars associated with each developer.

(Effectiveness Above 0.8) As discussed in Section 2.3.1, the ML

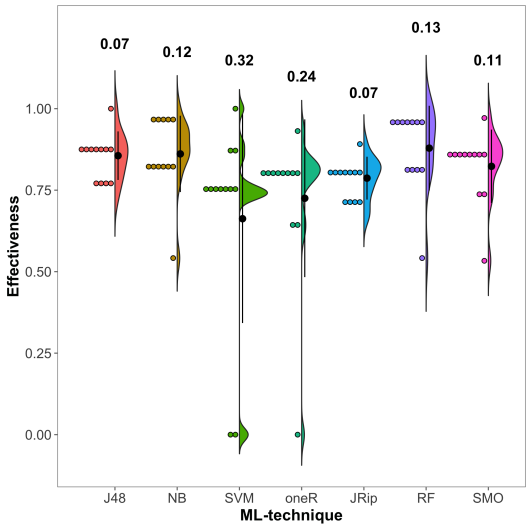


Figure 2.12: DCL Density

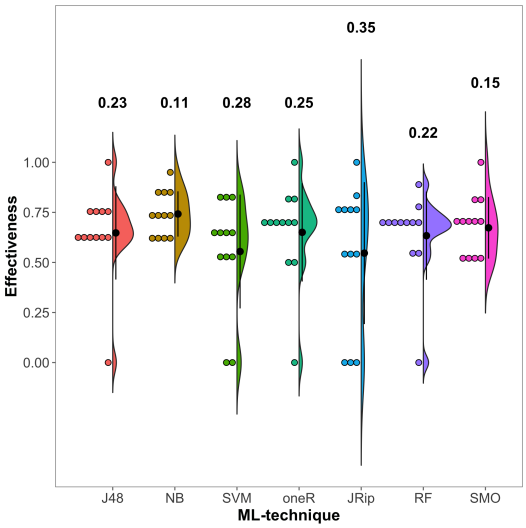


Figure 2.13: FE Density

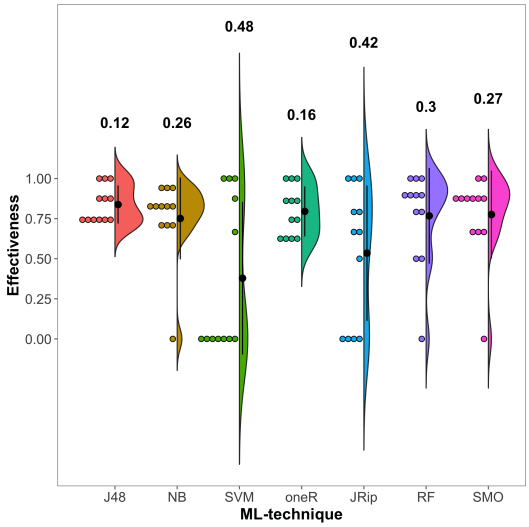


Figure 2.14: GC Density

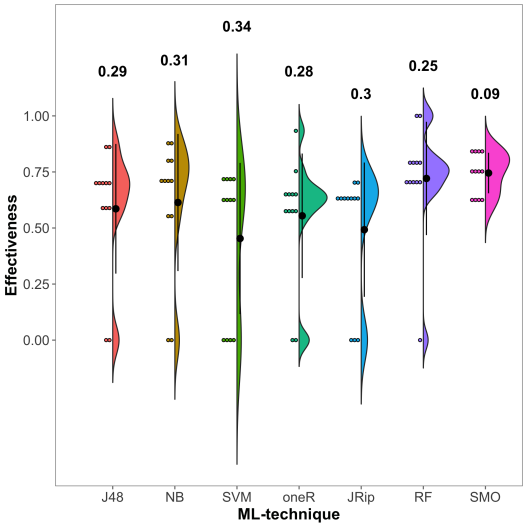


Figure 2.15: II Density

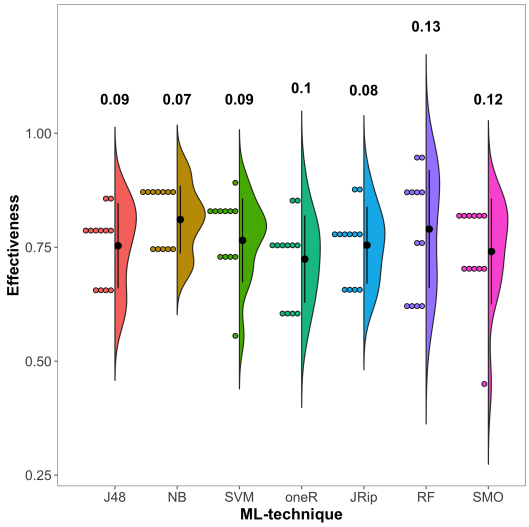


Figure 2.16: LM Density

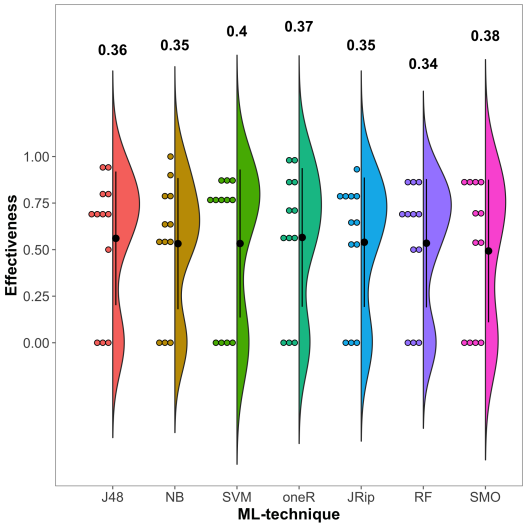


Figure 2.17: MC Density

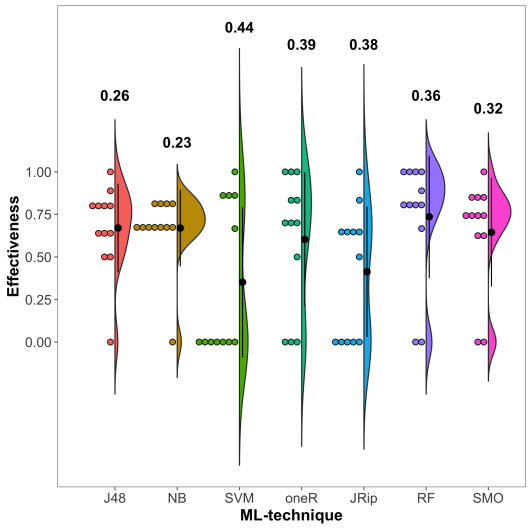


Figure 2.18: MM Density

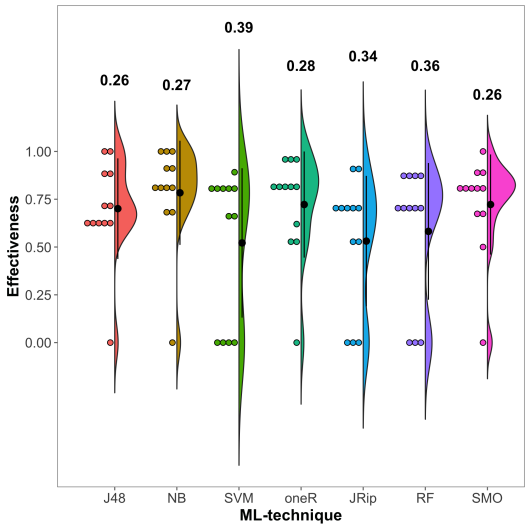


Figure 2.19: PO Density

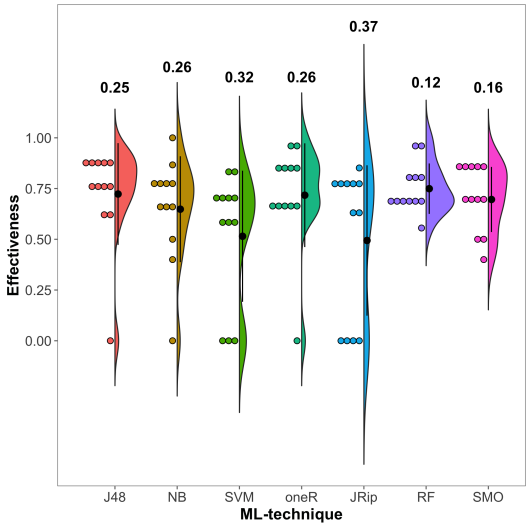


Figure 2.20: RB Density

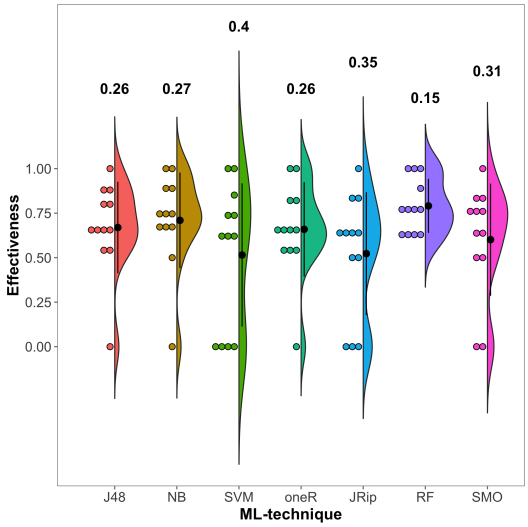


Figure 2.21: SG Density

techniques could not reach an effectiveness above 0.8 in the vast majority of the cases analyzed. Hence, we investigate how many techniques reached an effectiveness above 0.8 when we consider the developer's perception. For each smell type, we observe that the techniques reached an effectiveness above 0.8 for at least 8 of the 12 developers. In the case of the *Data Class* (see Figure 2.22), *God Class* (see Figure 2.24), *Long Method* (see Figure 2.26) and *Primitive Obsession* (see Figure 2.29), the results were even better since the techniques obtained such effectiveness for 11 of the 12 developers. Even in the cases in which the techniques could not obtain an effectiveness above 0.8, they obtained values that varied from 0.67 to 0.79.

(Effectiveness of 1.00) We also observe that the techniques reached an effectiveness equal to 1.00 for 8 of the 12 developers that evaluated *God Class*. For the remaining smell types, the techniques obtained such effectiveness for 2

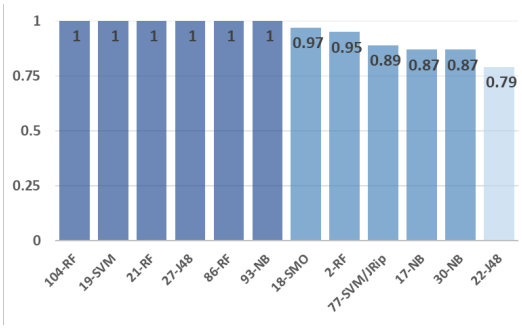


Figure 2.22: Effectiveness on DCL

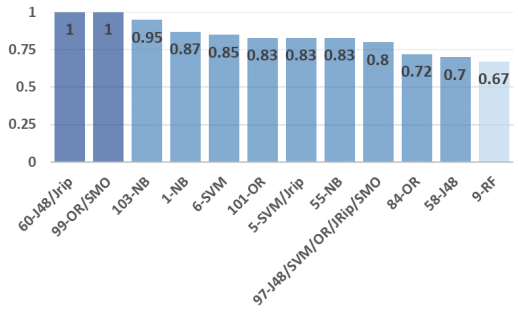


Figure 2.23: Effectiveness on FE

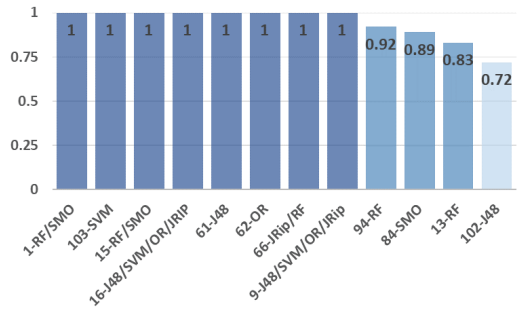


Figure 2.24: Effectiveness on GC

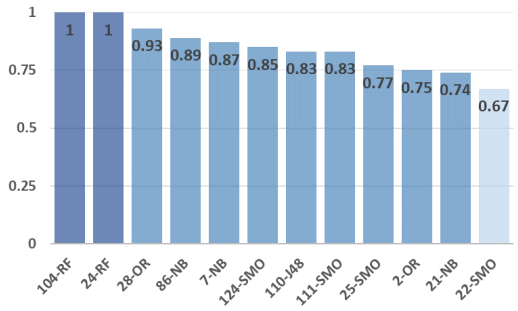


Figure 2.25: Effectiveness on II

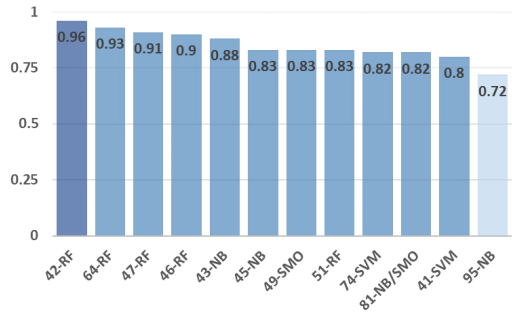


Figure 2.26: Effectiveness on LM

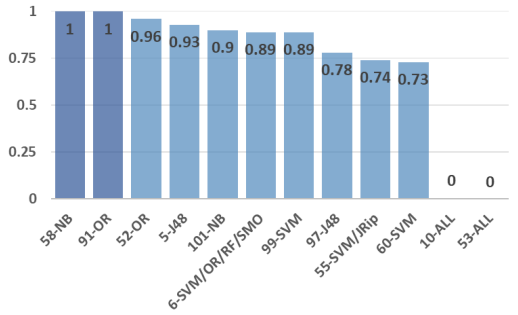


Figure 2.27: Effectiveness on MC

(*Feature Envy*, *Inappropriate Intimacy*, *Refused Bequest* and *Message Chains*) up to 7 (*Middle Man*) developers responsible for evaluating each smell type. The only exception was the *Long Method* in which none of the techniques could obtain an effectiveness of 1.00. Such results indicate that ML techniques are able to reach the highest effectiveness from a reduced number of code smells examples, *i.e.*, ML techniques were able to reach high effectiveness from only the 15 code snippets annotated by each developer as smell or not (see Section 2.2.3). This finding is important since the annotation of a large amount of examples may introduce an unfeasible additional effort to the developers.

(Highly Effective Techniques) After analyzing the effectiveness of the techniques on detecting smells for each developer, we investigated the highly effective techniques by counting the number of cases in which a technique reached highest effectiveness. For each smell type, we observe that different

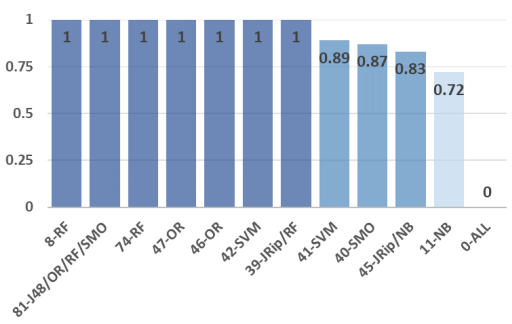


Figure 2.28: Effectiveness on MM

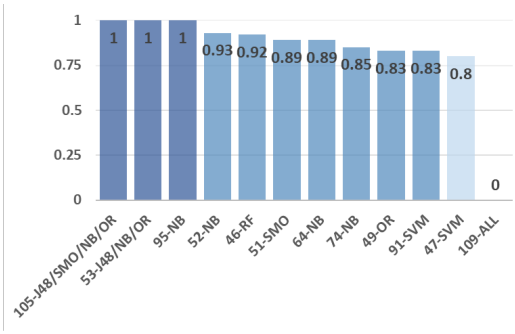


Figure 2.29: Effectiveness on PO

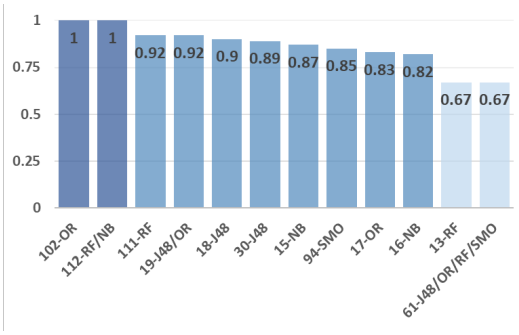


Figure 2.30: Effectiveness on RB

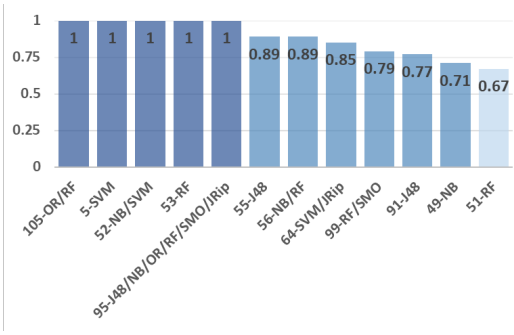


Figure 2.31: Effectiveness on SG

techniques could obtain highest effectiveness on detecting developer-sensitive smells. For instance, each technique could reach the highest effectiveness for at least one developer in the *Data Class*. Note also that a technique may reach the highest effectiveness in different smell types. For instance, the *Random Forest* obtained the highest effectiveness in the greatest number of smell types. Indeed, it reached the highest effectiveness in 6 of the 10 smell types analyzed. The *OneR* reached the highest effectiveness for the *Feature Envy* and *Refused Bequest*. The remaining techniques obtained the highest effectiveness in only one smell type.

(Lowly Effective Techniques) Even though the *OneR* has obtained the highest effectiveness in two cases (*Feature Envy* and *Refused Bequest*), it reached the lowest effectiveness in the *Data Class*, *Long Method* and *Speculative Generality*. The *JRip* presented the worst performance by obtaining the lowest effectiveness in seven of the 10 cases analyzed. The remaining techniques were lowly effective in only one smell type.

A previous study (50) indicates a statistically significant divergence among the developers' perceptions about the existence of a same code smell and, hence, smells detection techniques must consider the individual perception of each developer. **Our results reinforce such finding since we observe that ML techniques can be highly effective on detecting smells for each developer. Note that the techniques reached an effectiveness above 0.8 in the vast majority of the cases analyzed. Indeed, the**

techniques obtained an effectiveness of 1.00 in a high number of the cases analyzed. Finally, we also observe that while the *Random Forest* was the most effective technique, the *JRip* was the less effective one.

2.3.3 Efficiency

According to the results for **RQ2**, ML techniques were able to reach high effectiveness on detecting developer-sensitive smells. However, we do not know the efficiency of these techniques (see **RQ3**), *i.e.*, the percentage of examples required by each technique to reach high effectiveness. Figures 2.32 to 2.41 present the results that support the discussions regarding this research question. These figures represent the efficiency reached by the ML techniques on detecting the smell types analyzed. The *x-axis* describes the percentage of the examples used in the training phase of the techniques, while the *y-axis* represents the *median* of the effectiveness values obtained by each ML technique on detecting smells for different developers.

For each smell type, we observe that different techniques are able to reach the highest effectiveness whereas we increase the percentage of training examples. Let's consider the *Refused Bequest* smell, while the *Random Forest* reached the highest effectiveness from 20% up to 60%, the J48 obtained the highest one from 70%. Even though different techniques can reach the highest effectiveness whereas we increase the number of examples, we observe that some techniques are more effective in a greater number of percentages analyzed. For instance, the *Random Forest* reached the highest effectiveness in the majority of the percentages related to the *Refused Bequest*. Regarding the *Inappropriate Intimacy*, the *Random Forest* obtained results even better by reaching the highest effectiveness in all the percentages analyzed. We also observe that the *Random Forest* obtained the highest effectiveness in almost all percentages related to the *Data Class*, except when we consider 70% of the training examples. In such case, the *Naive Bayes* reached the highest effectiveness.

Similarly to the *Random Forest*, the *SMO* was the most effective in a greater number of percentages analyzed in three smell types: *Primitive Obsession*, *God Class* and *Speculative Generality*. Regarding the *Primitive Obsession*, we observe that the *SMO* reached the highest effectiveness in the vast majority of the percentages analyzed, except in 80% in which the *Naive Bayes* obtained the highest effectiveness. Note also that the *SMO* obtained the highest effectiveness from 50% up to 80% of the percentage analyzed in the

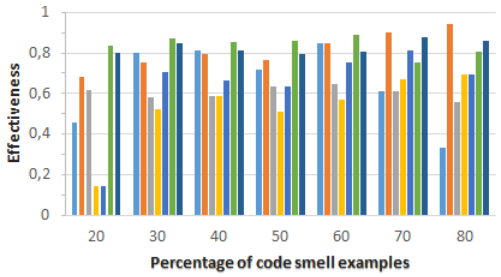


Figure 2.32: Efficiency on DCL

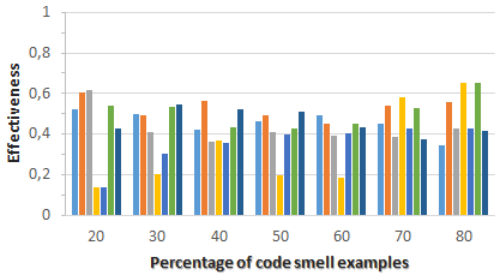


Figure 2.33: Efficiency on FE

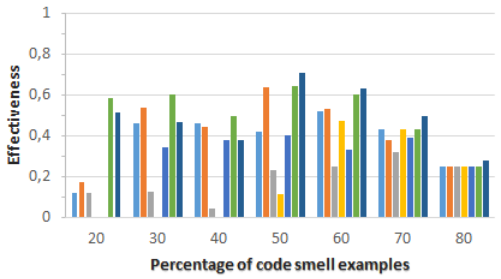


Figure 2.34: Efficiency on GC

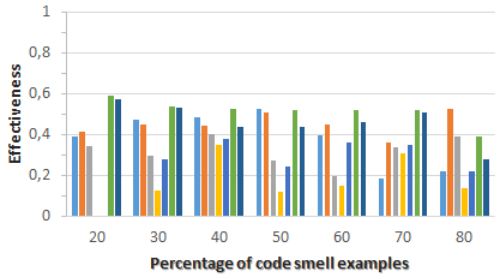


Figure 2.35: Efficiency on II

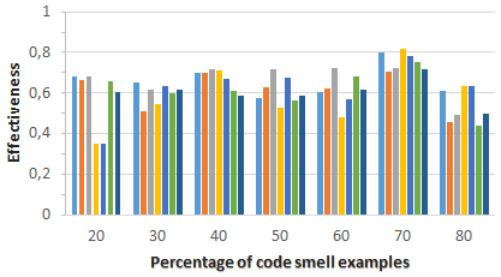


Figure 2.36: Efficiency on LM

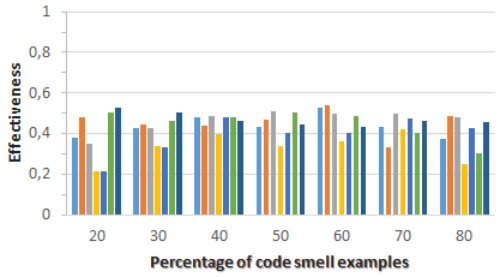
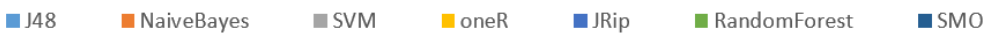


Figure 2.37: Efficiency on MC



God Class. Concerning the *Speculative Generality*, four techniques were able to reach the highest effectiveness: *SMO* (20%; 40%; 50%; 80%), *Random Forest* (30%), *JRip* (60%) and *Naive Bayes* (70%). But, we observe that the *SMO* reached the highest effectiveness in 4 of the 7 percentages analyzed.

Both the *SVM* and *NB* were most effective in a greater number of percentages analyzed in two smell types. While the *SVM* reached a greater number of highest effectiveness in the *Long Method* and *Message Chain*, the *Naive Bayes* was most effective in the *Middle Man* and *Feature Envy*. Regarding the *Long Method* and *Message Chain*, the *SVM* reached the highest effectiveness in four of the 7 percentages analyzed in both these smell types. The *Naive Bayes* reached the highest effectiveness at least four of the 7 percentages analyzed in both the *Middle Man* and *Feature Envy*.

Such results indicate that *Random Forest*, *SMO*, *SVM* and *NB* were most

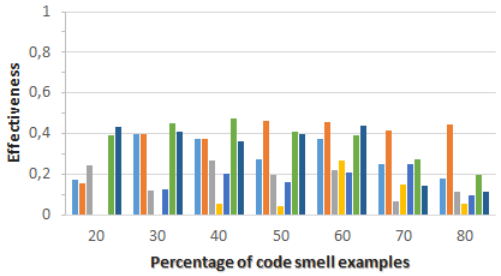


Figure 2.38: Efficiency on MM

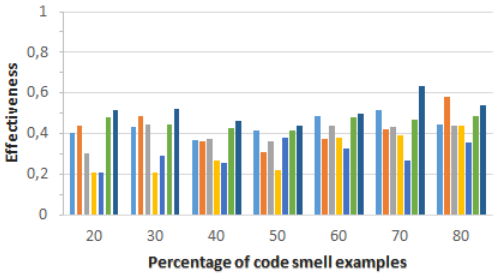


Figure 2.39: Efficiency on PO

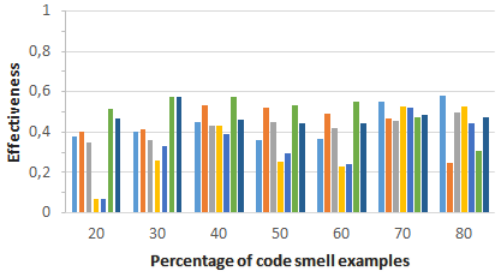


Figure 2.40: Efficiency on RB

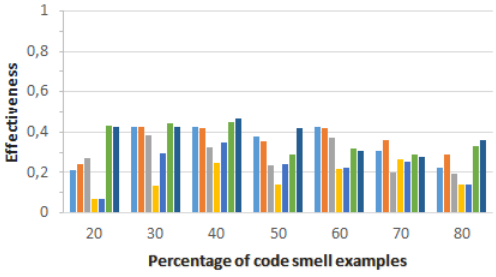


Figure 2.41: Efficiency on SG

effective in the majority of the percentages analyzed. However, we observe that the *Random Forest* presented an effectiveness higher than the other techniques when we consider between 20% and 60% of the examples related to *Refused Bequest*, *Inappropriate Chain* and *Data Class*. Actually, even in the *God Class*, in which the *SMO* was most effective, the *Random Forest* could obtain the highest effectiveness when we consider up to 40% of the training examples. This finding suggest that **the *Random Forest* is able to detect developer-sensitive smells more effectively with lower number of examples than the other analyzed techniques on detecting *Data Class*, *Inappropriate Intimacy*, *Refused Bequest*, *God Class* and *Speculative Generality*.**

2.4 Threats to Validity

In this section we discuss the threats to validity in accordance with the criteria defined in (72).

Construct Validity. The datasets that supported our study were built from code snippets manually evaluated by developers. In this case, the developers evaluated each snippet by reporting the option "YES" or "NO", referring the presence or absence of a given code smell into the snippet. Providing only these two options may be a threat, since the developers could

not inform the degree of confidence in their answers. However, we adopted such procedure aiming at ensuring that the developers were able to decide about the existence of a code smell and we could obtain a set of examples that enables to perform our study.

The used code snippets may contain more than one code smell. However, the presence of more than one code smell in a snippet does not unfeasible the assessment, once we ask explicitly to the developers about the existence of a specific smell. Besides, the existence of several smells types does not change the fact that the type we want to observe still exists. Finally, the chosen set of metrics used for training the ML techniques are specific for the studied smell, which improves its detection.

The developers used to classify the code smells are not the same as the project's developers. However, it is normal that new developers work on legacy projects. Because of this, the ML techniques need to be able to evaluate in that context. Besides, it is impracticable for the project developers to classify all the code snippets since the projects have existed for years. Finally, often developers work in group in a certain code snippet, so we would not know which developer did each code snippet to have that correct evaluation.

Internal Validity. The use of the Weka package of the R platform to implement the techniques analyzed in our study enabled to experiment a variety of configurations, which affect the training process of the techniques. In such context, the configurations considered in our experiments may impact in the effectiveness and efficiency of the techniques. In order to mitigate this threat, we configured all ML techniques according to the better settings defined in (30). Indeed, (30) performed a variety of experiments in order to find the best adjust for each technique.

External Validity. The code snippets evaluated by the developers were extracted from five Java projects. Such projects have been widely used in other works related to code smell (28, 29, 48, 54). However, although the implementation of these projects present classes and methods with different characteristics (*i.e.* size and complexity), our results might not hold to other projects. In the same way, even though we have performed our experiments with 63 different developers, our results might not also hold for other developers since they may have different perceptions about the code smells analyzed in our study (12, 13, 59, 60).

2.5 Related Work

Several machine learning techniques have been adapted to enable an automatic detection of code smells (*e.g.*, (27, 47)). Although these studies report interesting results concerning the effectiveness and efficiency of ML techniques to detect code smells, there is still little knowledge about the sensitivity of ML techniques to recognize developer-sensitive smells.

In (28), the authors proposed the *Bayesian Belief Network* (BBN) to detect instances of *God Class*. They used four graduate students to validate a set of classes, reporting if each class contains a *God Class* instance or not. From such procedure, they built a dataset containing 15 consensual smell instances and then they applied a 3-fold cross-validation on this dataset in order to evaluate the performance of the BBN. They obtained an accuracy of 0.68 on detecting *God Class*. In (48), the authors extended the study (28) by applying the BBN to detect instances of*Blob*,*Spaghetti Code* and *Functional Decomposition*. They involved seven students to create datasets and then they evaluated the effectiveness of BBN to detect these smell types.

The study described in (29) assessed the effectiveness of *Support Vector Machine* in the detection of four types of code smell: Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife. The SVM obtained an accuracy up to 0.74. In (49) the authors proposed the use of *Decision Tree* technique to detect code smells. The authors used a single dataset containing a huge number of examples validated by few developers. The results indicate that the *Decision Tree* is able to reach an accuracy up to 0.78.

Fontana *et al.* (30) presented a large study that compares and experiments different configurations of machine learning techniques to detect four types of smell. To perform the training of these techniques, the authors used a dataset containing several examples of code smells manually validated by few developers. The *J48* and *Random Forest* obtained the highest accuracy, reaching a values up to 0.95. However, a recent study (73) indicate that the dataset used by Fontana et al. (30) had a high influence in the accuracy obtained by the techniques.

Several studies (27, 47) analyzed the accuracy and efficiency of ML techniques in the detection of only 4 distinct smell types. The results of both studies indicated that the *Random Forest* is able to reach a high effectiveness and efficiency. Such results reinforce the findings identified in our study, which suggest that the *Random Forest* is an promising way to identify developer-sensitive smells.

2.6 Conclusion

This paper presented a study that analyzed the sensitivity of ML techniques on the detection of developer-sensitive smells. Firstly, we evaluated the overall effectiveness of the ML techniques to recognize smells. Then, we investigated the effectiveness variation of ML techniques on detecting smells for different developers. Finally, we analyzed the efficiency of the ML techniques by evaluating their effectiveness according to the number of examples used to perform the training process.

The results indicated that while the *Random Forest* and *Naive Bayes* reached the highest overall effectiveness on detecting smells, the SVM obtained the lowest one. We also observed that all the analyzed techniques are sensitive to the developers' perceptions and the SVM is the most sensitive one. The *Random Forest* is the most effective to detect developer-sensitive smells.

As future work, we intend to investigate the sensitivity of ML techniques on detecting other smell types. In addition, we also intend to replicate this study in controlled scenarios, considering developers and projects of a same organization. In this way, we expect to identify if developers, who work together, have the same influence on the detection of code smells.

2.7 Summary of Chapter 2

In this chapter, we performed a broader study aimed at investigating the ability of ML techniques to customize the detection of developer-sensitive smells, *i.e.*, when smell detection has to take into account the individual knowledge of each developer. We investigated the variation of the ML techniques' effectiveness to identify developer-sensitive smells. We also analyzed the variation of each technique's effectiveness as we gradually increased the number of examples used for training.

We evaluated the sensitivity of ML techniques based on their effectiveness of identifying 10 smell types according to the individual knowledge of 63 developers. The results showed that all the analyzed techniques are sensitive to the developers' knowledge, for all analyzed smell types and with a very low number of required training examples. Also, our results suggest that ML techniques can be highly effective in customizing their detection strategy for each developer. In fact, the techniques reached high effectiveness in the vast majority of the cases analyzed.

3
Customization of Code Smell Detection: A Second Study

The employed dataset used for the training stage of the ML techniques is likely to have a direct influence on the obtained effectiveness. This influence may compromise the assessment of those techniques as well as the validity of our findings in Chapter 2. In this context, we carried out a complementary second study aimed at investigating the ability of ML techniques on detecting developer-sensitive smells. In this study, we built a new dataset the behavior of the ML techniques for such new training data. Different from the first study, we focused here on assessing ML techniques only for detecting smells that actually ended up being refactored out by a developer.

The refactoring of smelly code indicates THAT the developer, either consciously or not, confirmed the relevance of a smell. Those smells can be considered relevant to the program as their removal helped the developer to achieve his maintenance goal, which may vary from pure structural improvement to bug fixing or feature addition. Moreover this second study involved developers who were working on their own projects. The strict assessment of the refactored smells ensures they were somehow harmful to a software developer's maintenance task. The previous study was focused instead on developers who are not the owners of the source code (*e.g.*, they represent situations of newcomers in the project), but they have the role of understanding and reviewing the source code.

Our second study is reported in the paper *"Assessing Machine Learning Techniques on Code Smell Detection"* (43), which is presented in this chapter and being submitted to Brazilian Symposium on Software Engineering (SBES) in May 2020. In the case the reader has already read Section 2.5, you may consider skipping the Section 3.2.2 in this chapter as it repeats similar content from that previous section. The same strategy can be considered for Section 2.4 and Section 3.5.1.

Assessing Machine Learning Techniques on Code Smell Detection

Daniel Oliveira[1]
doliveira@inf.puc-rio.br
Alessandro Garcia[1]
afgarcia@inf.puc-rio.br

[1]Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil

3.1 Introduction

Code smells are considered symptoms of poor design and implementation choices, which make the software system hard to evolve and maintain (46). Due to their harmfulness to software quality (3, 4, 5), code smell should be detected as early as possible to enable its removal. Unfortunately, several reasons make the code smell detection a challenging task.

Some studies (74, 75, 76) suggest different approaches to detect code smells. The vast majority of these approaches are based on detection strategies, which are rules composed by metrics and their respective thresholds. These approaches tend to analyze each code fragment and employ some previously defined rules to classify the fragment as the host of a specific smell (or not). Each smell type has its own rules, what turns the detection exhaustive and not generalized. This difficulty stems from the fact that the operationalization (*i.e.*, the rule definition) of the strategy for detecting each smell type requires proper reasoning. Such an operationalization can not be solely based on finding metrics and thresholds in accordance with the conceptual definition of a smelly type. The operationalization also needs to be customized by considering various contextual information of the program (and the organization), which only the developer has access to.

Given these challenges on smell detection, several studies (27, 30, 47, 49, 57, 77) have analyzed the use of machine learning techniques (ML techniques) to identify smells. In a nutshell, the ML techniques require a training set containing code examples annotated as smell or non-smell. From these training examples, the ML techniques generate smell detection models that are customized for each smell type, this customization has improved the techniques

effectiveness, as seen in (27, 77). Even though such studies indicate that ML techniques are a promising way to detect smells, there is a strong link between the employed dataset and the effectiveness obtained by the techniques. This link is explored by a recent study (73), which observed a possible threat related with the direct influence of the employed dataset on the obtained results.

In this context, this paper reports a study aiming at investigating the effectiveness of seven ML techniques on detecting smells for a new dataset that includes ten different projects. This new dataset is composed by active projects with different sizes and belonging to distinct domains. The dataset also enable us to understand how the ML techniques behave within a wider scope. We assess these techniques based on their ability of detecting six different smell types. We chose smell types that cover different system scopes, *i.e.*, classes, methods, fields and parameters. We performed our study through two main steps:

- **Overall Effectiveness**: We evaluated the overall effectiveness of the ML techniques on detecting each one of the six smell types.
- **ML techniques Efficiency**: We assessed the efficiency of the ML techniques by evaluating the effectiveness of each technique on detecting smells whereas we gradually increase the number of examples used to perform its training.

Finally, our study led to the following findings:

- Previous studies (27, 30, 77) found that *Random Forest* (RF) (58) has been reaching the highest overall effectiveness on detecting smells. In this study, *JRip* was able to detect smells with higher effectiveness. However, Random Forest also reached outstanding results, specially for four (out of six) smell types under analysis. On the other hand, differently from (30) the *Naive Bayes* (NB) (58) yielded the lowest overall effectiveness in two smell types (Section 3.4.1);
- The effectiveness of all the analyzed ML techniques were influenced by the smell type. As a consequence, each of the techniques obtained diverging performance results for each smell type. (Section 3.4.1);
- The ML techniques achieve similar effectiveness results for the same smell type. This finding leads us to believe that does not appear to have a better approach. In this way, the choice of the most convenient technique is being at the discretion of the developer.
- The ML techniques do not need a high number of examples to reach high results. In fact, the increase in the number of instances in the training

set did not appear to have a direct relationship with the increase in effectiveness for all ML techniques. (Section 3.4.2).

These findings indicate that, although RF and JRip obtained the best results between the analyzed techniques, all the studied ML techniques have similar behavior when working with different smells types. Besides that, they also have good support for detecting code smells in projects with different sizes, once they do not need a high number of examples to reach high results.

The remaining of this document is structured as follows. Section 3.2 describes the background of the study and the related work, followed by 3.3 that describes the design of our study including the research questions. Section 3.4 presents the results of the study and the answers to our research questions. Section 3.5 details the threats and limitations of the study. Finally, Section 3.6 presents the conclusions observed in our study and discusses future work.

3.2 Background and Related Work

3.2.1 Background

Code smell detection techniques have been widely investigated. Previous studies (74, 75, 76) suggest different approaches to detect code smells. These approaches are responsible for generating some rules which when they are fulfilled, classifies the code fragment as a certain code smell or not. However, each smell has its own rules what turn this detection process exhaustive and not generalized, once it is necessary a good understanding of each smell definition to composes its unique rules.

To avoid these limitations, several studies (*e.g.*, (27, 47)) have analyzed the use of ML techniques to identify code smells. ML Techniques generate a classifier model for each analyzed smell type based on the knowledge during the training stage. This model is responsible for classifying code fragments as a smell or not.

Our study aims at analyzing instances of six different code smell types. Table 3.1 describes all the selected smell types. We have chosen these smell types due to the different scopes of a program affected by them, *i.e.*, classes, methods or parameters.

These instances were detected from the source code of ten open source

Table 3.1: Types of Code Smells Investigated in this Study

Name	Description
Complex Class (CC)	Classes that involve a lot of different but related parts.
Class Data Should be Private (CDSBP)	Classes that expos its attributes unnecessarily.
God Class (GC)	Classes that tend to centralize the intelligence of the system.
Lazy Class (LC)	Classes that do not do enough.
Spaghetti Code (SC)	Code that has a complex and tangled structure.
Speculative Generality (SG)	Unused classes, methods, fields or parameters created to future features that never get implemented.

Java projects: Apache Ant[1], Apache Derby[2], Apache Tomcat[3], Elastic Search[4], Argouml[5], Apache Xerces[6], Google j2objc[7], Presto db[8], SpringFramework[9] and Achilles[10]. We selected such projects because they have been evaluated by existing smell detection techniques (78) and their source code contains a variety of suspicious code smells (78) that enable the execution of our study.

The seven chosen ML techniques to be evaluated are described below:

NaiveBayes: A probabilistic classifier based on the application of Bayes' theorem (63). This technique is highly scalable and completely disregards the correlation between the variables in the training set. This classifier describes the probability of an event, based on prior knowledge of conditions that might be related to the event.

Support Vector Machine (SVM): An implementation of integrated software for the classification of support vectors (64) that analyzes the data used for classification and regression analysis. SVM assigns new examples to one of the two categories introduced in the training set, making it a non-probabilistic binary linear classifier. In order to make this classification, SVM creates classification models that are a representation of examples as points in space. These points are mapped in such a way that the examples in each category are divided by a clear space that is as broad as possible. Each new instance is mapped in the same space and predicted as belonging to a category based on which side of space they are placed.

Sequential Minimal Optimization (SMO): An implementation of John Platt's minimal sequential optimization algorithm to train a support

[1]https://ant.apache.org/
[2]https://db.apache.org/derby/
[3]http://tomcat.apache.org/
[4]https://www.elastic.co/
[5]https://argouml.tigris.org/
[6]http://xerces.apache.org/
[7]https://github.com/google/j2objc
[8]https://prestodb.io/
[9]https://spring.io/
[10]http://www.ganttproject.biz

vector classifier (65). In other words, SMO is a technique for optimizing the SVM training turning it faster and less complex than the previous methods. For that, SMO breaks the problem to be solved into a series of smallest possible sub-problems, which are solved analytically.

OneRule (OneR): A classification technique that generates a rule for each predictor in the data. Then, it selects the rule with the lowest total error as its "single rule" (66). In order to create this rule, this technical analysis of the training set associating a single data to a specific category based on its frequency, in other words, if a specific *data* is usually classified as *category A*, then a rule is created linking them. After the rules creation, the technique choose the one with the lowest total error.

Random Forest (RF): A classifier responsible for building numerous classification trees representing a forest with random decision trees (67). The RF technique adds extra randomness to the model when during the tree's creation. Instead of looking for the best feature when partitioning nodes, it looks for the best feature in a random subset of features. This process creates a great diversity, which generally leads to the generation of better models, besides that this diversity also reduces the overfitting effect.

JRip: An implementation of an apprentice of propositional rules (68). It is based in association rules with reduced error pruning, a very common and effective technique found in decision tree algorithms. Different from the other algorithms, JRip splits its training stage into two steps, a growing phase, and a pruning phase. The first phase grows a rule by greedily adding antecedents (or conditions) to the rule until the rule is perfect, (*i.e.*, 100% of effectiveness). The second phase incrementally prune each rule and allow the pruning of any final sequences of the antecedents.

J48: A Java implementation of the C4.5 decision tree technique (69). J48 builds decision trees from a training data set. At each node of the tree, this technique chooses the data attribute that most effectively partitions its set of samples into subsets tending to one category or another. The partitioning criterion is the information gain. The attribute with the highest gain of information is chosen to make the decision. This process is repeated on the smaller partitions.

We chose these techniques because of their comprehensiveness. They involve different data analysis approaches, *i.e.*, decision trees, regression analysis and based-rule analysis that are responsible to create the classifier models. This divergence of the approach allows us to compare the effectiveness and efficiency of them on detecting each studied smell type. This comparison leads us to understand the scenarios that each approach can be better applied. An-

other reason is that they also are widely evaluated in previous studies related to code smell detection (*e.g.*, (27, 47)). We used the Weka package (70) of the R plataform[11] in order to implement these techniques.

3.2.2 Related Work

Several machine learning techniques have been adapted to enable automatic detection of code smells (*e.g.*, (27, 47)). Although these studies report interesting results concerning the effectiveness and efficiency of ML techniques to detect code smells, there is still little knowledge about the relation between the chosen dataset and the obtained result.

In (28), the authors proposed the *Bayesian Belief Network* (BBN) to detect instances of *God Class*. They used four graduate students to validate a set of classes, reporting if each class contains a *God Class* instance or not. From such procedure, they built a dataset containing 15 consensual smell instances. Finally, they applied a 3-fold cross-validation on this dataset in order to evaluate the performance of the BBN. They obtained an accuracy of 0.68 on detecting *God Class*. In (48), the authors extended the study (28) by applying the BBN to detect instances of*Blob*,*Spaghetti Code* and *Functional Decomposition*. In this new study, they involved seven students to detect the instances. Then, they evaluated the effectiveness of BBN to detect these smell types.

The study described in (29) assessed the effectiveness of *Support Vector Machine* in the detection of four types of code smell: Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife. The SVM obtained an accuracy of up to 0.74. In (49), the authors proposed the use of *Decision Tree* technique to detect code smells. The authors used a single dataset containing a huge number of examples validated by a few developers. The results indicate that the *Decision Tree* is able to reach an accuracy up to 0.78.

Fontana *et.al.* (30) presented a large study that compares and experiments different configurations of machine learning techniques to detect four code smell types. To perform the training of these techniques, the authors used a dataset containing several examples of code smells manually validated by a few developers. The *J48* and *Random Forest* obtained the highest accuracy, reaching values up to 0.95. However, a recent study (73) indicates that the dataset used by Fontana (30) had a high influence on the accuracy obtained by the techniques.

[11] https://www.r-project.org

The studies described in (27, 47) analyzed the accuracy and efficiency of ML techniques in the detection of 4 distinct code smell types. The results of both studies indicated that the *Random Forest* is able to reach high effectiveness and efficiency when detecting these smell types.

3.3 Study Design

Previous studies (27, 47) suggest that ML techniques are a promising way to identify code smells and have been reaching higher effectiveness. However, as discussed by a recent study (73), the employed dataset used for the training stage of the ML techniques has a direct influence on the obtained effectiveness, compromising the assess external validity. In this context, we built a new dataset to observe the techniques behavior for a new training set, more details about this dataset can be seen in 3.3.1. To evaluate this behavior, our study aims at investigating the capability of ML techniques to detect code smells, *i.e.*, to investigate the effectiveness of each technique in terms of two factors: (i) the type of smell analyzed; and (ii) the number of examples used to perform the training of the ML techniques.

Initially, we defined the research question **RQ1** aiming at investigating the overall effectiveness of ML techniques in detecting six smell types using code fragments from ten different projects. In our study, for each one of the six smell types analyzed, we performed the training of the ML techniques on a dataset containing 200 (non-)smell examples.

> **RQ1.** *How effective are the ML techniques on detecting smells?*

Finally, we investigated the **RQ2** aiming at analyzing the efficiency of the ML techniques on detecting smells, *i.e.*, how effective a ML technique detects smells whereas we gradually increase the number of examples used to perform its training. Although ML techniques have been considered a promising way to detect code smells, these techniques require code smell examples annotated to perform their training. However, the annotation of a large number of examples may introduce an unfeasible additional time and effort. Hence, it is important to analyze the effectiveness variation of the ML techniques whereas we vary the number of examples used to perform the training of these techniques.

> **RQ2**: *How efficient are the ML techniques on detecting smells?*

3.3.1 Data Collection

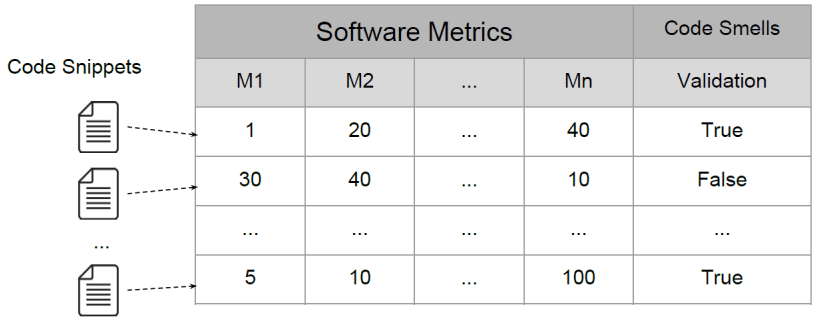
To answer RQ1, we extracted 200 potentially-smelly code fragments from the analyzed projects for each type of code smell studied. This high amount of code fragments allows us to observe the behavior of the techniques in a diversity of code smell instances, besides avoiding overfitting. The detection process was made using a detection tool (78). The tools are performed using rule-based strategies, where each strategy is defined based on a set of metrics and thresholds.

For this study, we selected only code fragments that were directly refactored by the respective software's developer. To collect the data about the refactored elements, we used the RefDetector[12]. This tool is a library/API written in Java that can detect refactorings analyzing the project history. From this information, it was possible to filter among the analyzed smells those that directly underwent a refactoring, in this way, we can ensure that these detected instances are a harmful smell, once the developers insist on refactoring this element affected by the smell. Another advantage of this filter is that it avoids being biased by a single set of detection rules.

Therefore, the application of rule-based strategies requires the collection of metrics for all source files in a project. For that, we used the Understand[13] to extract software metrics. Altogether, 43 metrics were analyzed. These metrics were used during the training process of ML techniques. Also, these features cover different information about classes, methods, and fields, indicating, *e.g.*, the number of lines of the code fragments, relations of complexity within and between elements and several other counters. Figure 3.1 presents the schema of the dataset containing the metrics (M1...Mn), and classifications (True or False) associated to the code fragment indicating the existence or not of the smell in that code. We created one dataset for each code smell type analyzed in our study.

To answer RQ2, we need a small training set and increase it gradually as we analyze the change in effectiveness for each ML technique. In order to obtain this increase, we break each dataset into six subdatasets of different sizes: 20, 40, 80, 120, 160 and 200 instances respectively. This division was made such that each higher subdataset has all the instances of the preceding dataset and some additional. This division property ensures that each code fragment used during the training process using a small subdataset, also will be used when analyzing a bigger subdataset.

[12]https://github.com/xai/RefDetector
[13]https://scitools.com/features/



Software Metrics				Code Smells
M1	M2	...	Mn	Validation
1	20	...	40	True
30	40	...	10	False
...	...	...	...	...
5	10	...	100	True

Figure 3.1: Schema of the Dataset.

3.3.2 Effectiveness Metrics

To assess the effectiveness of the ML techniques, we used the *F-measure* that considers both the *recall* and *precision* to compute a score. For our study, the *true positive* **(TP)** elements represent the code fragment classified by the ML techniques as a code smell that are, actually, a real code smell. The *false positive* **(FP)** elements refer to the code fragments wrongly classified as code smell. Similarly, the *true negative* **(TN)** represents the code fragments correctly classified as not-smell. Finally, the *false negative* **(FN)** represents the wrong ones. In this context, we can define the *recall* and *precision* as:

- **Recall (R)** : The Number of code fragments correctly classified as code smells among the total of code smell instances in the data collection.

$$R = \frac{TP}{TP + FN} \tag{3-1}$$

- **Precision (P)** : The Number of code fragments correctly classified as code smell among the total of code fragments classified as code smell by the ML technique.

$$P = \frac{TP}{TP + FP} \tag{3-2}$$

- **F-Measure**: Harmonic mean of precision and recall.

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{3-3}$$

This mean is widely used in previous studies (27, 77, 30) that assess the ML techniques on detecting code smells.

3.3.3 Operation

Using the datasets containing the classified (non-)smelly instances and the software metrics for each analyzed code fragment, we performed two different experiments. Each experiment aims at answering a research question.

(Overall Effectiveness) To answer **RQ1**, we used the datasets to analyze the effectiveness (in terms of *f-measure*) of the ML techniques on detecting a specific smell type. For each smell type, we calculated the overall effectiveness of each technique by applying a 5-fold cross validation procedure on the 200 classified instances.

(ML techniques Efficiency) To answer **RQ2**, we evaluated the efficiency of the ML techniques, *i.e.*, the effectiveness of each ML technique whereas we increment the number of code smells examples used to perform the training of these techniques. In other words, we repeat the effectiveness experiment six times, one for each subdataset of the respective smell. The repetition aimed to guarantee that both, the training and test sets, were composed of equals number of fragments classified as smell or not.

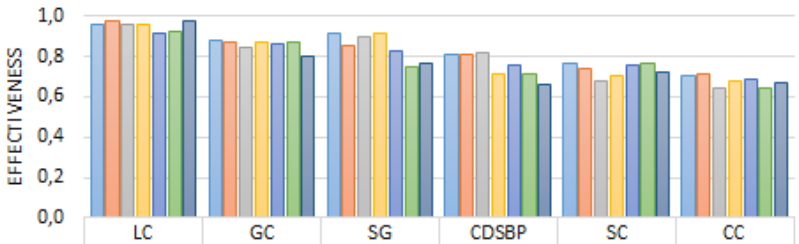
3.4 Results and Discussion

This section presents and discusses the main results of the study. The results are organized in terms of the two research questions presented in Section 3.3.

3.4.1 Overall Effectiveness

To answer **RQ1**, for each smell type studied, we analyze the effectiveness of the ML technique to detect the smells of the respective type. Figure 3.2 presents the overall effectiveness of the ML techniques on detecting each smell type. The *x-axis* is divided per smell and presents sequentially the ML technique used to detect the respective code smell. Meanwhile, the *y-axis* describes the values of the effectiveness (in terms of *f-measure*) obtained by the ML technique on detecting the respective smell type. To improve readability, we attach the median value of the *f-measure* in the table below the bars associated with each smell and highlighted the higher results.

Complex Class. Regarding *Complex Class* smell, RF reached the highest effectiveness of 0,715, while J48 and SMO obtained the lowest values with a slight difference between them. Note that none of the ML techniques reached effectiveness above 0.8. If we look closer to this smell definition, we



	LC	GC	SG	CDSBP	SC	CC
JRip	0,955	**0,877**	**0,913**	0,807	**0,765**	0,702
RF	0,971	0,869	0,849	0,806	0,742	**0,715**
J48	0,956	0,845	0,894	**0,814**	0,675	0,644
OR	0,955	0,874	0,912	0,710	0,706	0,676
SVM	0,913	0,861	0,826	0,756	0,755	0,690
SMO	0,919	0,867	0,749	0,713	0,764	0,645
NB	**0,975**	0,804	0,763	0,662	0,722	0,666

Figure 3.2: Effectiveness Reached by the ML Techniques on Detecting Smells.

observe that there is not a clear threshold about how complex the class needs to be. In general, developers do not agree with some smell classifications because they can associate complexity with different metrics (27, 47). Similarly, this disagreement can also occur with the ML techniques during the training stage, once we are working with more than 40 distinct metrics, which can explain these low results.

Spaghetti Code. Although a bit better, the ML techniques' effectiveness observed in the detection of *Spaghetti Code* is similar to *Complex class*. None of the techniques was able to reach effectiveness above 0.8. J48 reached the lowest effectiveness (0,675) for this smell type. The highest effectiveness (0,765) was obtained by JRip.

Class Data Should Be Private. Except for J48, JRip, and RF, ML, the remaining techniques did not obtain high effectiveness for *Class Data Should Be Private*. J48, JRip, and RF have reached effectiveness over 0.8. NB obtained only 0,662. SVM, OR, and SMO obtained values between 0,7 and 0,8.

God Class. Different from the previously discussed smells, the result for *God Class* reached higher values. All ML techniques obtained effectiveness higher than 0,8. JRip reached the highest effectiveness (0,877). Similarly to *Class Data Should Be Private*, the lowest one was obtained by NB. Note that the coverage of metrics also could implicate the effectiveness obtained by the techniques as seen in *Complex Class*, however some metrics, as Lines of Code, have higher influence when detecting *God Class* (74, 79). Besides that, *God Classes* are usually associated with high values of the metrics.

Lazy Class. The detection of *Lazy Class* is by far the better results observed in this study. All techniques reached values above 0,9. Another difference between the smells previously observed is regarding the NB technique,

which reached the highest result in contrast with its previous results. It is also possible to observe that RF obtained effectiveness slightly close to NB, this proximity also occurs between SVM and SMO, besides J48 and JRip.

Speculative Generality. The ML techniques were able to reach values higher than 0,9 for *Speculative Generality*. JRip, once again, reached the highest effectiveness (0,913), followed closely by OR that also exceeded 0,9. The lowest obtained effectiveness is in charge of the SMO technique, which reached 0,749. An important fact to note is that *Speculative Generality* should be difficult to detect whether we look at a single instance at a time as the ML techniques do because this smell occurs when a developer implements an element (*i.e.*, methods, classes, and fields) that never is used. In other words, it should be necessary to look at this element along different element's versions to decide of the existence or not of this smell, which contradicts the good result obtained by some of the algorithms.

In general, the ML techniques obtained similar results for the same smell, the highest divergence (0,164) is observed in the detection of SG smell between JRip and SMO. Note that JRip reached the highest effectiveness in three out of six types of smell (GC, SC, and SG) at the same time that NB obtained two lowest results (CDSBP and GC). Despite these lowest results, NB reached the highest result for LC, this behavior can be also observed to J48 that obtained the highest result for CDSBP and the lowest one for CC. Although RF had obtained only one highest result, it also has obtained closer results whether compared with the betters one in others four types of smell (CDSBP, GC, LC, and SC), in particular, RF has had good results in previous studies (27, 30, 77) and this behavior was also maintained in this study.

These findings indicate that JRip and RF had better effectiveness in detecting all of the six analyzed smells. Both reached at least good results for five out of six smells, only for CC they obtained a reasonable result, even being their lowest results, were still the best for detection of this smell.

Another implication observed is related to the variation between the minimum and maximum effectiveness of the same smell. It is possible to note that, although JRip and RF obtained the best results, the other techniques did not obtain very different results, which makes possible the use of them at the discretion of the developer.

We can conclude for **RQ1** that, in average, **the ML techniques were not able to detect all the six analyzed types of smells with high effectiveness. On the other hand, they reached high effectiveness on the detection of God Class, Speculative Generality and specially Large Class. These results suggest the effectiveness of these techniques are**

sensitive to the type of smell analyzed.

3.4.2 Efficiency

According to the results for **RQ1**, ML techniques were able to reach high effectiveness on detecting smells for specifics smells. However, we do not know the efficiency of these techniques (see **RQ2**), *i.e.*, the number of instances required by each technique to reach high effectiveness. Figures 3.3 to 3.9 present the results that support the discussions regarding this research question. These figures represent the efficiency reached by the ML techniques on detecting each smell type. The *x-axis* describes the number of the examples used in the training phase of the techniques divided per smell, while the *y-axis* represents the *median* of the effectiveness values obtained by each ML technique on detecting smells.

Is possible to observe that the ML techniques do not follow a unique behavior when the number of analyzed example grows. Some techniques as RF and SVM had a significant increase in SG detection during the addition of new (non-)smelly instances in the training set. In contrast to J48 where a lower training set resulted in higher effectiveness, this same behavior can be seen in JRip when detecting CC smell and NB when detecting CDSBP and GC.

In general, the ML techniques reached results near to their best results in this study on detecting the respective smell with a low number of examples. Some cases, such as the detection of CC using NB, are exceptions, in these cases, using a dataset containing a low number of instances did not reach high results. We can also note that all algorithms did not need more than 20 instances to reach effectiveness above 0.8 for LC and GC smells.

The analysis of these results provides us with the answer to the **RQ2**. The results show that, in most cases, **the techniques did not need a higher number of examples to reach their best detection results. In fact, the increase in the number of instances in the training set did not appear to have a direct relationship with the increase in effectiveness for all ML techniques**. These results contradict those results found in previous studies (27, 30, 77), which stated that the techniques needed a large number of examples to get good results.

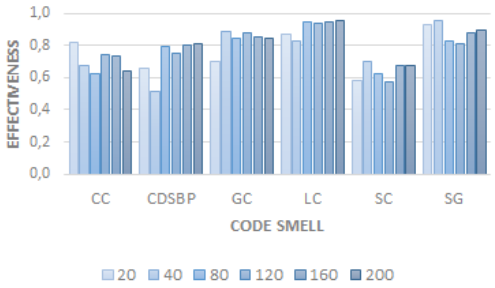


Figure 3.3: J48 Efficiency

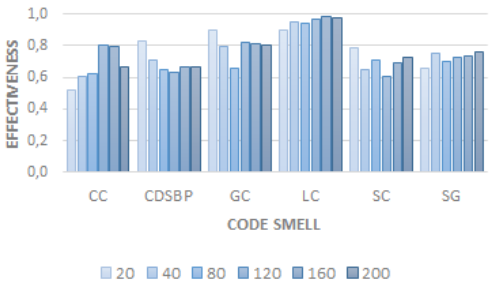


Figure 3.4: NB Efficiency

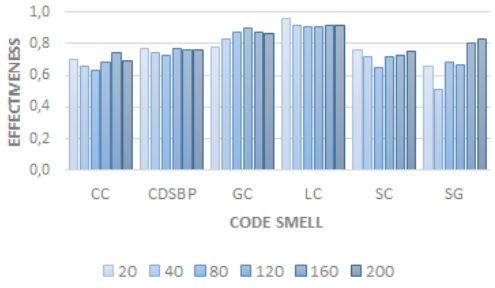


Figure 3.5: SVM Efficiency

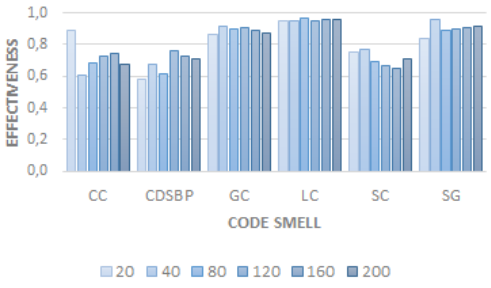


Figure 3.6: OR Efficiency

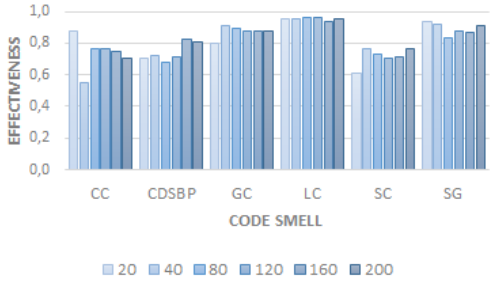


Figure 3.7: JRip Efficiency

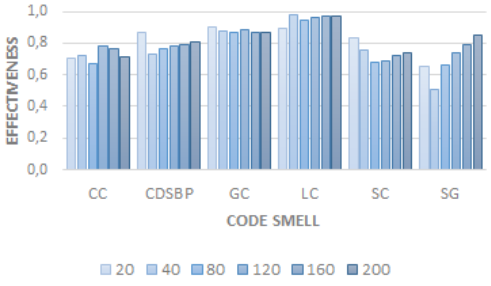


Figure 3.8: RF Efficiency

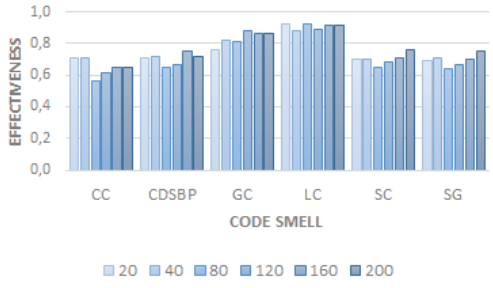


Figure 3.9: SMO Efficiency

3.5 Limitations and Threats to Validity

3.5.1 Threats to Validity

This section discusses the threats to validity in accordance with the criteria defined by Wohlin *et al.* (72).

Construct Validity. The datasets that supported our study were built from code fragments collected using rule-based strategies that have a set of metrics and thresholds. These thresholds are threats, once they can bias the techniques learning because of the analyzed smelly fragments were filtered by these thresholds. To lessen this bias, we filter the smells by selecting only those that caught the developer's attention to refactor them.

Another important factor to observe is that these datasets were built including the same metrics for all smell types. However, we can note that each smell affects a specific domain of the software and not all metrics are directly related to that domain. Therefore, ignoring this relation between the chosen metrics and the software scope can affect the results. Although the chosen smell types affect different scopes of the analyzed project, their importance was not taken into account during the choice. It is also possible to observe that specific smells were not studied for a specific scope, *e.g.*, such as smelly structures affecting only the internal body of a method.

Internal Validity. The use of the Weka package of the R platform to implement the techniques analyzed in our study enabled us to experiment a variety of configurations, which affect the training process of the techniques. In such context, the configurations considered in our experiments may impact the effectiveness and efficiency of the techniques. In order to mitigate this threat, we configured all ML techniques according to the better settings defined in (30). Indeed, (30) performed a variety of experiments in order to find the best adjust for each technique.

External Validity. The code fragments were extracted from ten Java projects. However, although the implementation of these projects presents classes and methods with different characteristics (*i.e.*, size and complexity), our results might not hold to other projects.

3.5.2 Limitations

This section discusses the limitations found during the study, which will be considered in future studies.

Number of Smells. The catalog of smell types presented in (46) categorizes the smells based on their area of action in the code, besides defining a high number of smell types than those addressed in our empirical study. These additional smells can also harm the quality of the software, making their detection important. However, their detection through machine learning requires the evaluation of code fragments that are suspicions of containing these smells, which leads us to the second limitation.

Evaluated Projects. Ten different projects are currently covered in our dataset. However, all of these projects are open source projects written with the Java programming language. These common characteristics among the chosen projects tend may reduce the variety of particular manifestations of a smell type. A larger dataset, including both closed source and additional open source projects, can expose a wider variety of smell structures.

Classifier Model Customization. We observed that each ML technique did not support general, highly-accurate detection of all smell types. However, the achieved an improvement in their overall effectiveness when are analyzing a subset of specific smell types. This improvement could be related to the classifier model built by the techniques. It is important to note that this model can be improved manually changing the parameters during the technique implementation, or automatically through trial and error. Previous studies (27, 77) suggest that this improvement by customization could also be explored to better detect smells for specifics developers.

Project Sensitive Customization. Better behavior of a ML technique perhaps could also be observed if the training and the detection involves a single software project. Given this narrower scope, we would reduce the number of developers involved in the dataset. Thus, the ML techniques may be able to better adapt themselves during the training process. If we further narrow the scope to the system's modules, we will have code fragments with similar responsibilities and a subset of developers in charge. This change may allow the techniques to customize their detection for the specific concerns being addressed by each module, hopefully further improving their effectiveness but in detriment of possibly not having a reasonable number of smell instances to properly train the model.

3.6 Conclusion and Future Work

This study presented a study that analyzed the effectiveness and efficiency of ML techniques for detecting code smells. Firstly, we evaluated the overall effectiveness of the ML techniques to detect smells. Then, we analyzed the efficiency of the ML techniques by evaluating their effectiveness according to the number of examples used to perform the training process.

The results indicated that while the *JRip* and *RF* reached the highest overall effectiveness on detecting smells, the NB obtained the lowest performance. We also observed that all the analyzed techniques are sensitive to the smell type analyzed, as observed in previous studies.

Regarding the techniques' efficiency, we observed a different result from

previous studies, where the increase in the number of instances in the training set did not appear to have a direct relationship with the increase of effectiveness. Once the ML techniques do not need a high number of examples to reach their best results, the effort to train the techniques is reduced, enabling the use of this technique in projects with different sizes. We can also conclude that the dataset is possibly linked to the result obtained by the techniques, in addition to the interference in the number of examples needed in the training to obtain them.

As future work, we intend to investigate the effectiveness of ML techniques on detecting other smell types. In addition, we also intend to replicate this study in controlled scenarios, reducing the analyzed scope per project and, after that, per system's modules. In this way, we expect to identify the behavior of the techniques in more specific contexts.

3.7 Summary of Chapter 3

In this chapter, we performed a complementary study aimed at investigating the ability of ML techniques on detecting developer-sensitive smells. We built a new dataset to observe the techniques' behavior for a new training set. Also, we focused on assessing ML techniques for detecting smells that ended up being refactored by a developer of the respective software project.

Similar to Chapter 2, the results also showed that all the analyzed techniques are sensitive to the developers' knowledge. Besides, the results indicated that the ML techniques still reached high effectiveness when detecting harmful smells. This finding was observed for the vast majority of the smell types. The results also provided us evidence that some ML techniques can better customize their detection model for specific smells types. For instance, the enhancement of JRip results shown when detecting harmful smells that reached the highest effectiveness in three out six types of smells for this new dataset.

4
Customization of Refactorings

Chapters 2 and 3 present the influence that developers' knowledge has in the effectiveness obtained by ML techniques. This influence reinforces the need for supporting the automatic customization of smell detection. In a similar vein, the smell removal process is also often performed by a developer, who has the knowledge to properly perform the code refactoring. Therefore, developers are responsible to evaluate the context of the source code affected by the smell. They also have to decide which refactoring is appropriate for each context. Indeed, a previous study (15) listed various reasons that motivate developers to apply refactorings, which depend on the contextual aspects of the task and the source code being changed. Many of these reported reasons are directly or indirectly related to the removal of code smells.

Thus, the application of refactoring may need to be customized as it occurs with smell detection. The application of specific customized refactorings can be decisive to either remove, partially mitigating or introduce a code smell. However, customization of refactoring is rarely investigated, as much as the impact of customized refactorings and code smells. Thus, this chapter presents the paper *"How Do Developers Customize Refactoring in Practice?"* (44), which is being submitted to the 35th International Conference on Automated Software Engineering (ASE) in May 2020.

In this paper, we performed a retrospective study involving 13 projects, from which we identified and analyzed 1,162 refactoring instances. We focused the analysis on four of the most frequent refactoring types, namely *Extract Method*, *Inline Method*,*Pull Up Method*, and *Move Method*. These refactoring types were also selected due to the fact their application is needed to remove various smell types analyzed in the previous studies. We analyzed what code modifications developers performed along with such refactorings. This analysis enable us to reveal whether developers often have to customize such refactorings. Such modifications include adding or discarding modifications that are tailored to their program's needs. We also investigated how these refactoring customizations interact with code smells.

The analysis can help one to understand how customized refactorings: (i) reduce or increase the occurrence of a particular type of smell, (ii) reduce or

increase the intensity of the smell, that is, affect the code measures associated with a smell, making it less or more harmful than before the refactoring. Finally, we also discussed the refactoring support provided by Eclipse for the application of customizations commonly made by developers. We focused on the analysis of the Eclipse IDE at it is a very popular environment for Java development. Eclipse is frequently used in the literature on automated refactoring support (*e.g.*, (40, 41)).

How Do Developers Customize Refactoring in Practice?

A Retrospective Study of 13 Software Projects

Daniel Oliveira
PUC-Rio, Rio de Janeiro, Brazil
doliveira@inf.puc-rio.br

Ana Carla Bibiano
PUC-Rio, Rio de Janeiro, Brazil
abibiano@inf.puc-rio.br

Kleber Tarcísio
UFCG, Paraiba, Brazil
klebertosantos@gmail.com

Alessandro Garcia
PUC-Rio, Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br>

Baldoino Fonseca
UFAL, Alagoas, Brazil
baldoino@ic.ufal.br

Márcio Ribeiro
UFAL, Alagoas, Brazil
marcio@ic.ufal.br

Anderson Santos
PUC-Rio, Rio de Janeiro, Brazil
asilva@inf.puc-rio.br

Paulo Bernardo
UFAL, Alagoas, Brazil
pbaf@ic.ufal.br

Rohit Gheyi
UFCG, Paraiba, Brazil
rohit@dsc.ufcg.edu.br>

4.1 Introduction

Code refactoring is a key technique to promote program comprehensibility, maintainability and other quality attributes. Each code refactoring type is composed of one or more primitive modifications that aim at improving program structure, thereby facilitating program comprehension and further program changes. For instance, let us consider the *Extract Method* refactoring, which is the most common refactoring type according to recent studies (2, 15, 80). Fowler (1) states all instances of an *Extract Method* consists of 'default' (*i.e.*, core) set of modifications in the program, including the creation of a new method based on the extraction of statements from an existing method (16). *Extract Method* refactoring reduces the method's size and the separation of the responsibility now fulfilled by the extracted method. These modifications possibly enhance the comprehensibility (1) of both the existing and the extracted method. They also facilitate the maintenance and the reuse of that segregated responsibility in further changes of the program.

However, even apparently simple refactorings, such as *Extract Method*, are often hard to be realized in a software project (9, 17). Developers may need to customize the set of core modifications associated with a refactoring type. They might need to perform additional modifications or even discard the core ones prescribed in Fowler's catalogue (1), and/or those modifications induced by IDEs or tools for automated refactoring. In other words, there might be some variations of each refactoring type, including ones not documented in Fowler's catalogue (1) and not supported by popular IDEs and tools. We call each possible variation of a refactoring type as a *customized refactoring*.

Such customizations may be required to satisfy recurring developers' needs. These needs may range from the refactoring adjustment to: (i) some typical structures within and across projects to (ii) the removal of certain

smells, or (iii) even the application of complementary modifications in the clients of the refactored code. Recent empirical studies (17, 18) suggest that developers seem to not strictly follow the modifications prescribed by IDEs to realize their refactorings (41). On the other hand, there is a limited understanding of how customized refactorings occur in practice. No study has analyzed in depth the typical variations of well-known refactoring types across multiple software projects. Key questions have not been addressed by the literature, including: are the most frequent modifications of each refactoring type in line with the core modifications recommended in Fowler's catalogue? are there recurring non-default, additional modifications performed for the same refactoring type across multiple projects? Are there typical patterns of customized refactorings? Are they fully or partially supported by automated refactoring tools and detected by state-of-the-art refactoring detection techniques?

The refactoring practice is often studied, especially in terms of its impact on the program quality (*e.g.*, (1, 9, 31, 38, 78)). There are also a few studies investigating the refactorings required in certain software projects (32, 33, 34). However, there is little understanding about how developers customize refactorings. A recent study (35) also observed certain relationships between refactorings and other code changes. This study concludes that more refactorings occur in classes in which developers applied changes realizing additions of new features. However, the study does not investigate how customized refactorings are applied across projects. None of the aforementioned questions are explicitly addressed by the literature.

In order to address this gap, we have performed a retrospective study involving 13 open software projects, from which we identified and analyzed 1,162 refactoring instances. We focused our analysis on four of the most frequent refactoring types, namely *Extract Method*, *Inline Method*, *Pull Up Method*, and *Move Method*. We observed that the core modifications of each analyzed refactoring occurred frequently during the application of the respective refactoring. However, most of the refactoring instances also encompassed recurring additional modifications. These additional modifications are responsible to adjusting the refactoring to specific surrounding structures in developers' programs.

We also observed that these recurring additional modifications are not limited to core classes affected by the refactoring. In fact, we observed that refactoring often affects larger program scopes, than the ones described by Fowler's catalogue (1) and those ones covered by popular IDEs (81). We also reported several patterns of customized refactorings that are not covered by

existing refactoring tool provided by Eclipse IDE. Finally, we also analyzed the impact of these patterns on code smells. We inferred the impact of each pattern on code smells by extracting the following information for each instance of these patterns: (i) which code smell types were *introduced*; (ii) which code smell types were *removed*; and (iii) which code smell types had their intensity reduced albeit not fully removed.

This paper is structured as follows: Section 4.2 provides background information regarding key concepts about customized refactorings. It also presents a motivating concrete example of customized refactoring. Section 4.3 describes our study settings, including the study goal and experimental steps. Section 4.4 presents our findings about the frequency and how developers customized refactorings in practice. Section 4.5 discusses the threats to validity of our study. Finally, Section 4.6 concludes this paper and suggests future work.

4.2 Background

This section presents the main concepts about refactoring types and their customization.

Table 4.1: Refactoring Scope

Type	Description	source	target
Extract Method	Create a method based on statements extracted from an existing method	Method where the extraction was performed	Extracted method
Inline Method	Incorporate the body of a method into an existing method	Method to be inlined	Method that inlined the source
Pull Up Method	Move a method from a child class to its parent class	Method in the subclass	Pulled up method in the superclass
Move Method	Move a method from one class to another class	Method to be moved	Method after being moved

4.2.1 Refactoring in Practice

Code refactoring consists of applying modifications on code structures for enhancing program comprehensibility, maintainability and other quality attributes (1, 8, 9). This practice is often adopted by large companies, such as Microsoft (9), and the several other companies that adhere the agile methods. The literature cataloged different types of refactorings in order to guide the developers to enhance their code structure (1). For this study we focused on four of the most popular refactoring types. They are listed and described in Table 4.1. The last two columns describe the source element and the target element of each refactoring type. These refactoring types were chosen because they: (i) have different scopes, *i.e.*, they cover both class-level and method-level refactorings, and (ii) are among the most frequently applied refactorings in practice (2, 8, 80). For each refactoring, we defined the source

and target methods. These methods represent the main methods of each refactoring indicating the main changed method (*i.e.*, the source) and the method produced after the refactoring (*i.e.*, the target). Table 4.1 presents the source and target method of each refactoring.

Besides, each refactoring type specifies a set of core modifications that should be applied, as shown in Table 4.2. Core modifications are those described in the Fowler's catalogue and employed by state-of-the-art refactoring detection tools

Table 4.2: Refactoring Core Modifications

Type	Core Modifications
Extract Method	• Create the target method with code extracted from the source method • Update variables' references • Add in the source method's body a call to the target method
Inline Method	• Replace each call to the source method with its method body • Remove the source's method declaration
Pull Up Method	• Create target method in the superclass and copy the source's method body • Remove from all subclasses the source's method declaration • If possible, change source methods calls, with call to the target method
Move Method	• Create target method with a copy of the source's body method • If removed source's method: replace calls to target method • If did not remove source's method: add target call in source's body

In order to illustrate the core modifications of a refactoring type, lets consider the *Extract Method*, which is the most frequent refactoring type popularly adopted by developers (15, 16) after renaming. According to Fowler's catalog (1), the realization of the *Extract Method* requires to creation of a new method (target method) based on the extraction of statements from the body of the source method. Check if any variables from the extracted code need to be redeclared or passed as an argument to the extracted target method. Finally, replace all the calls to the source method with calls to the target method. To perform the *Inline Method*, all method calls should be replaced with its body and the source method declaration should be removed. To execute the *Pull Up Method*, a new method (target method) in the superclass should be firstly created with a copy of the source method's body. Then, all source method definitions should be removed from all subclasses or replaced with a call to the new superclass method. To perform a *Move Method*, a new method (target method) should be created with a copy of the method source's body. Then, removal of the source method's declaration is optional. If not removed it can be kept as a delegating method. If the source method was removed, all source method calls should be removed.

The application of a refactoring is a complex activity, which requires specialized effort (9, 82). It is because developers need to know when and where the refactoring can be applied, what refactoring type can be applied, and how to apply this refactoring type (1). Previous studies have investigated

refactoring opportunities to facilitate the application of refactorings (20, 82, 83). Another study investigated the benefits and challenges to apply refactorings (9). Studies also have proposed automated refactoring tools to support the application of refactorings (20, 84). Besides, existing IDEs such as Eclipse also provide the automated application of some refactoring types such as *Extract Method* and *Move Method*.

These studies and tools are limited to respectively analyze and support how developers apply refactorings even considering only the core modifications involved in each refactoring type. However, a recent study conducted by Oliveira *et al.* (18) indicates that developers customize refactorings according to their development context and existing automated refactoring tools do not support this customization. By customized refactoring, we mean a refactoring (i) involving core modifications and at least an additional modification, i.e, modifications not considered in the Fowler's catalogue and state-of-the-art refactoring detection tools; or (ii) removing a core modification.

4.2.2 Understanding Refactoring Customization

Although a recent study (18) indicates that developers have applied customized refactorings manually according to their development context, the knowledge about customized refactoring in practice is limited. The literature does not systematically investigate (i) what are the code modifications in customized refactorings, and (ii) how developers apply customized refactorings for each refactoring type. This lack of knowledge also applies to the most common types of refactoring such as *Extract Method* and *Move Method* (15). Figure 4.1 presents a real example of customized refactoring, which occurred in a open source project.

The example presents a customized *Move Method* that was applied on the Apache Tomcat software project in the commit c_i = f69c17895.[1] In that case, the developer manually moved a method called setAllowCasualMP[2] from the Connector class to the StandardContext class. This example is composed of the following modifications: (i) a method was moved from a class to another class, and (ii) a method signature of this method was created on the interface (Context) of the target class. The first modification is a core modifications of the *Move Method* defined in Table 4.2.

However, the second modification is an additional one that customize the *Move Method*. This additional modification moved the setAllowCasualMP

[1] `https://github.com/apache/tomcat/commit/f69c17895`. Access Date: 03/03/2020

[2] The method name was adapted due to paper short space. The real name is setAllowCasualMultipartParsing.

method to the target class and made it and abstract method of the interface implemented by the target class. This additioned modification is really important, because there is a test class (TestStandardContext) that calls the setAllowCasualMP method directly from the Context interface. Therefore, the additional modification is essential for the correct functioning of the project. Otherwise, the project would have a compilation error.

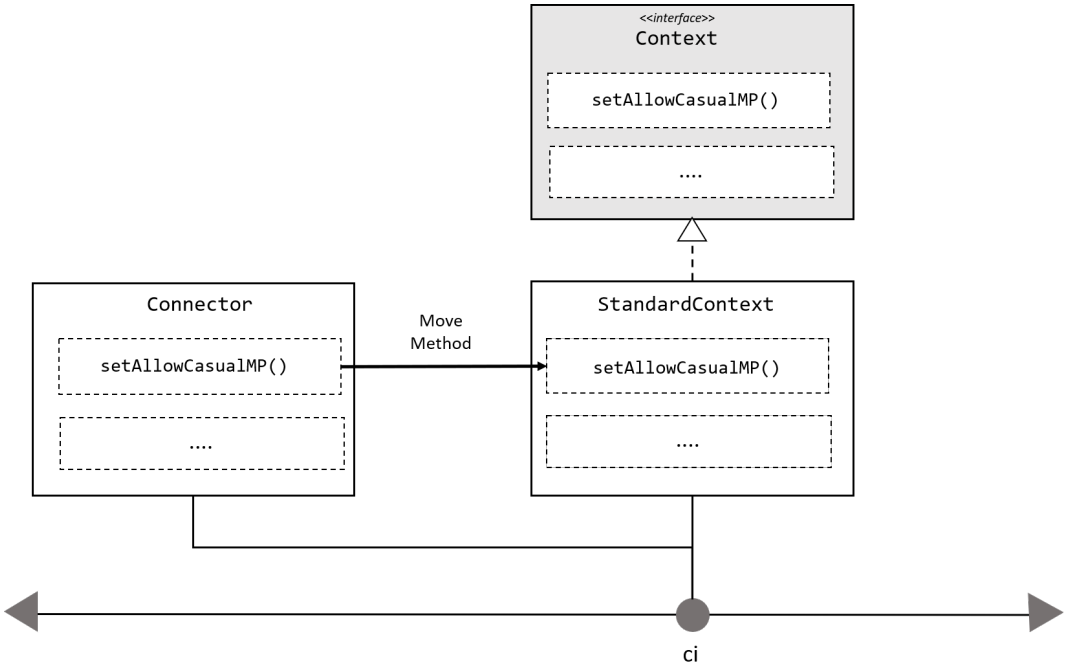


Figure 4.1: Real Example of Customized Refactoring

This customization of *Move Method* is not supported by the IDEs, such as Eclipse and IntelliJ, and any other tools for supporting refactoring. These existing solutions only allow moving a method to other class with a very limited flexibility. For instance, Eclipse's refactoring automated tool only allows developers to change both the method's name and parameters' name, and the target class. This limited flexibility may force developers to manually apply the full or partial set of modifications of a refactoring, which is cumbersome and error-prone (9). Thus, existing tools would better cover developers' needs if they are designed to support a comprehensive catalog of mutable set of code modifications for enabling customized refactorings. However, in order to better understand the developers' need, we need to properly investigate whether, to what extent and how developers perform alternative modifications along their customized refactorings. In case there is a wide range of recurring customized refactorings, a catalog can help to better characterize the typical core and additional modifications of each customized refactoring patterns. This catalog can also better inform tool designers to

support developers with proper customization features.

4.3 Study Settings

Developers are responsible to analyze various contextual aspects of the code, which a possible target for applying a refactoring. This analysis is required to enabling the developer to decide which refactoring – including its type and code modifications – is appropriate for a given context. However, developers may feel uncertain about the appropriate refactoring decisions within specific contexts. In addition, the tools may not provide adequate support for the specific customization that the developers need. To make it worse, the erroneous choice of refactoring modifications can induce developers to not reach the results in principle expected, or even somehow induce the (possibly unconscious) introduction of code smells.

Given the contextual particularities of where a certain refactoring type is applied, which may be frequent across projects, certain customized refactorings may also occur with a non-negligible frequency. However, the empirical knowledge about customized refactorings is quite limited. In this sense, we investigated how developers have applied customized refactorings in practice. To reach this goal, we detected and analyzed the possible customizations for each of four common types of refactoring. This study was performed along a retrospective study that identified the frequent code modifications made in the context of the analyzed refactoring types.

We also distinguished what are the core and additional modifications that tend to be applied together in instances of a refactoring type. Whenever a certain grouping of the same modifications occur together with a certain frequency, we call it a customization pattern (or simply pattern). These patterns may reveal insights about how developers apply and customize refactorings on their projects. We also discuss occurrences of interesting common patterns and to what extent they are supported by existing IDEs and other tools. Finally, we can also discuss the impact on the code structure quality after instances of a customization pattern were applied in a code change. The analysis of the code structural quality was based on the observation of smells either introduced or removed within the scope of the customized refactoring.

Our study findings can reinforce potential requirements on the design and improvement of refactoring tools. They can also provide insights on the design of recommenders for assisting developers in properly selecting custom code modifications along a refactoring. In this way, we split our study in three research questions:

RQ1: *What code modifications developers perform when refactoring?*

This RQ focuses on identifying the most frequent code modifications related to each analyzed refactoring type. We then investigated the code modifications that compose the refactorings across projects and the frequency of the each of such modifications. Finally, we also observed where the modifications occur within the refactored code, in particular, if the modifications are within the source and target methods (or, alternatively, in the surrounding code) of each analyzed refactoring type (Table 4.1).

RQ2: *How often developers apply customized refactorings?*

The second RQ aims at investigating the most common customization patterns for each refactoring type and the frequency that these patterns occur. The results of this RQ provided a catalogue of the most frequent patterns for each refactoring. Finally, we also investigated if the existing IDE-based automated tools support the application of frequent patterns. Therefore, our results can suggest how those existing IDE tools can improve the support for developers performing customized refactorings.

RQ3: *Does customized refactoring reduce the intensity of code smells?*

Finally, in our last research question aims at investigating the impact of each pattern on code smells. We wanted to observe whether each customization pattern consistently reduce or increase the intensity of code smells; and additionally if they remove or introduce particular instances of a specific smell type. This analysis is important to better guide developers on performing not-harmful customized refactorings. Developers may wish to be warned on the potential negative impact on the code structure when they are making certain refactoring customization decisions. Developers may also receive recommendations of alternative customized refactorings, which are beneficial to the code structure and still helping them to achieve their non-structural goals in the task at hands.

4.3.1 Study Design

This section details the five steps performed in the preparation of our study dataset. These steps are illustrated in Figure 4.2. We indicated each step by using a number from 1 to 5.

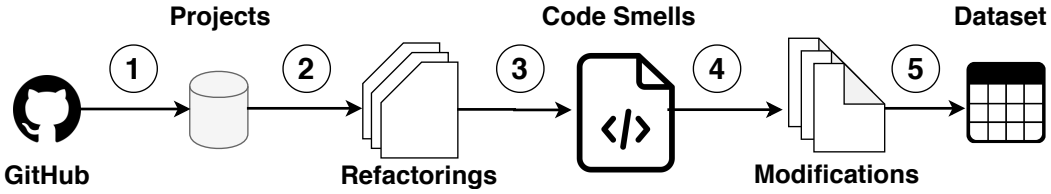


Figure 4.2: Study Design Steps

Step 1: Project Selection. We selected 13 projects based on the following criteria. First, the projects must be open source, being available in some GitHub repository. This decision is intended at facilitating the replication of our study by other researchers. We then ordered all open software projects by the stars count. The number of starts is given by developers who have access to the project in GitHub. This filter is useful to indicate active and popular projects (86). We filtered the top-100 projects based on this criterion. Then, we selected the software projects that have at least 90% of source code written in Java, due to the characteristics of the robust, highly-accurate tools used to build our dataset. Besides that, we performed a manual evaluation of these projects to satisfy the following criteria: our set of projects should be of different sizes, domains, and often used in previous studies about refactoring (2, 15, 80, 87, 88). The latter is important to ensure other desirable complementary criteria, used in those previous studies, also affected our selection of projects. Moreover we could eventually position our findings under the perspective of previous findings about software refactoring.

Step 2: Refactoring Detection. We used the Refactoring Miner (15, 38) tool to detect refactorings that occurred in the selected projects. Refactoring Miner is widely used in the literature (2, 15, 80, 38). This tool has a satisfactory accuracy: 87.2% of recall and 98% of precision (15, 81). In order to enable an in-depth analysis, we focused only on 4 refactoring types (Table 4.1), which are frequent in practice (2, 15). It is possible to observe that we did not include refactorings with wider scope; for example, those refactorings affecting a high proportion of classes located in a package or multiple packages). We decided to focus on refactorings with narrower scope because otherwise there would be a increasing likelihood of noise related to other modifications, which are not related at all to the refactoring, across the affected packages. This noise would undesirably interfere our study outcome in a significant manner. In any case, those refactoring types are not frequently used in practice (2, 8, 80) and, in fact, they were quite rare in our dataset.

Step 3: Code Smell Detection. We used a smell detection tool that was already used by previous studies about refactoring (2, 80). This tool implements smell detection strategies based on software metrics. These strategies were validated by previous work (89) with a resulting precision and recall (90) of 72% and 81%, respectively. Altogether, we collected 17 types of code smells in this study. We collected the smells before and after the application of each refactoring detected in Step 2. Finally, we classified a smell as *introduced* if the smell occurred in an element affected by a refactoring. Similarly, a code smell is considered as *removed* if the smell was cleared after the refactoring. Finally, a smell can be considered as *mitigated* if at least one of the metrics used to detect the smell had its corresponding value been improved, without worsening any of the other measures used in its detection. In other words, a mitigated smell is one in which its intensity is somehow reduced (regardless the degree of reduction). We use the term *mitigation* as the anomalous nature of a certain smell was just reduced to make the developers' task possible. The complete list of detected smells is described in Table 4.3.

Table 4.3: Smell Types Analyzed

Smell Type	Definition
Brain Method	Method overloaded with software features
Dispersed Coupling	Method that calls too many methods
Feature Envy	Method "envying" other classes' features
Intensive Coupling	Method that depends too much from a few others
Long Method	Too long and complex method
Long Parameter List	Too many parameters in a method
Message Chain	Too long chain of method calls
Shotgun Surgery	Method whose changes affect many methods
Brain Class	Class overloaded with software features
Class Data should be Private	Class that overexposes its attributes
Complex Class	Too complex software features into a class
Data Class	Only data management features into a class
God Class	Too many software features into a class
Lazy Class	Too short and simple class
Refused Bequest	Child class rarely uses parent class features
Spaghetti Code	Too much code deviation and nesting
Speculative Generality	Useless abstract class

Step 4: Modification Detection. We detected modifications made by developers while applying refactorings. We used a library from Eclipse's JDT 3.10.0. to collect these modifications [3]. This library allows us to turn Java source code into an Abstract Syntax Tree (AST) through a parser. ASTs are widely used in the literature to detect refactorings (81, 91, 92). The Eclipse AST parser is the base library used by Eclipse for many powerful tools, including their current automated refactoring tool. This library can capture the semantic structure of a Java program, allowing to identify and perform code modifications. For instance, we have the METHOD_INVOCATION node from Eclipse AST. As the

[3] `https://help.eclipse.org/mars/index.jsp`. Access Date: 03/03/2020

name suggests, the code fragments that are classified in this node are method calls. Another example is the node METHOD_DECLARATION. This node appears in the AST when occurs a declaration of a method.

Then, for each detected refactoring in step 2, we collected the information from the two relevant versions, *i.e.*, before (v) and after (v_{+1}) the refactoring occurrence. We collected information related to the classes affected by this refactoring and their clients. We classified a class as affected by a refactoring when the refactoring modifications occurred within that class. For instance, an *Extract Method* will have only one affected class. On the other hand, a refactoring of the type *Pull Up Method* or *Move Method* will have at least two affected classes, once a method will be moved from a class to another one. Finally, we classified as a client of a class every class that interacts with (*e.g.*, importing it and/or calling a method of) the affected class.

Once we have two subsequent versions of a class, we can define the code modifications as follow:

$$AST_v = \{modification_i, modification_{i+1}, ..., modification_n\} \quad (4\text{-}1)$$

Let AST_v be the set of modifications obtained by the AST in version v. The set of modifications added to the source code between two subsequent versions is given by the resulting set of the difference between $AST_{v+1} - AST_v$. Similarly, the set of modifications removed from the source code is given by the difference of $AST_v - AST_{v+1}$.

Figure 4.3 illustrates the diff between two subsequent versions of a class from Facebook Fresco project.[4] The lines with green highlight indicate the added statements. Similarly, the lines with red highlight indicate the removed statements. Table 4.4 presents a partial list of modifications obtained when analyzing the code presented in Figure 4.3. The list includes high granularity modifications. The first column presents the modification type. The second one describes the statement that characterizes this modification. The third column indicates the code element where the modification occurred. Finally, the last column indicates if the modification was an addition or removal.

Table 4.4: Modification List

Modification Type	Statement	Element	Status
METHOD_DECLARATION	Animated...BackendWrapper.clear()	Class	Addition
IF_STATEMENT	mAnimated...Backend != null	clear()	Addition
IF_STATEMENT	mAnimated...Backend != null	onInactive()	Removal
METHOD_INVOCATION	CloseableReference.closeSafely(m...Frame) mAnimated...Backend.dropCaches()	onInactive()	Removal
METHOD_INVOCATION	CloseableReference.closeSafely(m...Frame) mAnimated...Backend.dropCaches()	clear()	Addition
METHOD_INVOCATION	clear();	onInactive();	Addition

[4] https://github.com/facebook/fresco/commit/2d82c6c185. Access Date: 03/03/2020

```
111 +   @Override
112 +   public void clear() {
113 +     if (mAnimatedDrawableCachingBackend != null) {
114 +       mAnimatedDrawableCachingBackend.dropCaches();
115 +     }
116 +     CloseableReference.closeSafely(mLastDrawnFrame);
117 +     mLastDrawnFrame = null;
118 +   }
119 +
...  ...
121 130     @Override
122 131     public void onInactive() {
123     -     if (mAnimatedDrawableCachingBackend != null) {
124     -       mAnimatedDrawableCachingBackend.dropCaches();
125     -     }
126     -     CloseableReference.closeSafely(mLastDrawnFrame);
127     -     mLastDrawnFrame = null;
    132 +     clear();
128 133     }
```

Figure 4.3: Modifications between Two Subsequent Versions

First, we have a METHOD_DECLARATION indicating the creation of the method CLEAR(). Then, we can observe that some statements belonging to ONINACTIVE(), characterized as IF_STATEMENT and METHOD_INVOCATION modifications, were removed. These same modifications were added in the method CLEAR(). In addition, a new method call to the method CLEAR() was added in the method ONINACTIVE(). This new call is also characterized as METHOD_INVOCATION.

The complete list of possible modifications that can be detected using AST can be found in Eclipse's JDT documentation.[5] However, to avoid duplicated data, we disregard modification types that contain statements already presented at a higher level granularity modification. Thus, for our study, we will use only the modification types listed in Table 4.5. In this table, we also grouped the modification types based on their similarities. For instance, the modification types CONDITIONAL_EXPRESSION, IF_STATEMENT, SWITCH_CASE and SWITCH_STATEMENT were grouped. These modifications types are all related to conditional control, thus we grouped them into a category named Conditional.

Step 5: Dataset Construction. In the previous step, we obtained the collected modifications associated with affected classes and their clients. However, some refactorings do not affect all elements of a class. For instance, Fowler (1)

[5] `https://help.eclipse.org/mars/index.jsp`. Access Date: 03/03/2020

Table 4.5: Grouped Modifications

Modification Category	Modification Types
Annotation	ANNOTATION_TYPE_DECLARATION, ANNOTATION_TYPE_MEMBER_DECLARATION, MEMBER_VALUE_PAIR, QUALIFIED_TYPE, NAME_QUALIFIED_TYPE, MARKER_ANNOTATION, NORMAL_ANNOTATION, SINGLE_MEMBER_ANNOTATION
Enum	ENUM_DECLARATION, ENUM_CONSTANT_DECLARATION
Method Declaration	FIELD_DECLARATION, METHOD_DECLARATION, INITIALIZER, LAMBDA_EXPRESSION, MODIFIER
Exception Handler	TRY_STATEMENT, CATCH_CLAUSE, THROW_STATEMENT, UNION_TYPE
Comments	JAVADOC, BLOCK_COMMENT, LINE_COMMENT, METHOD_REF, METHOD_REF_PARAMETER, MEMBER_REF, TAG_ELEMENT, TEXT_ELEMENT
Array Modifier	ARRAY_CREATION, ARRAY_INITIALIZER, ARRAY_ACCESS, ARRAY_TYPE, DIMENSION
Literal Modifier	BOOLEAN_LITERAL, CHARACTER_LITERAL, NULL_LITERAL, NUMBER_LITERAL, STRING_LITERAL, TYPE_LITERAL
Class Creation	CLASS_INSTANCE_CREATION, ANONYMOUS_CLASS_DECLARATION, TYPE_PARAMETER, CREATION_REFERENCE, TYPE_METHOD_REFERENCE
Conditional	CONDITIONAL_EXPRESSION, IF_STATEMENT, SWITCH_CASE, SWITCH_STATEMENT
Method Access	FIELD_ACCESS, METHOD_INVOCATION, SUPER_FIELD_ACCESS, SUPER_METHOD_INVOCATION, THIS_EXPRESSION, CONSTRUCTOR_INVOCATION, SUPER_CONSTRUCTOR_INVOCATION, EXPRESSION_METHOD_REFERENCE, SUPER_METHOD_REFERENCE
Operator Expression	INFIX_EXPRESSION, POSTFIX_EXPRESSION, PREFIX_EXPRESSION, ASSIGNMENT
Cast	INSTANCEOF_EXPRESSION, CAST_EXPRESSION, INTERSECTION_TYPE
Variable Declaration	VARIABLE_DECLARATION_EXPRESSION, VARIABLE_DECLARATION_FRAGMENT, VARIABLE_DECLARATION_STATEMENT, SINGLE_VARIABLE_DECLARATION
Class Control	IMPORT_DECLARATION, PACKAGE_DECLARATION
Loop Flow Control	DO_STATEMENT, FOR_STATEMENT, BREAK_STATEMENT, CONTINUE_STATEMENT, ENHANCED_FOR_STATEMENT, WHILE_STATEMENT
Type Modifier	SIMPLE_TYPE, TYPE_DECLARATION, TYPE_DECLARATION_STATEMENT, PRIMITIVE_TYPE, PARAMETERIZED_TYPE, WILDCARD_TYPE
Return Modifier	RETURN_STATEMENT

suggests that the modifications related to the refactoring *Extract Method* occur mainly in both the source method (the method that suffered the extraction) and the target method (the method built from the extracted code).

Then, we disregard modifications that are not related to the refactoring's source and target method. For instance, if we are detecting modifications related to an *Extract Method* refactoring, we consider as acceptable: (i) modifications that happened within the source and target methods, and (ii) modifications that somehow interact with the source method or the target method. An example of interaction is when a method, not the source one, contains a METHOD_INVOCATION calling the extracted method.

Finally, we divided the modifications into 5 groups according to the modification's location. First, the INSOURCE group corresponds to the set of modifications made in the source method. Second, the INTARGET group

is formed by the set of modifications made in the target method. Third, the SUR group consists of surrounding modifications, *i.e.*, those that satisfy all the following conditions: (i) they are made in other methods beyond the source and target methods, (ii) they are located in methods belong to the same classes in which the source and target methods are declared, and (iii) the modifications interact directly with the source method (in this case they are also part of the SUR_S subgroup) or the extracted method (in this case they are also part of the SUR_T subgroup).

For instance, in the *Extract Method* refactoring, let us consider that a method A.A() had some of its statements extracted into a method A.B(). Then, if a method A.C(), belonging to the same class, has a modification that interacts with method A.A(), this modification will belong to the subgroup SUR_S. We also have the CLASS group and the EXT group. The CLASS group is composed of modifications that occurred in the same class(es) that contains the source and target methods and interact with the source (CLASS_S) or target method (CLASS_T). However, different from the SUR group, the CLASS group is composed of modifications at the class level. Finally, the EXT group is composed of modifications outside the the target and source classes and interact directly with the source method or the extracted method. The EXT group is also divided into EXT_S and EXT_T.

Altogether, we found hundreds of refactoring instances and nearly 100K modifications related to those refactorings. We found the following amount of instances and modifications for each refactoring type: (i) 856 instances and 77,306 modifications related to *Extract Method*, (ii) 174 instances and 14,126 modifications related to *Inline Method*, (iii) 78 instances and 5,856 modifications related to *Move Method*, and (iv) 54 instances and 3,734 modifications related *Pull Up Method*.

4.4 Results and Discussion

The following subsections present the results in terms of each of our research questions (Section 4.3).

4.4.1 RQ1: What code modifications developers perform when refactoring?

For RQ_1, we gathered the most frequent modification categories for each evaluated refactoring type. We collected both additions and removals of program elements. Moreover, we specified the location (see Section 4.3.1, step 5) and percentage of occurrence of each modification. Tables 4.6 and 4.9

list the five most frequent modifications, either additions or removals. The first column describes the change type, the second and third columns present the location and occurrence of the modification, respectively. We present the modification occurrences in terms of the percentage of instances that have at least one modification of the respective type.

Table 4.6: *Extract Method* Common Modifications

Addition			Removal		
Modification	Local	Occurrence	Modification	Local	Occurrence
Type Modifier	INTARGET	99.77%	Method Access	INSOURCE	94.74%
Method Access	INSOURCE	99.18%	Operator Expression	INSOURCE	72.31%
Method Declaration	CLASS_T	98.48%	Variable Declaration	INSOURCE	67.99%
Method Declaration	INTARGET	96.26%	Conditional	INSOURCE	63.90%
Method Access	INTARGET	94.74%	Type Modifier	INSOURCE	62.38%

When applying an *Extract Method*, one would expect the developer would perform the three steps as part of the core modifications, described in Table 4.2. One of the steps is the creation of a new method throughout the code extraction in the source method. In fact, we observed that the developers added this call in 98.48% of the instances, which corresponds to the addition of a Method Declaration located in CLASS_T (*i.e.*, the third row of Table 4.2). After performing a manual validation in the remaining instances (*i.e.*, 1.52%), we observed that the extracted code statements were added to methods already present in the affected class.

This result reveals that **even though the Method Declaration can be considered a core modification for the *Extract Method*, this modification does not occur in 1.52% of the cases in which this refactoring type was applied**. In particular, the developers responsible for this customized refactoring added code statements, which were extracted from a larger method, to a method with fewer lines of code. In such cases, the body of the target method is the result of the merge the existing code with the extracted one. As a next step, the source method was modified to call the target one. As a consequence of this invocation, the source method is now performing all the statements of the target method, even those that were located in the target method before the refactoring.

The invocation of the target method from the source one (represented by a Method Access located in the source method, *i.e.*, INSOURCE modification – represented in the second row of the table) is also one of the core modifications for *Extract Method*. However, this modification did not occur in 100% (instead, 99.18%) of the *Extract Method* instances. We manually analyzed the remaining instances, revealing that they were either: (i) false positives from Refactoring

Miner (15, 38) tool; or (ii) the source method already called the target method before refactoring. These cases were the same ones identified in the validations involving the lack of Method Declaration of the target method.

An observation of Table 4.6 also reveals that the most frequent addition modification type affects primitive types (Type Modifier modification), which has been added to the target method in 99.79% of *Extract Method* instances. On the other hand, the Type Modifier modification was removed from the source method only in 62.38% of the cases. The significant difference between the number of additions and removals of Type Modifier indicates that this kind of modification is often an additional modification that occurred during the *Extract Method*. Finally, we observed that there is a frequent removal of method calls in the source method. These method calls are also added in the target method, which means that this statement is frequently extracted from the source and moved to the target method as part of the *Extract Method* refactoring.

Finding 1: In 1.52% instances of the *Extract Method*, the extracted methods already existed before the refactoring. This observation was not expected as one usually expect the a new method is always created.

Table 4.7: *Inline Method* Common Modifications

Addition			Removal		
Change	Local	Occurrence	Change	Local	Occurrence
Method Access	INTARGET	90.23%	Type Modifier	INSOURCE	98.85%
Operator Expression	INTARGET	77.59%	Method Access	INTARGET	95.40%
Conditional	INTARGET	67.24%	Method Declaration	CLASS_S	94.25%
Variable Declaration	INTARGET	64.94%	Method Declaration	INSOURCE	93.68%
Type Modifier	INTARGET	58.62%	Method Access	INSOURCE	92.53%

Table 4.7 shows us the most frequent code modifications performed by developers when applying an *Inline Method* refactoring. Table 4.2 reveals that one of the core modifications for *Inline Method* is the removal of the source method, represented by the removal of the source method declaration (represented by the modification Method Declaration located in CLASS_S). However, this modification occurred in 94.25% of instances. **In other words, 5.75% of the instances of *Inline Method* did not remove the source method**. Indeed, a recent study (41) presented that some IDEs, by default, do not remove the source method when applying *Inline Method* by their automated refactoring tool. To make matters worse, this study also indicates that developers prefer the complete removal of the source method.

Finding 2: A rate of 5.75% did not remove the inlined method.

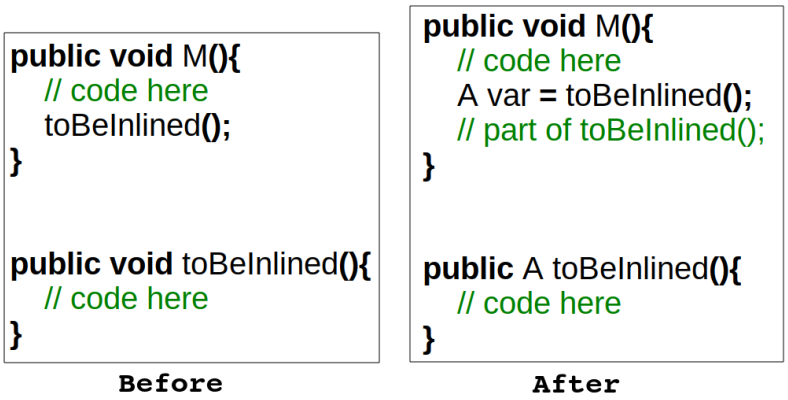


Figure 4.4: *Inline Method* that did not Remove Source Method

Regarding the finding 2, Figure 4.4 presents a real case of the *Inline Method* application[6] in which the source method has not been removed after the refactoring. The developer's intent clearly concerns the appliction of the refactoring, as mentioned in the commit message: *"Refactor to avoid a deadlock caused by different sections of code obtaining the same locks in a different order."*. In this case, only part of the source method was used at the target method, and the call to the source method was not removed. Besides, the source method's body has been modified. However, it remained with 71.00% (lines of code) of the source method's as compared to previous version of this method before the refactoring.

The analysis of Table 4.7 also shows that method calls are frequently added to the target method, *i.e.*, 90.23% of all occurrences. Indeed, in 92.53% of the instances, the method calls are removed from the source method as well. Nevertheless, we did not observe this behavior for the remaining addition modifications, *i.e.*, the additions to the target method do not correspond to the removals from the source one.

Table 4.8: *Move Method* Common Modifications

Addition			Removal		
Change	Local	Occurrence	Change	Local	Occurrence
Type Modifier	INTARGET	98.72%	Method Declaration	CLASS_S	98.72%
Method Declaration	CLASS_T	93.59%	Type Modifier	INSOURCE	97.44%
Method Declaration	INTARGET	93.59%	Method Declaration	INSOURCE	94.87%
Method Access	INTARGET	85.90%	Method Access	INSOURCE	85.90%
Variable Declaration	INTARGET	84.62%	Variable Declaration	INSOURCE	80.77%

[6] `https://github.com/apache/tomcat/commit/2561dfccb`. Access Date: 03/03/2020

Table 4.8 presents the 5 most frequent modifications applied by developers when performing a *Move Method* refactoring. When applying this refactoring, one expects the developer would remove the source method and add the target method, where the latter has a similar or equal content to the former (*i.e.*, the same method now located in a different class). However, we noticed that the removal of the source method declaration, a core modification in Table 4.2, occurred in 98.72% of the cases. The only case in which it was not possible to detect this modification occurred in the Netty[7] project. In this specific case, the developers moved the validateHeaderName(String) method. This method was implemented in the inner-class HttpHeaders. This inner-class was instantiated in the DefaultHttpChunkTrailer class. This peculiar case is a false negative in the detection of Refactoring Miner (15, 38).

Another core modification is the declaration of the target method (represented by the modification Method Declaration located in CLASS_T – the second line of addition side). However, this modification did not occur in 6.41% of *Move Method* instances. For these cases, two situations occurred: (i) the method was moved to a class where there was already an abstract method of the same declaration, and; (ii) the method was moved to a class that already had an implemented method of the same name and parameters. This observation reinforces that, differently from what is mentioned in Table 4.2, it is not necessary to create a method when making a move. To be considered a *Move Method*, only the content of the method being transferred to the new location is necessary. The existence of an abstract method with the same name and parameter as presented in situation (i) is, in fact, common, as seen in our finding 3.

> **Finding 3:** In 17% of the instances, when the method is moved to a class that belongs to a hierarchy, this modification requires the creation of a method with the same name and parameters. These alternative and recurring modifications of various *Move Method* instances are not supported in existing tools and IDEs.

This potentially happened because the method has been moved to a hierarchy and was overridden by some subclass. We illustrated a real case for this scenario in Section 4.2. Similarly, in 23% of the instances, when the method was moved out from the class hierarchy, it leads to the removal of methods with the same name and parameters. For example, a method has been moved to the outside its original hierarchy. This method was abstract

[7] `https://github.com/netty/netty/commit/4ede085edcd`. Access Date: 03/03/2020

(at the superclass) and implemented at the subclass. In this situation, the *Move Method* has been considered for the superclass, but the method has been removed from the subclass.

In Table 4.8, we also noticed that different from the previous refactorings the most frequent removal of INSOURCE modifications correspond to the addition of INTARGET modifications. Nevertheless, we can still observe a higher frequency of additions when compared to removals. This reinforces one more time the application of refactorings together with other code modifications.

Finally, in 13 cases (16.7%), the target method has a different name or parameters. In these cases, the target method usually has a body similar to the source method's. However, there are modifications added in target method that may justify the changes in the name and parameters.

Finding 4: For *Move Methods*, 16.70% of the instances had modifications in their target method's names or parameters.

Table 4.9: *Pull Up Method* Common Modifications

Addition			Removal		
Change	Local	Occurrence	Change	Local	Occurrence
Method Access	INTARGET	90.74%	Method Declaration	CLASS_S	100.00%
Type Modifier	INTARGET	90.74%	Type Modifier	INSOURCE	100.00%
Method Declaration	INTARGET	85.19%	Method Declaration	INSOURCE	98.15%
Method Declaration	CLASS_T	79.63%	Method Access	INSOURCE	87.04%
Literal Modifier	INTARGET	70.37%	Literal Modifier	INSOURCE	72.00%

Table 4.9 shows the most frequent modifications when applying *Pull Up Method* refactorings. We observe that, similarly to *Move Method*, the addition modification performed in the target method corresponds to a removal modification performed in the source method, although their occurrences are slightly different. We also observed that, differently from the *Move Method*, the source method removal (represented by the modification Method Declaration located in CLASS_S) happened in 100.00% of the analyzed instances. This reinforces the core modifications for this refactoring. However, the target method creation did not happen in 20.37% of the instances.

Similarly to the *Move Method* refactoring, the method was not created, since it already existed, but is now at the parent's class as an abstract method. This way, we observe that, to apply the *Pull Up Method* refactoring, there might be no need to create a new method, once it might exist as an abstract method at a superclass.

Table 4.10: Modification Spread per Location and per Type (Source, Target)

	SUR				EXT				CLASS							
	S		T		S		T		S		T		INSOURCE		INTARGET	
	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-
Extract Method	5.84%	12.73%	42.06%	0.93%	11.92%	14.84%	25.93%	1.99%	1.52%	0.47%	98.48%	0.93%	100.00%	100.00%	100.00%	1.40%
Inline Method	3.45%	28.16%	13.22%	6.32%	5.17%	14.37%	14.94%	9.77%	3.45%	94.25%	0.57%	2.30%	5.17%	100.00%	98.85%	99.43%
Move Method	0.00%	35.90%	26.92%	0.00%	0.00%	58.97%	75.64%	0.00%	0,00%	97.44%	93.59%	3.85%	1.28%	98.72%	98.72%	5.13%
Pull Up Method	0.00%	0.00%	5.56%	0.00%	0.00%	68.52%	42.59%	0.00%	0.00%	100.00%	79.63%	0.00%	0.00%	100.00%	100.00%	20.37%

Finding 5: The creation of the method in the parent's class did not occur in 20.37% of the Pull Up instances.

In addition, we observed that in three (5.50%) instances of the *Pull Up Method* refactoring, some class has implemented the pulled up method. The method has been removed from some classes in 25 (46.20%) instances. The removal happened because the method is duplicated in distinct classes of the same hierarchy. This removal is a core modification presented in Table 4.2. Once the method has been pulled up, there was no need to have such method in both subclasses. This way, the method has been removed.

Modification Spread. Until now, we have discussed the five most frequent modifications for each refactoring analyzed in this paper. We then focused on where the modifications frequently occur. Table 4.10 illustrates, for each refactoring, the percentage of instances that had at least one change in the respective type (addition and removal) and location (SUR, EXT, CLASS, INSOURCE, INTARGET). Besides, this table also describes whether the location is related to the source (S) or target (T). We observed that there is a considerable percentage of SUR and EXT modifications in some of the types. This shows that the modification goes beyond the source and target methods, affecting a larger scope of the project, especially for *Move Method* and *Pull Up Method* refactorings. For these, the amount of changes that occurred in the EXT group is much greater than the ones in SUR group. This difference in quantity informs us that these changes have a greater impact on other classes of the project. Therefore, we can see that the developers applied changes around the source and target in (i) 68.81% for *Extract Method*, (ii) 48.85% for *Inline Method*, 85.06% for *Move Method* and 26.44% for *Pull Up Method*, of the analyzed refactoring instances.

Table 4.11: Method Invocation on Refactoring Clients

Refactoring	source call deletions	target call additions	source to target swap
Extract Method	9.35%	49.65%	11.45%
Inline Method	30.46%	12.64%	6.32%
Move Method	41.02%	57.69%	44.87%
Pull Up Method	22.22%	27.77%	9.25%

The modifications that are not located in the INSOURCE or INTARGET, occurred in clients of the refactoring. The most frequent modification that occurred in the clients of all refactorings was Method Invocation. Since the source and target methods are modified by refactorings, their respective clients need to be adjusted accordingly. For example, consider a client of the *Extract Method* source method. This client may choose to (i) stop calling the source method, (ii) switch to the target method as well, (iii) exchange the call to the source method with the call to the target method. Table 4.11 shows, for each refactoring, the number of clients that performed each of these three options.

For *Extract Method*, we can notice that the addition to the target method occurred in 49.65% of the time. This suggests that the method extraction is useful to avoid duplicate the code by adding a new call to the target in the client. For *Inline Method*, we can already see that the highest concentration of method invocations is related to the deletion of the call to the source method. Once the source method ceases to exist, it is necessary to remove its call in the client methods. Nevertheless, the low inclusion of the call to the target or the exchange of calls from the source to the target, implies that (i) the client methods no longer perform the functionality of the source method, (ii) added functionality in their own body, it can even be duplicated. For *Move Method*, we have a large number of the three situations, but in particular, for the addition of the so-called target. Finally, for *Pull Up Method*, we have a similar occurrence between removing calls to the source method and adding calls to the target.

Summary: The core modifications of each analyzed refactoring occurred frequently during the application of the respective refactoring. However, most of the refactoring instances involved additional modifications. *Pull Up Method* presented the expected core modifications in 79.63% of the instances. We also observed a high number of additional recurring modifications to either support floss refactoring or adjusting the refactoring to specific structures (*e.g.*, move a method across hierarchies). Finally, we also observed that the modifications are not limited to occur in the source and target methods, indicating that the refactoring affects the code in a larger scope than the one described by Fowler's catalogue (1) and supported by existing refactoring tools (81).

4.4.2 RQ2: How often developers apply customized refactorings?

On RQ_1, we focused on revealing what are the refactoring modifications in isolation. We discussed the types of modifications that occurred in each

refactoring type, in addition to the locations where each modification occurred. In particular, we highlighted to what extent core modifications are indeed frequent and the intriguing cases of modifications that are not part of such a core. However, in order to understand which modifications developers perform together to compose a single refactoring, we decided to investigate common modification patterns. A pattern is a set of modifications that occurred together in multiple instances of the same refactoring. Thus, the most frequent patterns are possible candidates to be considered as a customized refactoring.

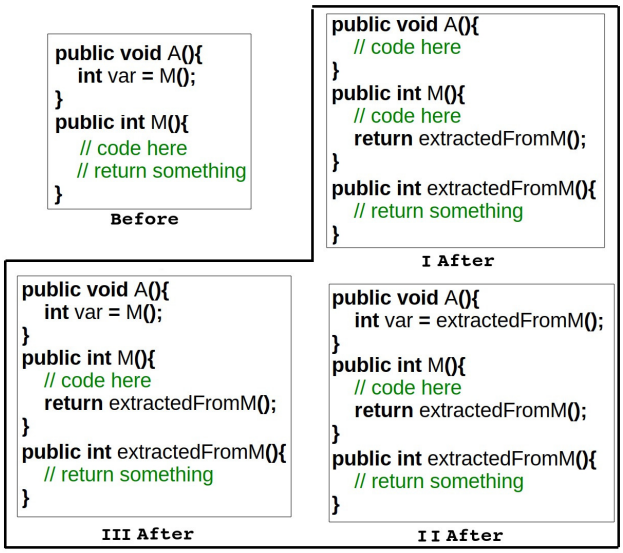


Figure 4.5: Different Patterns for *Extract Method*

Understanding Patterns. Figure 4.5 shows three possible ways to perform an *Extract Method* for the same code snippet. These possible ways, enumerated as I, II, and III, represent the frequent modification patterns found at the analyzed instances of *Extract Method*. In all three patterns, we can observe all the core modifications for *Extract Method*. The method EXTRACTEDFROMM was extracted from method M. Moreover, it was added a call for the method EXTRACTEDFROMM within method M. Thus, these patterns should have at least the modifications of (i) addition of Method Declaration located in CLASS_T, indicating the creation of the extracted method (target); (ii) addition of Method Access located in INSOURCE, indicating a call to the target method within the source method. However, for each pattern, other modifications were made in conjunction with those mentioned so that the extracted method could better suit the applied scenario. In the pattern I, we can also observe the modification of removal Method Access. In this context, this modification indicates the removal of the call to the source method within method A. Removing this call changes the behavior

of method A and may indicate a possible incomplete refactoring. As seen in Table 4.11, this removal occurred in 9.35% of the *Extract Method* instances.

Regarding Pattern II, in addition to removing the source method M, the target method call, EXTRACTEDFROMM(), was added. So, in this case, the new method was not just called within the source method. This case presents a situation in which the client was not interested in the entire source method M and therefore is using only the extracted part. In addition, this case may indicate an unexpected behavior change in method A.

Pattern III is a simpler case of applying the *Extract Method*, where a method is extracted from method M and that method is called in the method where it was extracted, that is, the source method of that refactoring. In this pattern, it is possible to observe that we do not represent in the examples a modification that explicitly characterizes the transfer of the code from one method to another. This type of modification is already implicit in the creation of a new method through Method Declaration.

Figures 4.6 to 4.9 present the most frequent patterns for each refactoring type. The percentage above each modification represents the percentage of instances that have the pattern composed of all modifications from the extreme left until the modification indicated with the percentage. For instance, the pattern composed of the modifications Method Declaration (target) and Method Access (target) in Figure 4.6 occurred in 60.86% of *Extract Method* instances. The text source/target below each modification indicates whether the modification interacts with the source method (S) or the target method (T). In this analysis, we considered only modifications that occurred in client methods, *i.e.*, we disregarded the modifications that occurred inside the source and target methods. Finally, the green and red colors indicate whether the modification is an addition modification (+) or removal modification (-), respectively. For instance, let us consider the same pattern composed of the modifications Method Declaration (target) and Method Access (target) in Figure 4.6. This pattern will be expressed as {Method Declaration.T+, Method Access.T+}.

4.4.2.1 Frequent Customization Patterns

Figure 4.6 presents the most frequent patterns for *Extract Method*. We observed that the modification Method Declaration occurred in almost all *Extract Method* instances (98.48%). The pattern with the two most frequent modifications is {Method Declaration.T+, Method Access.T+}. This pattern indicates that the client methods usually (60.86%) add a call to the target

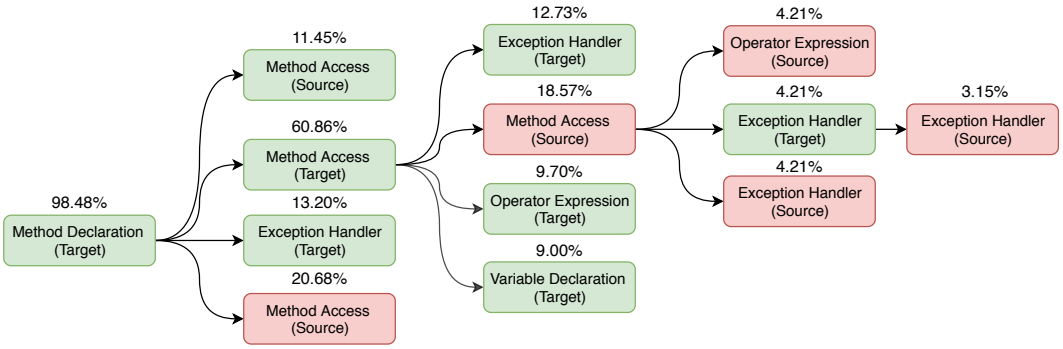


Figure 4.6: Most Common Patterns for *Extract Method*

method after the extraction. Meanwhile, only in 11.45% of the instances, the clients added a call to the source method after the extraction ({Method Declaration.T+, Method Access.S+}). This percentage is less than the percentage of clients that removed the call to the source method after the extraction (20.68%). Furthermore, we observed in the pattern {Method Declaration.T+, Method Access.T+, Method Access.S-} that almost all instances that have client methods removing a call to source method also have client methods adding a call to the target Method. However, only in 11.45% of these cases, the swap of call occurred in the same client method, as observed in Table 4.11.

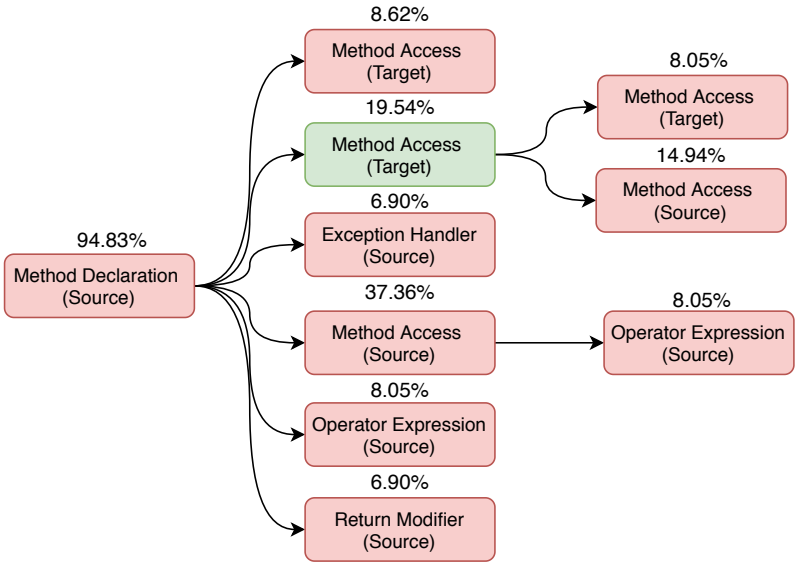


Figure 4.7: Most Common Patterns for *Inline Method*

Figure 4.7 presents the most frequent patterns for *Inline Method*. We observed that the deletion of the source method occurred in 94.83% of the *Inline Method* instances. Indeed, a previous study (41) showed that developers sometimes prefer to keep the source method when applying the refactoring *Inline Method*. Keeping the source method after the refactoring differs from what is proposed as core modifications in Table 4.2. We also

observed in Figure 4.7 that most of the modifications are of the removal type and interact with the source method. This indicates that the clients of the source method needed to be adjusted to remove the interactions that they have with the source method. Besides, in 37.36% of the *Inline Method* instances, the pattern {Method Declaration.S-, Method Access.S-} occurred. This pattern indicates that the client methods removed a call to the source method. However, according to Table 4.11, only in 6.32% of the instances, the client methods also added a call to the target method along with the removal of the call to the source method. In this way, client methods that removed the call to the source method and did not replace the removed call to a call to the target method had their functionality reduced. Finally, we can also observe that in 19.54% of the instances the {Method Declaration.S-, Method Access.T+} pattern occurred, that is, client methods added a call to the target method after the refactoring.

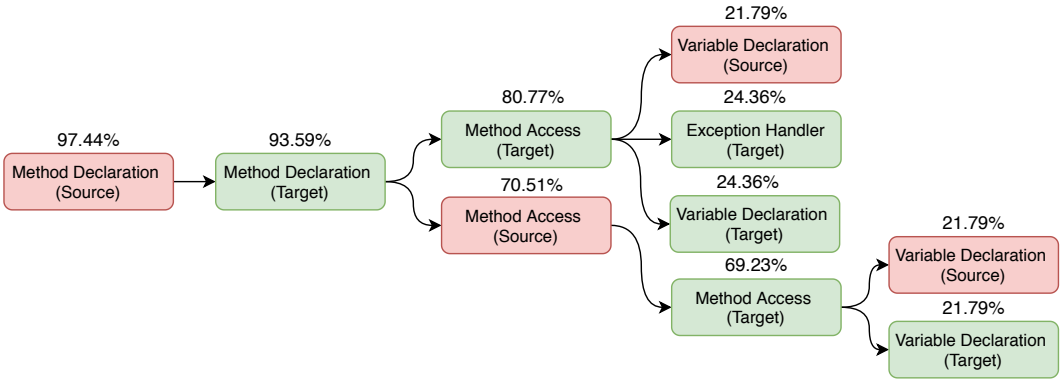


Figure 4.8: Most Common Patterns for *Move Method*

Figure 4.8 presents the most frequent patterns for *Move Method*. We observed that in 93.59% of the instances the method was removed from one class and added in another class, according to the pattern {Method Declaration.S-, Method Declaration.T+}. In addition to this pattern, we observed that more complex patterns tend to add calls to the target method (80.77%) and remove calls to the source method (70.51%), according to the patterns ({Method Declaration.S -, Method Declaration.T+, Method Access.T+}) and ({Method Declaration.S-, Method Declaration.T+, Method Access.S-}), respectively. However, according to the Table 4.11, only in 44.87% of the instances of *Move Method*, there are clients who switched the call from the source method to the target method. This indicates that developers more frequently add new clients to the moved method when performing the *Move Method*.

Figure 4.9 presents the most frequent patterns for *Pull Up Method*. We observed that in all *Pull Up Method* instances the source method was

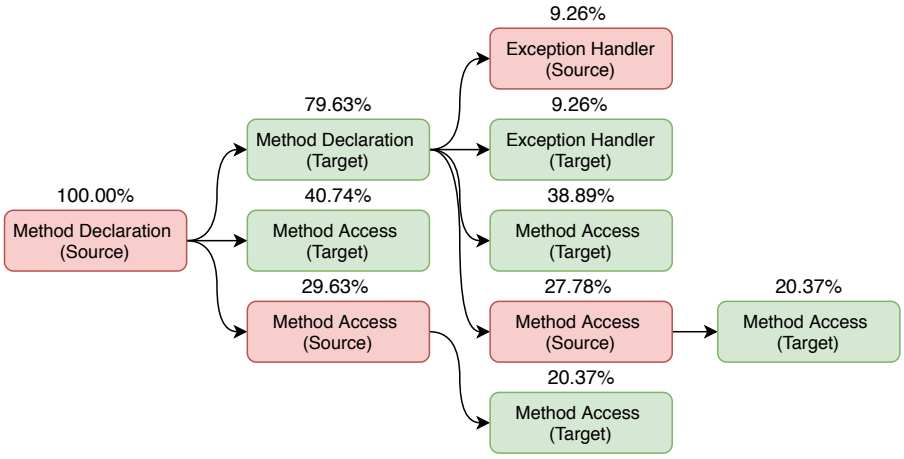


Figure 4.9: Most Common Patterns for *Pull Up Method*

removed. However, the removal of the source method together with the addition of the target method occurred in 79.63% of the *Pull Up Method* instances, represented by the pattern {Method Declaration.S-, Method Declaration.T+}. As discussed earlier, the mother class in the hierarchy may already have a method with the same signature, or an abstraction, than the method pulled up, before the refactoring. Besides, similar to the *Move Method*, we observed that there are a considerable number of clients that added a call to the target method (38.89%) or that removed a call to the source method (27.78%), in addition to the pattern {Method Declaration.S-, Method Declaration.T+}. However, different from *Move Method*, the percentage of instances that have clients that made both modifications is only 9.25%, according to the table 4.11.

> **Finding 6:** The current Fowler's catalog of modifications are not directly in line with the Figures 4.6 to Figures 4.9. One might reconsider to extend Fowler's catalogue to properly document customized refactorings.

4.4.2.2 Automated refactoring tools

Until now, we identified the most frequent modification patterns applied in practice by developers when performing each of the refactoring types. Taking these patterns into account, we investigated if the automated refactoring tool provided by Eclipse properly supports the application of these patterns. We chose Eclipse because is a very popular development tool for Java. Besides, Eclipse is frequently used in literature, *e.g.* (40, 41). We observed the source code associated with each pattern described in Figures 4.6 to 4.9. We minimally adapted the code to be reproducible in our Eclipse's environment. Then, we

manually invoked the Eclipse's refactoring tool in order to reproduce the refactoring applied by the developer in its software project. Finally, we listed which patterns could not be reproduced using the Eclipse's refactoring tool. In this reproduction, we used Eclipse Version: 2019-12 (4.14.0).

Table 4.12: List of Eclipse's Refactoring Automated Tool Limitations.

Id	Limitation
1	Modification only supported if occurred in source/target methods
2	It is not possible to remove source method invocation in client methods
3	It is not possible to remove target method invocation in client methods
4	It is not possible to add source method invocation in client methods
5	It is not possible to add target method invocation in client methods
6	There is no exception support for methods different than source and target ones
7	No exception handler is added if there is an exception error before the refactoring application
8	It is not possible to manage who should handle the exception
9	It is necessary that the extracted code is duplicated and the duplication recognized by the Eclipse *-Exclusive for Extract Method*
10	It is not possible to remove the modification without replacing it with the inlined method body *-Exclusive for Inline Method*
11	The swap of the call from source to target must occur in the same client *-Exclusive for Pull Up Method and Move Method*
12	It is mandatory to create the moved method, even if there is already a method with the same name in the destination class *-Exclusive for Move Method*

Tables 4.13 to 4.16 present the results of our reproduction. The first column indicates the reproduced pattern. The second column indicates the support provided by Eclipse's refactoring tool to the application of the respective pattern. We classified the support into three categories. The first category is named Full support. This category means that the refactoring tool is able to reproduce the pattern completely for all reproduced scenarios. The second category is named Partial support. This category means that the refactoring tool is able to reproduce the pattern completely only in specific scenarios. Finally, the last category is named No support. As the name says, the refactoring tool is not able to reproduce the complete pattern. The last column indicates the id of the limitation. This id is associated with the Table 4.12. Where each row of Table 4.12 describes the limitation id followed by its description.

We observed that most of the limitations, identified by id 1 to 8 in Table 4.12, are limitations common to more than on refactoring type. We also observed that the refactoring tool focuses on providing support to the developer when they are performing modifications in the target and source methods. Thus, the support provided by the refactoring tool for modifications made in client methods is very limited or null, though the occurrence of modifications in these methods are frequent, as shown and discussed in the Table 4.10.

Table 4.13: Limitations in Eclipse's Automated *Extract Method* Tool.

Patterns	Eclipse	Limitation Id
Method Declaration.T+	Full support	
Method Declaration.T+,Method Access.S+	No support	4
Method Declaration.T+,Method Access.T+	Partial support	9
Exception Handler.T+,Method Declaration.T+	Partial support	6,7,8
Method Declaration.T+,Method Access.S-	No support	2
Exception Handler.T+,Method Declaration.T+, Method Access.T+	No support	6,7,8,9
Method Declaration.T+,Method Access.S-, Method Access.T+	No support	2,9
Method Declaration.T+,Method Access.T+, Operator expression.T+	Partial support	1(Operator expression),9
Method Declaration.T+,Method Access.T+, Variable Declaration.T+	No support	1(Variable Declaration),9
Method Declaration.T+,Method Access.S-, Method Access.T+,Operator Expression.S-	No support	1(Operator expression),2,9
Exception Handler.T+,Method Declaration.T+, Method Access.S-,Method Access.T+	No support	2,6,7,8,9
Exception Handler.S-,Method Declaration.T+, Method Access.S-,Method Access.T+	No support	2,6,7,8,9
Exception Handler.S-,Exception Handler.T+, Method Declaration.T+,Method Access.S-, Method Access.T+	No support	2,6,7,8,9

Table 4.14: Limitations in Eclipse's Automated *Inline Method* Tool.

Patterns	Eclipse	Limitation Id
Method Declaration.S-	Full support	
Method Declaration.S-,Method Access.S-	Partial support	10(Method Access)
Method Declaration.S-,Method Access.T+	No support	5
Method Declaration.S-,Method Access.S-, Method Access.T+	No support	5,10
Method Declaration.S-,Method Access.T-	No support	3
Method Declaration.S-,Method Access.S-, Operator expression.S-	Partial support	10(Operator expression), 10(Method Access)
Method Declaration.S-,Method Access.T+, Method Access.T-	No support	3,5
Method Declaration.S-,Operator expression.S-	Partial support	10(Operator expression)
Exception Handler.S-,Method Declaration.S-	Partial support	6,7,8
Method Declaration.S-,Return modifier.S-	Partial support	10(Return modifier)

Table 4.13 presents the found tool limitations when applying *Extract Method* refactoring. Except for the pattern Method Declaration.T+, all the other patterns have no or partial support. Limitation 2 is the most frequent

among the patterns, once most of the patterns include the removal of a Method Access in a client method. Limitations 6, 7, and 8 affect the modification Exception Handler. For instance, if the selected statements for *Extract Method* throws an exception, the target method will throw this exception, even if the exception thrower is completely extracted. Thus, Eclipse does not allow developers to define who (source/target/clients method) must handle that exception. This inflexibility force all the methods that invoke the target one to handle themselves the exception.

Eclipse's refactoring tool maintains the same limitations, as discussed for *Extract Method*, for the remaining refactoring types. In this way, this tool only provides full support for patters composed of core modifications. However, there are some particularities for each refactoring type. For *Inline Method*, we have the limitation 10. In this limitation, developers can choose to exchange the call to the source method to the source's body. However, the refactoring tool does not let the developer only remove the call to the source method or swap the call to the source method to a call to the target method, as occurred in 6.32% of Inline Method instances (Table 4.11). For *Move Method* and *Pull Up*, we have the limitation 11. This limitation indicated that the refactoring tool allows the exchange of the call to the source method for the call to the target, but not allows only the addition of the call to the target Method or only removal of the call to the source method. This limitation affects especially the *Move Method*, because the only addition of a call to the target method occurred in 57.69% of *Move Method* instances (Table 4.11). Finally, the limitation 12 is exclusive for *Move Method*. This limitation indicates that is not possible to move only the method content to a method with the same signature in the destination class. In this way, developers are forced to: (i) apply the *Move Method* manually, or (ii) force the method to be moved, leaving the destination class with two methods with the same signature.

Finding 7: Eclipse's automated refactoring tool is not able to properly support customized refactoring.

Summary: We observed that exist different ways to apply the same refactoring depending on particular program characteristics. Besides, we observed in Figures 4.6 to Figure 4.9 that the most frequent patterns usually include the core modifications and invocations of both the target and source methods in different classes and methods. We also observed that handle exceptions are a common task during the refactoring application. Finally, we could observe that Eclipse's refactoring tool is not able to provide proper support for

Table 4.15: Limitations in Eclipse's Automated *Move Method* Tool.

Patterns	Eclipse	Limitation Id
Method Declaration.S-	No support	12
Method Declaration.T+	Full support	
Method Declaration.S-,Method Declaration.T+	Full support	
Method Declaration.S-,Method Declaration.T+, Method Access.T+	No support	5
Method Declaration.S-,Method Declaration.T+, Method Access.S-	No support	2
Method Declaration.S-,Method Declaration.T+, Method Access.S-,Method Access.T+	Partial support	11
Method Declaration.S-,Method Declaration.T+, Method Access.T+,Variable declaration.T+	No support	Variable Declaration(1),5
Exception Handler.T+,Method Declaration.S-, Method Declaration.T+,Method Access.T+	No support	5,6,7,8
Method Declaration.S-,Method Declaration.T+, Method Access.S-,Method Access.T+, Variable declaration.S-	No support	Variable Declaration(1),11
Method Declaration.S-,Method Declaration.T+, Method Access.S-,Method Access.T+, Variable declaration.T+	No support	Variable Declaration(1),11

Table 4.16: Limitations in Eclipse's Automated *Pull Up Method* Tool.

Patterns	Eclipse	Limitation Id
Method Declaration.S-	Full support	
Method Declaration.S-,Method Declaration.T+	Full support	
Method Declaration.S-,Method Declaration.T+, Method Access.T+	No support	5
Method Declaration.S-,Method Declaration.T+, Method Access.S-	No support	2
Method Declaration.S-,Method Declaration.T+, Method Access.S-,Method Access.T+	Partial support	11
Method Declaration.S-,Method Access.T+	No support	5
Method Declaration.S-,Method Access.S-	No support	2
Method Declaration.S-,Method Access.S-, Method Access.T+	Partial support	11
Exception Handler.S-,Method Declaration.S-	Partial support	6,7,8
Exception Handler.T+,Method Declaration.S-, Method Declaration.T+	Partial support	6,7,8

the frequent patterns of each of the refactoring types presented in Tables 4.16 to 4.13. In this way, we listed in Table 4.12 the limitations found when we

reproduced the refactorings applied by developers in practice.

4.4.3 RQ3: Does customized refactoring reduce the intensity of code smells?

On RQ_3, we focused on understanding the impact that different customizations have on code smells. For this purpose, we analyzed all the most common patterns identified in Section 4.4.2. We analyzed the impact that each pattern has on code smells by extracting the following information about the instances of these patterns: (i) which code smell types were *introduced*; (ii) which code smell types were *removed*, and; (iii) which code smell types were *mitigated*, but not outright removed. This last criteria determines a code smell type as *mitigated* if at least one of its metrics has improved, without worsening any of its other metrics. We chose metrics popularly used in literature (*e.g.*, (2, 80, 89, 93)) for each smell type. Altogether, we are able to detect and evaluate 17 different types of code smell as mentioned in Section 4.3.

Tables 4.17 to 4.20 present the impact of the patterns on smells for each refactoring. The first column presents the pattern. The followed columns present the smells introduced, removed and mitigated, respectively. We focused on present only the three smells with the highest percentages for each column that occurred at least in two instances of the pattern.

We can observe in Table 4.17 that the most frequently introduced smells are Feature Envy, Complex Class, Long Method, and Long Parameter List. Complex patterns, *i.e*, with a higher number of modifications, tend to introduce Long Parameter List more frequently, overcoming the introduction of the Feature Envy. Thus, in these patterns, the target method tends to have more parameters, though it is expected that it would have a simpler signature due to the reduced size. We can also observe that the pattern {Method Declaration.T+, Method Access.S-, Method Access.T+, Operator Expression.S-} was the only pattern that introduced Class Data Should be Private (CSDP) and Speculative Generality. In addition, this same pattern does not tend to introduce complex smells, such as Feature Envy or Complex Class. On the other hand, this same pattern does not tend to remove any smell type.

We also observed that simpler patterns tend to frequently remove a higher number of smells than more complex patterns. Although the Complex Class smell is not removed, it is often mitigated. Besides the Complex Class, other smell types that are inserted, such as Feature Envy and Long Method, are also mitigated. Manual analysis indicated that this happens because the extracted code tends to be a complex code fragment. In this way, the source method

Table 4.17: Impact of the Extract Method Patterns on Smells.

Pattern	Smell Introduced	Smell Removed	Smell Mitigated
Method Declaration.T+	FeatureEnvy: 18.0% ComplexClass: 17.0% LongMethod: 12.5%	FeatureEnvy: 6.3% LongMethod: 5.5% IntensiveCoupling: 3.7%	LongMethod: 6.3% ComplexClass: 6.3% FeatureEnvy: 4.3%,
Method Declaration.T+,Method Access.S+	FeatureEnvy: 19.4% LongMethod: 15.3% ComplexClass: 13.3%	IntensiveCoupling: 5.1% LongMethod: 5.1% FeatureEnvy: 4.1%	ComplexClass: 4.1%
Method Declaration.T+,Method Access.T+	FeatureEnvy: 18.8% ComplexClass: 15.0% LongParameterList: 12.1%	FeatureEnvy: 7.1% LongMethod: 4.4% IntensiveCoupling: 3.3%	ComplexClass: 6.5% LongMethod: 4.0% FeatureEnvy: 3.8%
Exception Handler.T+,Method Declaration.T+	ComplexClass: 20.4% FeatureEnvy: 19.5% LongMethod: 15.0%	LongMethod: 6.2% IntensiveCoupling: 3.5% FeatureEnvy: 3.5%	ComplexClass: 10.6% FeatureEnvy: 8.0% LongMethod: 7.1%
Method Declaration.T+,Method Access.S-	FeatureEnvy: 18.6% LongParameterList: 18.1% ComplexClass: 14.1%	FeatureEnvy: 7.9% LongMethod: 4.0% IntensiveCoupling: 3.4%	ComplexClass: 4.0%
Exception Handler.T+,Method Declaration.T+, Method Access.T+	ComplexClass: 21.1% FeatureEnvy: 19.3% LongMethod: 13.8%	LongMethod: 5.5% IntensiveCoupling: 3.7% FeatureEnvy: 3.7%	ComplexClass: 11.0% FeatureEnvy: 7.3% LongMethod: 6.4%
Method Declaration.T+,Method Access.S-, Method Access.T+	FeatureEnvy: 18.2% LongParameterList: 17.6% ComplexClass: 13.2%	FeatureEnvy: 8.8% LongMethod: 4.4% IntensiveCoupling: 3.8%	ComplexClass: 3.8%
Method Declaration.T+,Method Access.T+, Operator expression.T+	LongParameterList: 13.3% ComplexClass: 12.0%	FeatureEnvy: 3.6%	ComplexClass: 9.6% FeatureEnvy: 8.4% IntensiveCoupling: 3.6% LongMethod: 3.6%
Method Declaration.T+,Method Access.T+, Variable declaration.T+	LongParameterList: 16.9% FeatureEnvy: 15.6% ComplexClass: 13.0%	FeatureEnvy: 10.4% LongMethod: 5.2%	ComplexClass: 3.9% FeatureEnvy: 3.9%
Method Declaration.T+,Method Access.S-, Method Access.T+,Operator expression.S-	LongParameterList: 22.2% CDSP: 11.1% SpeculativeGenerality: 5.6%		ComplexClass: 5.6%
Exception Handler.T+,Method Declaration.T+, Method Access.S-,Method Access.T+	ComplexClass: 20.0% LongParameterList: 20.0% FeatureEnvy: 17.1%	LongMethod: 8.6% IntensiveCoupling: 5.7%	
Exception Handler.S-,Method Declaration.T+, Method Access.S-,Method Access.T+	LongParameterList: 22.2% FeatureEnvy: 22.2% ComplexClass: 19.4%	LongMethod: 8.3% DispersedCoupling: 5.6%	
Exception Handler.S-,Exception Handler.T+, Method Declaration.T+,Method Access.S-, Method Access.T+	ComplexClass: 25.9% LongParameterList: 25.9% FeatureEnvy: 18.5%	LongMethod: 7.4%	

that previously had a smell, *e.g.* Complex Class, had the metrics improved, so the smell is mitigated. However, the extracted method can be complex enough to be considered a new smell. Finally, similar to smell removal, the simplest patterns tend to mitigate a greater number of smells frequently if compared with more complex patterns.

Table 4.18 presents the relation between the most frequent patterns for *Inline Method* refactoring and code smells. We observed that patterns with only the deletion of the source method (Method Declaration.S-) and some modification of method invocation (Method Access) tend to remove more types of smells. In particular, these patterns tend to remove Feature Envy, reaching up to 19.2% of instances, for the pattern {Method Declaration.S-, Method Access.S-, Method Access.T+}. On the other hand, patterns with the modification Operator Expression did not remove smells. To make worse,

these patterns frequently inserted 5 different types of smells. This is a high amount compared to an average of 3 smells types introduced for the other patterns. Regarding the pattern {Method Declaration.S-, Return Modifier.S-}, we observed a frequent removal of the smell Long Parameter List. This removal is not common for the other patterns, once they had Feature Envy as the most frequently removed smell.

Finding 8: There are patterns that frequently introduced or removed more a certain type of smell.

Finally, we can observe that the smell Complex Class is the more frequently introduced smell. Besides, Feature Envy and Long Method are also commonly introduced. Although *Inline Method* is the opposite of the *Extract Method*, both had similar results to the types of smells introduced. Another important observation is the recurrent mitigation of complex smells that affects classes, such as Complex Class, Spaghetti Code, and God Class. Regarding the smell Spaghetti Code, we observed that this smell was frequently introduced (25%) only for the pattern {Exception Handler.S-, Method Declaration.S-}, being this pattern the only one with the modification Exception Handler.

Table 4.19 presents the code smells introduced, removed and mitigated when performing different patterns for *Move Method*. Similar to the refactoring *Inline Method*, the most common smells mitigated when applying a *Move Method* were Complex Class, God Class, and Spaghetti Code. Another important observation is that more complex patterns removed and mitigated a less variety of smell types. Besides, unlike the *Inline Method*, we also have the frequent introduction of the smell Speculative Generality. It is important to note that this smell is repeatedly introduced by patterns with at least 3 modifications, where there are modifications of the type Method Access. This modification type indicates a call to the source (Method Access.S) or target (Method Access.T) method. Another difference is the constant insertion of the smell CDSP, present in almost all the frequent patterns of the Move Method. Finally, we observed that the insertion of the smell Speculative Generality appeared in customizations, becoming even more frequent for more complex patterns. This same smell did not appear for patterns that have only core modifications.

Finding 9: More complex patterns tend to remove and mitigate fewer different smell types.

Table 4.18: Impact of the Inline Method Patterns on Smells.

Pattern	Smell Introduced	Smell Removed	Smell Mitigated
Method Declaration.S-	ComplexClass: 20.6% LongMethod: 12.7% FeatureEnvy: 12.1%,	FeatureEnvy: 9.7% DispersedCoupling: 5.5% LongMethod: 5.5%	ComplexClass: 21.8% SpaghettiCode: 13.3% GodClass: 12.7%
Method Declaration.S-,Method Access.S-	ComplexClass: 24.6% FeatureEnvy: 13.8% LongMethod: 13.8%	FeatureEnvy: 9.2% LongParameterList: 4.6% LongMethod: 4.6%	ComplexClass: 20.0% SpaghettiCode: 12.3% GodClass: 10.8%
Method Declaration.S-,Method Access.T+	ComplexClass: 20.6% FeatureEnvy: 11.8% LongMethod: 11.8%	FeatureEnvy: 14.7% IntensiveCoupling: 5.9% LongMethod: 5.9%	ComplexClass: 23.5% GodClass: 8.8% CDSP: 8.8% SpaghettiCode: 8.8%
Method Declaration.S-,Method Access.S-, Method Access.T+	ComplexClass: 15.4% FeatureEnvy: 15.4% CDSP: 11.5% LongMethod: 11.5%	FeatureEnvy: 19.2% IntensiveCoupling: 7.7% LongMethod: 7.7%	ComplexClass: 26.9% GodClass: 11.5% CDSP: 11.5% SpaghettiCode: 11.5%
Method Declaration.S-,Method Access.T-	ComplexClass: 26.7% LongMethod: 13.3%	FeatureEnvy: 13.3%	ComplexClass: 13.3%
Method Declaration.S-,Method Access.S-, Operator expression.S-	ComplexClass: 21.4% FeatureEnvy: 21.4% CDSP: 14.3% LongMethod: 14.3% ShotgunSurgery: 14.3%		ComplexClass: 35.7% SpaghettiCode: 28.6%
Method Declaration.S-,Method Access.T+, Method Access.T-	ComplexClass: 28.6% LongMethod: 14.3%	FeatureEnvy: 14.3%	ComplexClass: 14.3%
Method Declaration.S-,Operator expression.S-	ComplexClass: 21.4% FeatureEnvy: 21.4% LongMethod: 14.3% CDSP: 14.3% ShotgunSurgery: 14.3%		ComplexClass: 35.7% SpaghettiCode: 28.6%
Exception Handler.S-,Method Declaration.S-	ComplexClass: 33.3% SpaghettiCode: 25.0% FeatureEnvy: 16.7% LongMethod: 16.7%	FeatureEnvy: 16.7%	ComplexClass: 25.0%
Method Declaration.S-,Return modifier.S-	ComplexClass: 33.3% CDSP: 16.7%	LongParameterList: 16.7%	ComplexClass: 33.3% SpaghettiCode: 33.3% GodClass: 25.0%

Finally, Table 4.20 presents the smells related with the most common modification patterns for *Pull Up Method*. We observed that simpler patterns with only core modifications introduced and removed 3 different types of smell, in contrast with more complex patterns that only introduced Complex Class. Also, customizations of the refactoring *Pull Up Method* did not remove smells. Only two customizations removed at least one smell, Feature Envy. We also observed that Complex Class, Spaghetti Code, and Data Class are frequently mitigated by almost all patterns for *Pull Up Method*.

Summary: We can see that, in general, patterns tend to lean towards having a negative effect on the code structure, introducing more smells than removing or mitigating them. Besides, similar smell types were introduced for different patterns of the same refactoring. This observation is also valid for the removal and mitigation of smells. However, for *Pull Up Method* we observed a significant difference in the types of smells introduced between: (i) simpler patterns, mostly having only core modifications, and (ii) more complex

Table 4.19: Impact of the Move Method Patterns on Smells.

Pattern	Smell Introduced	Smell Removed	Smell Mitigated
Method Declaration.S-	ComplexClass: 19.7% FeatureEnvy: 10.5% CDSP: 10.5%	ComplexClass: 3.9% DataClass: 3.9%	ComplexClass: 30.3% GodClass: 15.8% SpaghettiCode: 10.5%
Method Declaration.T+	ComplexClass: 21.3% CDSP: 10.7% FeatureEnvy: 10.7%	ComplexClass: 4.0% DataClass: 4.0%	ComplexClass: 29.3% GodClass: 14.7% SpaghettiCode: 10.7%
Method Declaration.S-,Method Declaration.T+	ComplexClass: 20.5% CDSP: 11.0% FeatureEnvy: 11.0%	ComplexClass: 4.1% DataClass: 4.1%	ComplexClass: 28.8% GodClass: 13.7% SpaghettiCode: 9.6%
Method Declaration.S-,Method Declaration.T+, Method Access.T+	ComplexClass: 20.6% CDSP: 12.7% SpeculativeGenerality: 11.1% GodClass: 11.1% FeatureEnvy: 11.1%	ComplexClass: 4.8% DataClass: 4.8% FeatureEnvy: 3.2%	ComplexClass: 30.2% GodClass: 14.3% SpaghettiCode: 9.5%
Method Declaration.S-,Method Declaration.T+, Method Access.S-	ComplexClass: 20.0% CDSP: 14.5% SpeculativeGenerality: 10.9% FeatureEnvy: 10.9%	ComplexClass: 5.5% DataClass: 5.5% FeatureEnvy: 3.6%	ComplexClass: 30.9% GodClass: 14.5% SpaghettiCode: 10.9%
Method Declaration.S-,Method Declaration.T+, Method Access.S-,Method Access.T+	ComplexClass: 20.4% CDSP: 14.8% SpeculativeGenerality: 11.1% FeatureEnvy: 11.1%	ComplexClass: 5.6% DataClass: 5.6% FeatureEnvy: 3.7%	ComplexClass: 31.5% GodClass: 14.8% SpaghettiCode: 11.1%
Method Declaration.S-,Method Declaration.T+, Method Access.T+,Variable declaration.T+	SpeculativeGenerality: 26.3% ComplexClass: 26.3% CDSP: 26.3%		ComplexClass: 21.1%
Exception Handler.T+,Method Declaration.S-, Method Declaration.T+,Method Access.T+	ComplexClass: 42.1% SpaghettiCode: 26.3% SpeculativeGenerality: 21.1% CDSP: 21.1% GodClass: 21.1%	ComplexClass: 10.5%	ComplexClass: 31.6% GodClass: 21.1% SpaghettiCode: 15.8%
Method Declaration.S-,Method Declaration.T+, Method Access.S-,Method Access.T+, Variable declaration.S-	SpeculativeGenerality: 17.6% FeatureEnvy: 11.8%		ComplexClass: 35.3%
Method Declaration.S-,Method Declaration.T+, Method Access.S-,Method Access.T+, Variable declaration.T+	SpeculativeGenerality: 29.4% ComplexClass: 29.4% CDSP: 29.4%		ComplexClass: 23.5%

patterns, in which there is more customization. Besides, we observed that there are specific customized patterns that frequently introduced or removed more than a certain type of smell. For instance, a representative example of this situation is the smell Spaghetti Code, which is frequently introduced by the pattern {Exception Handler.S-, Method Declaration.S-} of *Inline Method* refactoring. Besides, we also observed that, while more complex customized refactoring patterns specialized the refactoring to affect certain smell types, they also reduced some of the refactoring positive effects. It is not clear whether the developer is achieving good trade-off decisions in these cases. Moreover, more complex patterns tend to remove a fewer number of smell types. However, as discussed in RQ_2, this could be caused by a lack of support from tools that aim to support the refactoring process (*e.g.*, those embedded in IDEs), for the additional modifications present in these customized refactorings.

Table 4.20: Impact of the Pull Up Method Patterns on Smells.

Pattern	Smell Introduced	Smell Removed	Smell Mitigated
Method Declaration.S-	ComplexClass: 14.8% SpeculativeGenerality: 13.0% FeatureEnvy: 7.4%	FeatureEnvy: 16.7% DispersedCoupling: 5.6% LongMethod: 5.6%	ComplexClass: 35.2% SpaghettiCode: 16.7% DataClass: 11.1%
Method Declaration.S-,Method Declaration.T+	ComplexClass: 16.3% SpeculativeGenerality: 11.6% FeatureEnvy: 7.0%	FeatureEnvy: 16.3% MessageChain: 4.7% LongMethod: 4.7%	ComplexClass: 27.9% SpaghettiCode: 18.6% DataClass: 14.0%
Method Declaration.S-,Method Declaration.T+ Method Access.T+	ComplexClass: 28.6% SpeculativeGenerality: 9.5%		ComplexClass: 19.0% SpaghettiCode: 14.3% DataClass: 9.5%
Method Declaration.S-,Method Declaration.T+ Method Access.S-	ComplexClass: 20.0%	FeatureEnvy: 20.0%	ComplexClass: 33.3% SpaghettiCode: 26.7% DataClass: 13.3%
Method Declaration.S-,Method Declaration.T+ Method Access.S-,Method Access.T+	ComplexClass: 27.3%		ComplexClass: 27.3% SpaghettiCode: 18.2%
Method Access.S-,Method Access.T+	ComplexClass: 27.3%		ComplexClass: 27.3% SpaghettiCode: 18.2%
Method Declaration.S-,Method Access.S-	ComplexClass: 18.8%	FeatureEnvy: 18.8%	ComplexClass: 43.8% SpaghettiCode: 31.2% DataClass: 12.5%
Method Declaration.S-,Method Access.S- Method Access.T+	ComplexClass: 27.3%		ComplexClass: 27.3% SpaghettiCode: 18.2%
Exception Handler.S-,Method Declaration.S- Exception Handler.T+,Method Declaration.S- Method Declaration.T+	ComplexClass: 60.0%		

4.5 Threats to Validity

Internal Validity. Refactoring Miner (15, 38) may yield false positives and false negatives. It has an effectiveness of 87.2% for recall and 98% (81) for precision, which is the best effectiveness among refactoring detection tools. In addition, we manually inspected some instances identified by it during our analysis. We relied on code smell detection tool used by recent studies (2, 80). However,these strategies were validated by previous work (89) with a resulting precision and recall (90) of 72% and 81%, respectively.

Construct Validity. The Refactoring Miner detects 15 types of refactorings, but we are considering only four types of refactorings. Although these four refactorings may not fully embrace all forms of refactoring customizations, these four refactorings are amongst the most frequently used by developers in practice. Finally, these refactorings affect the program structure differently. For instance, *Extract Method* is an inter-class refactoring, affecting directly only one class. Different from *Extract Method*, *Move Method* and *Pull Up Method* affect more than one class, including changes affecting a class hierarchy.

The collected modification types may not consider all possible modification types. We used Eclipse's JDT library, once this library has a very low level of granularity. In this way, we can detect a large number of modifications. Besides, this library is commonly used to build automated refactoring tools for Eclipse.

External Validity. Regarding the generality of our findings, we performed an in-depth analysis of refactoring instances from 13 Java projects, which satisfy the criteria defined in Section 4.3.1. Our results might not necessarily hold to other projects involving other primary programming languages and/or from domains not covered by our dataset. Moreover, we focused our analysis on open-source software projects. The nature of rafactoring in closed-source software projects is not necessarily the same as refactoring in open-source software projetcts. However, popular open-source projects have a major concern with software modularity; their teams tend to continuously refactor the source code to promote better code comprehensibility and maintainability by experienced and novice developers.

4.6 Conclusion

In this paper, we presented a study to understand how developers customize refactorings in practice. In particular, we investigated which are the core and additional modifications that developers performed when refactoring and how often they applied customized refactorings. We performed our study in 1,162 refactoring instances of four refactoring types, namely (*Extract Method*, *Inline Method*, *Move Method* and *Pull Up Method*) from 13 software projects.

Our results revealed that the core modifications of each refactoring type occurred frequently accompanied by additional modifications. These additional modifications customize the refactorings for the specific developer scenario where the refactoring will be applied. Besides, these additional modifications may be located in different code locations, not necessarily close to the source and target methods. We also observed that the most frequent patterns included the addition and removal of both target and source method invocations in different methods.

Unfortunately, the Eclipse's automated refactoring tool is not able to provide adequate support for automating these invocations. To make it worse, further analyses of the modification patterns revealed that the occurrence of those patterns tend to introduce some specific types of code smells. We also found an interesting relationship between the complexity of the pattern (*i.e.*, the number of additional modifications) and the variety of code smells introduced, as more complex patterns were shown to introduce fewer types of code smells, especially for *Pull Up Method*. Moreover, more complex customized refactoring patterns reduced the positive effects of the refactoring.

Finally, it is important to highlight that the current Fowler's catalog of modifications for each refactoring type should be revisited. The core modifi-

cations presented in Table 4.2 are not directly in line with the patterns found in Figures 4.6 to 4.9. It is also important to consider the fact that the lack of support for refactoring customization in existing tools might be one of the root causes for customized refactorings to be performed poorly.

Thus, as future work, we plan to design and implement tool support for better assisting developers on performing customized refactorings. The support should be enable flexible definition of customized refactoring patterns, while helping developers to avoid reusing patterns that might worsen their code structure quality. Our tool support is going to be integrated in the Eclipse IDE. We also plan to perform further studies, either quantitative or qualitative, that consider other refactoring types and other software projects, including close-sourced software projects.

4.7 Summary of Chapter 4

In this chapter, we investigated how developers apply four of the most frequent refactoring types in practice. These refactoring types are namely *Extract Method*, *Inline Method*,*Pull Up Method*, and *Move Method*. We observed that developers often customize refactorings. We also observed that the most frequent modification patterns include the addition and removal of both target and source method invocations across different methods. However, Eclipse's refactoring tool is not able to provide adequate support for automating these invocation-related modifications.

We also found a relationship between the complexity of the pattern (*i.e.*, the number of additional modifications) and the variety of code smells introduced. The more complex the modification pattern is, the fewer types of code smells are introduced. However, more complex customized refactoring patterns also reduced the positive effects of the refactoring. This observation indicates the need to come up with tools that assist developers in avoiding choices related to harmful customized refactorings. In fact, the problem could be indeed caused by a lack of support from tools that aim to support the refactoring application (*e.g.*, those embedded in IDEs), which does not guide developers in performing additional modifications present in typical customized refactorings.

5
Support for Refactoring Customization

As seen in Chapter 4, developers often needed to customize refactorings to be able to achieve their goals. We also observed that developers tend to use the same core and additional modifications to customized their refactorings. We were able to come up with a catalog of typical modification patterns used for creating customized refactorings. However, almost all frequent customizations are not completely supported by the refactoring tool provided by Eclipse. This tool's features do not allow developers to properly create their own custom refactoring. Developers are instead restricted to tailor refactorings only through basic configurations, which often do not satisfy their needs. For instance, let us consider the refactoring *Extract Method*. Eclipse's automated refactoring tool allows developers only to change the method signature, such as the access modifier and exception throws as possible customization. To make it worse, these configurations are specified by each refactoring type.

In order to address this problem, we proposed a flexible approach for enhancing refactoring automated support with customization features. The approach allows a developer to: (i) compose a customized refactoring according to his context's needs, and (ii) reuse the custom refactorings in similar contexts. We also discuss potential benefits of our proposal and elaborate on some implementation issues. This approach is described in a paper previously published. Thus, this chapter presents this paper, which is entitled *"On the Customization of Batch Refactoring"* (45) and published in the Proceeding of the 3th International Workshop on Refactoring, co-located with the International Conference on Software Engineering, held in Montreal, Canada, in May 2019.

On the Customization of Batch Refactoring

Daniel Tenorio
Pontifical Catholic University of Rio de Janeiro - Brazil
Email: doliveira@inf.puc-rio.br

Ana Carla Bibiano
Pontifical Catholic University of Rio de Janeiro - Brazil
Email: abibiano@inf.puc-rio.br

Alessandro Garcia
Pontifical Catholic University of Rio de Janeiro - Brazil
Email: afgarcia@inf.puc-rio.br

5.1 Introduction

Refactorings are program transformations that aim to improve the code structure, thereby making programs easier to understand and maintain (1). However, refactoring is a complex software maintenance practice which requires specialized effort for its application. In fact, developers often apply refactoring's program transformations in a *batch*, *i.e.*, various transformations in a sequence on a certain program location in order to achieve a specific goal (8, 94). Given the complexity of software refactoring, development companies may change their routine to adopt this practice (9).

Aiming to support developers' and companies' needs, IDEs, such as Eclipse[1] and IntelliJ[2], include tools for supporting automated refactoring. These tools have been widely explored in previous empirical studies, where they presented some advantage over the manual refactoring (17, 19). They make the refactoring process easier as well as reduce costs and failure proneness (40). Despite of these claimed benefits, developers are still reluctant to use automated refactoring along software development (8, 17, 40, 95).

The literature explores several limitations to understand the disuse of automated refactoring tools (9, 17). One of the key limitations is that existing tools provide limited support for refactoring customization (17, 96). IDE features do not allow developers to properly create their own custom refactoring. Developers are instead restricted to tailor transformations only through very basic configurations, which often do not satisfy their needs. The lack of customization also impairs the adaptation and reuse of predefined automated refactorings for different contexts. With all these downsides, developers often feel reluctant to use existing tools, and end up applying their refactorings manually (8, 94).

Developers consider that automated refactorings are a restrict practice where they have limited access. This limited access restricts the control of all modifications that will be applied by the refactoring, especially in the final

[1]https://www.eclipse.org/
[2]https://www.jetbrains.com/idea/

result (9, 17). Even worse, the code generated by automated refactoring tools for the same refactoring varies between IDEs (41). This divergence further discourages the use of these tools, once it reduces the predictability of the result. Because of this reduction, the results end up being different from the expected by the developers (41).

In this paper, we propose an approach for developers to customize refactoring. Our approach makes the automated refactoring more flexible. The additional flexibility allows the adaption to the context where the refactoring will be applied. For that, we split each refactoring into a set of minor program modifications that compose it. We call these minor modifications as *primitive modifications*. The primitive modifications are used to change the formerly defined behavior of a refactoring, allowing the developer to create custom refactorings. Custom refactorings can be adapted to the developer's context, allowing to apply it to similar contexts. The approach also allows developers to customize batches through from the composition of two or more custom refactorings.

The proposed approach focuses on reducing the limitations caused by the low customization flexibility of automated refactoring tools. The approach also allows development companies to refine a library of standard and custom refactorings to adhere to their quality standards. In addition, our approach motivates that researchers to investigate better practices for customization of refactoring. Finally, it also motivates studies about the applicability of custom refactoring and its best practices.

This paper is structured as follows: The Section 5.2 presents the limitations of the current automated refactoring tools. Section 5.3 introduces the new approach. Section 5.4 discusses the approach‘s advantages and main challenges. Finally, Section 5.5 discusses the final considerations.

5.2 Motivation

This section presents the limitations when executing automated refactoring using Eclipse. Subsection 5.2.1 details the problems that were observed in the literature. Finally, subsection 5.2.2 illustrates a real example from Eclipse project in which a current automated refactoring tool does not provide complete support to realize the needed changes.

5.2.1 Problem Statement

Developers usually do not apply automated refactoring tools, despite the fact these tools are available for years through IDEs (8, 17, 95). This happens because developers are not satisfied with automated refactoring tools' usability (17). This disappointment occurs due to existing limitations that restrict the applicability of automated refactoring. These limitations make it difficult or even impossible for developers to adapt refactoring within their context. Indeed, the automated refactoring tools provided by Eclipse are not fully customizable, *i.e.*, developers can customize the generated code only through the settings provided by Eclipse. However, these settings are specified by each program transformation type.

For instance, let us consider the refactoring transformation *Extract Method*. Figure 5.1 presents how the developer can change the method signature as the access modifier and exception throws through Eclipse IDE. However, some modifications are not allowed, such as (i) the addition of a call to the extracted method from other methods different from the source one (ii) the extraction of statements from different source methods. Indeed, modifications that go beyond the basic modifications described by Fowler (1) for each refactoring are not usually allowed.

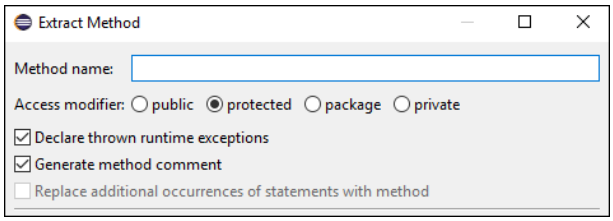


Figure 5.1: Eclipse's Extracted Method Tool Configuration

Because of this limitation, the automated refactoring tool provided by Eclipse does not support the creation of custom refactorings. The lack of customization can harm the adaptation of automated refactoring for different contexts. Once it is difficult to adapt, developers usually avoid the use of these tools, especially in critical program locations. Consequently, developers prefer to apply manually each refactoring to have higher control of the program correctness after refactoring (8, 17, 94).

5.2.2 Running Example

Figure 5.2 presents an example of a batch composed by two *Extract Methods* and two *Rename Methods* applied manually. In this example, we

can observe code modifications along the batch application. Unfortunately, Eclipse's automated refactoring tools do not allow their application. These code modifications are detailed in **Phase 1** (before the batch application) and **Phase 2** (along the batch application).

Phase 1 presents the `RepositoriesView` class. This class has three methods: `getGitDirs`, `saveDirStrings`, and `addActionsToToolbar`. Phase 2 presents an applying batch on the `RepositoriesView` class.

Code Modifications along a Batch As starting point, the developer applied two *Extract Methods* on `getGitDirs` and `saveDirStrings` methods, creating a new method called `getPrefs`. Developer applied two *Rename Methods* on these methods, resulting on the change of their names to `GetDirs` and `saveDir`. Then, the developer called the extracted method in the `getDirs` and `saveDir` methods. The developer also applied another code modification, calling the extracted method in the method `addActionsToToobal`, *i.e.*, a method different from the source ones.

These two applications of *Extract Method* illustrate two code modifications that are not supported by the automated tools provided by Eclipse. These code modifications are (i) to extract the source code of different methods to create a new method, and (ii) to call the extracted method from other methods that are not involved in the Extract Method. A lack of automated support to apply these code modifications force the developer manually select the source code of each method to create a new method. Because of these limitations, developers have to redo the automated *Extract Method* many times as needed to satisfy their goal. The repetitive application of automated refactorings requires additional effort and discourages the use of automated tools.

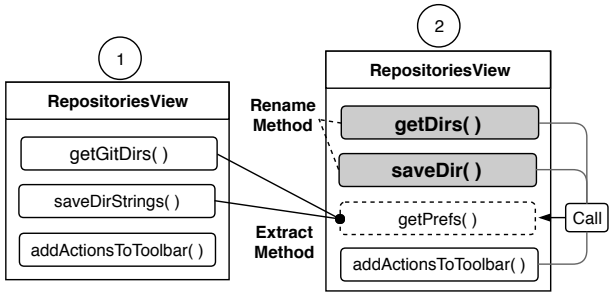


Figure 5.2: Eclipse's Batch Refactoring Example

Composing a Batch Developers often perform a refactoring by applying a program transformation in conjunction with other program transformations (8,

97). These program transformations are part of a batch. Figure 5.2 shows that the developer needs to complete the *Extract Methods* with *Rename Methods*. Once the renamed methods became cleaner, the choice of a new name is required; the new name will better represent the method's meaning after the extraction. Indeed, these two program transformation types are frequent and often used to compose batches (94). However, developers have reported the difficulty to compose a batch, in particular when choosing what program transformations may be combined (82).

5.3 Approach

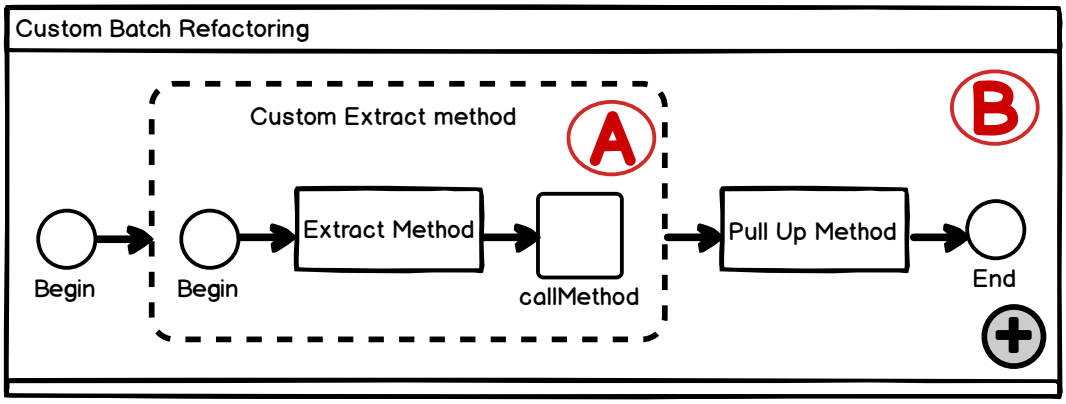


Figure 5.3: Prototype of the Refactoring Customization Process

We propose an approach to make these tools more flexible and address the limitations of the existing automated refactoring tools. Our approach allows that developers compose custom refactorings according to the context where they will be applied. For that, we split each refactoring into a set of minor modifications that compose it. We call them as *primitive modifications*. For instance, an *Extract Method* transformation can be fragmented into a set of **primitive modifications**. The first one consists of creating a new method. The second primitive modification is the extraction of a set of statements from the source method to the new method. The third primitive modification is the call to the new method from the source one. As well as the *Extract Method*, the other refactorings presented by Fowler (1) can also be split into primitive modifications.

Figure 5.3 presents an interface prototype for a possible plugin for Eclipse. This figure lets us better understand the refactoring customization process through the use of primitive modifications. It is possible to observe a line flow indicating the order of the modifications that will be applied. We call this flow as modification flow. Besides, we can also see two indicators **A** and **B**.

The box containing the indicator B represents a custom batch refactoring that includes a previously custom refactoring, the box A. Inside each box, we have a beginning point indicating where the modification flow will start. Next, we have a list of primitive modifications that will be applied in order, from left to right. Note that refactorings are composed by a set of primitive modification as seen previously for *Extract Method*. Thus, in order to improve the usability, the developer is able to directly add refactorings to the modification flow. In this way, the developer does not have to individually add each primitive modification that composes a well-known refactoring. From now, we will call as modification a modification flow's item.

Every modification has an output and may also need an input. The input data are the required configurations that are fundamental for the application of the modifications. For instance, when applying an *Extract Method*, we must select a set of statements that will be extracted into a new method. The selected statements are the input for this modification. Each modification may have some optional configurations that can be modified by the developer. For instance, Figure 5.3 has the primitive modification *callMethod*. For this modification, a possible optional configuration is to set where the method will be called: at the beginning or end of the caller method. The input, which is a fundamental configuration, in this case will be the selection of which method will be called and by whom.

Finally, the output is the generated code after the application of the modification. Each output will be used as input for the next modification on the flow. For instance, the extracted method could be considered the output for the transformation *Extract Method*. Therefore, if we look again at the primitive modification *callMethod* in Figure 5.3, we can note that it is applied right after the *Extract Method*. Thus, the modification *callMethod* will use the *Extract Method's* output as its input.

5.3.1 Customizing Single Refactoring

Previous studies show that a single refactoring transformation may not be enough to fix the target structural problem (94). Hence, developers often edit the refactored code to reach their initial goal. In other words, the developer is customizing refactoring by adding or removing modifications to adapt to their context.

To customize a refactoring using our approach, the developer should select first which existing refactoring will be customized. After selecting the refactoring, the modification flow will be displayed similarly to Figure 5.3.

Then, the developer can change the refactoring by adding or removing modifications on the modification flow. To add a new modification, the developer needs to click on the plus button and select which kind of modification he wants to add. Then, the developer will select where the modification will be inserted on the flow. Finally, the developer will define the input and the output based on his context.

5.3.2 Customizing Batch Refactoring

As seen in 5.2.2, developers apply complementary transformations as part of a batch. Through our approach, developers will be able to create custom batches. Refactorings can be added to modification flow of other existing ones in the same way as primitive modifications. Thus, the developer can compose multiple refactorings in a modification flow as seen in Figure 5.3 in box **B**. In this box, the custom batch refactoring will apply first the custom *extract method*, then the *pull up method*. The approach allows the developers to create any batch refactoring they need, as long as the input data are correctly provided.

5.4 Discussion

This section discusses the proposed approach regarding limitations raised by previous works. We indicate how our approach overcomes or mitigates them. Finally, we discuss the main challenges to implement the approach.

5.4.1 Advantages

Flexibility. The literature discusses the need for a tool that can support definition of new program transformation types (9, 19). These studies expose the lack of flexibility to support the automated refactoring for different contexts. To address this limitation, we propose an approach to enable refactoring customizations. Developers can customize an existing refactoring or create a new one changing the modification flow. These custom refactorings can be adapted for different contexts. Developers can also produce customized batch refactorings that meet their needs.

Enhanced Predictability. The literature also discusses the difficulty in predicting how the code will become after the application of refactoring (9, 17). Using our approach, the developer can more easily understand the transformations, since the refactorings are customized by himself. Besides,

the developer can follow these transformations, step by step, throughout the modification flow. Hence, the final result after the application of the custom refactoring will be more predictable.

Step-wise Preview. Vakilian *et al.* (17) interviewed some developers about the usability of the preview window of automated refactoring tools. The interviewees mentioned that the preview window is not very useful because it does not show the complete code or does not highlight important points. In our approach, developers will be able to preview the code after each modification, *i.e.*, in a step-wise manner. The modified code will be highlighted after each modification along the modification flow. Thus, developers can stop the preview and adjust the modifications to ensure that the final result will be as planned. Hence, this feature also improves *predictability*.

Reuse. Each custom refactoring can be saved and reused whenever necessary, including as part of another larger one. This composition of refactorings allows the creation of more complex refactorings, including batch refactorings. The reuse reduces the effort when applying custom refactoring continually. Besides, custom refactoring can be edited whenever needed, either for a specific use or to keep updated.

5.4.2 Challenges

The proposed approach has some limitations. The first one is the difficulty to define a set of modifications that satisfy all needs of the developers. To address this challenge, our approach will be based on Eclipse's AST. The AST is the base framework used by Eclipse for many powerful tools, including their current automated refactoring tool. This framework captures the semantic structure of a Java program allowing to identify and performing modifications.

Another challenge for realizing the approach is the requirement of prior input for the application of the modification. Developers need to dedicate more time to define all inputs for a custom refactoring. To reduce the additional effort, we plan to compose a set of predetermined transformations. We plan to apply learning techniques (*e.g.*, (98)) in various software projects in order to: (i) "learn" through real examples of "similar" transformations observed across these programs, and (ii) (semi-)automatically derive these predetermined transformations. Each predetermined transformation will have some simple points of customization. These points are responsible for enabling the adaptation of the predetermined transformation to the context where it will be applied.

5.5 Final Remarks

The proposed approach focuses on reducing the existing limitations caused by the low flexibility of automated refactoring tools. Developers will be able to customize refactorings to satisfy their needs. Besides, developers can reuse refactorings already created in a different context. This is possible because custom refactorings are easily adaptable. As well as developers, development companies can also customize refactorings to refine their transformations. This refinement can aim to adhere to quality standards. In addition, our approach motivates researchers to investigate better practices for the creation of custom refactorings. The approach is also an introduction to think of how automated refactoring tools can be performed. Our goal is to promote better trustability and, consequently, improve the usability of automated refactorings. Finally, it also motivates the study about the applicability of custom refactoring for different contexts.

As future work, we intend to define and classify the primitive modifications in detail. Then, we will design and implement a plugin for Eclipse. This plugin will enable the use of the proposed approach.

5.6 Summary of Chapter 5

In this chapter, we proposed an approach focused on reducing the existing limitations caused by the low flexibility of automated refactoring tools. Through this approach, developers will be able to customize refactorings to satisfy their needs. Besides, developers can reuse refactorings already created for similar contexts. This is possible because customized refactorings are easily adaptable. Developers can add or remove a modification among those that compose well-known refactorings. This additional flexibility also allows developers to customize the refactoring for different scenarios within their systems. In this way, developers are not induced by the tool to perform the same code modifications in different scenarios even when they do not properly fit their context. As future work, we plan to design, implement and evaluate our approach. We are currently using a semiotic engineering method (99) to assist refinements on the design of our proposed solution.

6
Final Conclusions

Code smells are code structures that are potentially harmful to program comprehension and maintenance. Thus, code smells should be detected and considered to be removed from a program. However, the abstract nature of code smell descriptions makes the their detection a challenging task. To make it worse, different developers may have different opinions about the existence of a smell in a code fragment. Thus, code smell detection should be customized taking into account developers' knowledge, once they are often responsible to detect smells on their software projects.

Smell detection and code refactoring are often performed in conjunction. Similar to smell detection, developers are, in practice, a main source of all technical knowledge required to decide when and how to refactor their code. Given a certain smelly structure, there might be a wide range of different ways to apply the same refactoring type. Moreover, different developers in the same project may have different strategies to refactor the same smelly structure. It is up to the developer to analyze the poor structure, evaluate the existence of the code smell, and decide how to refactor the code to fix the poor structure.

However, automated techniques for code smell detection, as well as for code refactoring, are inflexible. They are rigid in the sense they do not easily accommodate customizations to perform the detection of the same smell type or the application of the same refactoring type. As a consequence, the developer often feels obliged to perform these customizations manually. Moreover, they may give up in reviewing and refactoring the code if there is no proper tooling support.

This dissertation aims to evaluate how the approaches to smell detection and refactoring applications can be properly customized. First, we analyzed how the use of machine learning techniques, often stated as a promising way to detect smells, would have their effectiveness affected when evaluating developer-sensitive smells. In addition, we also evaluated how ML techniques would behave when detecting smells considered potentially important to be refactored out by developers.

After, we studied whether and how refactoring customizations occur in thirteen software projects. In this dissertation, we summarized the main

modification patterns adopted by developers when performing four of the most frequent refactoring types, namely *Extract Method*, *Inline Method*,*Pull Up Method*, and *Move Method*. Then, we evaluated the patterns in terms of two aspects: (i) are they partially or fully supported by Eclipse's refactoring tool? and (ii) their relation with the the full removal, introduction and mitigation of code smells.

The main contributions and their possible impact on the state-of-art and the state-of-practice are described as follows.

- *Evaluation of the use of machine learning techniques to customize smell detection based on the developer's knowledge.*

  Empowering smell detection tools with automated customization can help developers and companies to consider code smells that indeed harmful according to their quality standards. Otherwise, constant warnings of rigid, non-customized detection strategies can cause waste of time on the inspection of irrelevant smells, hinder the developer concentration on harmful smells according to their perception, or camouflage smells that are considered more harmful according to the developer's perception.

  We noticed that the use of machine learning is indeed promising to improve the detection of developer-sensitive smells, i.e., those smells that are relevant according to the developer's knowledge. We confirmed the potential of machine learning in two contexts: detection of smells in source code not produced by the code reviewers, and detection of smells that were actually refactored out by the project's developers.

- *A catalog of core and additional modifications applied by developers when applying specific refactoring types.*

  It is widely known that developers often neglect automated support for performing popular refactoring types. Thus, it is important to understand how developers modify the source code during the application of such refactoring types. In this way, automated tools will be able to provide adequate support for developers' refactoring needs, *e.g.*, through recommendations that assist developers in composing their custom refactorings. With adequate support, we hope that developers will be able to apply refactorings more quickly and produce refactored code that are less smell-prone or even error-prone.

- *A catalog of customized refactorings patterns applied by developers in practice and the impact of these customizations on code smells.*

  There is a wide variety of customizations that developers can carry out in practice. Therefore, recommending the most suitable customization

for a specific scenario can be an arduous task. However, the availability of a catalog containing evaluations of the different customizations can guide these tools to recommend them properly.

Our catalog evaluates the recommendations through two criteria. The first one is the frequency. Based on this criterion, we cataloged and discussed the most frequent patterns of each refactoring type. The second criterion is the impact that these patterns have on code smells. In this criterion, we cataloged which code smell types are most frequently introduced, removed, and had their intensity reduced for each of the most frequent patterns. Understanding which customizations are most frequent and that less often introduce smells allow the proper design of tools. For instance, these tools can better support refactoring by properly recommending customizations to developers during the application of a refactoring (which are likely to be more beneficial or at least less harmful to the code).

- *A catalog of recurring modifications present in customized refactorings that are not supported by automated refactoring tool provided by Eclipse.*

  One of the reasons that force developers to apply their refactorings manually is the impossibility of performing them automatically (17, 9). This impossibility occurs because the current tools are inflexible and do not allow developers to properly customize refactorings for their scenarios.

  Therefore, we cataloged which code modifications that compose the most frequent customization patterns are not possible to be fulfilled by the Eclipse's automated refactoring tool. For this, we reproduced the developers' scenarios and tried to replicate the same refactoring performed by them in these scenarios. In this way, we were able to list the limitations encountered during the replication. We hope this contribution serves as a basis for the development of further refactoring tools and improvements.

- *The proposal of an automated refactoring tool that provides support for refactoring customization.*

  Once we understand which patterns are more frequent and which limitations prevent the application of these patterns in an automated tool, we decided to design the prototype of a more flexible tool in order to overcome these limitations.

  In our prototype, we treated refactorings as a mutable set of code modifications, where developers can add or remove code modifications

as he sees to fit. Then, developers will be able to organize the order that the modifications will be applied on his code in a step-wise manner. Finally, the prototype will save the customized refactoring to be reused whenever necessary. We hope that this approach will stimulate future research that involves the development of tools to support refactoring.

Bibliography

[1] FOWLER, M.. **Refactoring**. Addison-Wesley Professional, 1 edition, 1999.

[2] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the impact of refactoring on smells**. In: FSE'17, p. 465–475, 2017.

[3] ABBES, M.; KHOMH, F.; GUEHENEUC, Y.-G. ; ANTONIOL, G.. **An empirical study of the impact of two antipatterns, blob and spaghetti code, on programcomprehension**. In: 15TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), p. 181–190. IEEE, 2011.

[4] KHOMH, F.; PENTA, M. D.; GUÉHÉNEUC, Y.-G. ; ANTONIOL, G.. **An exploratory study of the impact of antipatterns on class change- and fault-proneness**. Empirical Software Engineering, 17(3):243–275, Aug. 2011.

[5] YAMASHITA, A.; MOONEN, L.. **Exploring the impact of inter-smell relations on software maintainability: An empirical study**. In: PROCEEDINGS OF THE 2013 INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '13, p. 682–691, Piscataway, NJ, USA, 2013. IEEE Press.

[6] HOZANO, M.; GARCIA, A.; FONSECA, B. ; COSTA, E.. **Are you smelling it? investigating how similar developers detect code smells**. Inf. Softw. Technol., 93(C):130–146, Jan. 2018.

[7] DE MELLO, R.; UCHÔA, A.; OLIVEIRA, R.; OIZUMI, W.; SOUZA, J.; MENDES, K.; OLIVEIRA, D.; FONSECA, B. ; GARCIA, A.. **Do research and practice of code smell identification walk together? a social representations analysis**. In: 2019 ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–6. IEEE, 2019.

[8] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A.. **How we refactor, and how we know it**. TSE'12, 38(1):5–18, 2012.

[9] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **An empirical study of refactoring challenges and benefits at Microsoft**. TSE'14, 40(7):633–649, 2014.

[10] FONTANA, F. A.; BRAIONE, P. ; ZANONI, M.. **Automatic detection of bad smells in code: An experimental assessment.** Journal of Object Technology, 11(2):5–1, 2012.

[11] PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R. ; LUCIA, A.. **Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells**. 2014 IEEE International Conference on Software Maintenance and Evolution, p. 101–110, Sept. 2014.

[12] MÄNTYLÄ, M. V.; LASSENIUS, C.. **Subjective evaluation of software evolvability using code smells: An empirical study**, volumen 11. Springer, May 2006.

[13] MÄNTYLÄ, M. V.. **An experiment on subjective evolvability evaluation of object-oriented software: explaining factorsand interrater agreement**. In: EMPIRICAL SOFTWARE ENGINEERING, 2005. 2005 INTERNATIONAL SYMPOSIUM ON, volumen 00, p. 10 pp.–, Nov. 2005.

[14] D. MELLO, R. M.; OLIVEIRA, R. F. ; GARCIA, A. F.. **On the influence of human factors for identifying code smells: A multi-trial empirical study**. In: 2017 ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 68–77, Nov 2017.

[15] SILVA, D.; TSANTALIS, N. ; VALENTE, M. T.. **Why we refactor?** In: FSE'16, p. 858–870, 2016.

[16] TSANTALIS, N.; CHATZIGEORGIOU, A.. **Identification of extract method refactoring opportunities for the decomposition of methods**. Journal of Systems and Software (JSS), 84(10):1757–1782, 2011.

[17] VAKILIAN, M.; CHEN, N.; NEGARA, S.; RAJKUMAR, B. A.; BAILEY, B. P. ; JOHNSON, R. E.. **Use, disuse, and misuse of automated refactorings**. In: PROCEEDINGS OF THE 34TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 233–243. IEEE Press, 2012.

[18] TENORIO, D.; BIBIANO, A. C. ; GARCIA, A.. **On the customization of batch refactoring**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL WORKSHOP ON REFACTORING, p. 13–16. IEEE Press, 2019.

[19] LIN, Y.; PENG, X.; CAI, Y.; DIG, D.; ZHENG, D. ; ZHAO, W.. **Interactive and guided architectural refactoring with search-based recommendation**. In: FSE'16, p. 535–546, 2016.

[20] TSANTALIS, N.; CHAIKALIS, T. ; CHATZIGEORGIOU, A.. **"ten years of jdeodorant: Lessons learned from the hunt for smells"**. In: 25TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 4–14, 2018.

[21] SZŐKE, G.; NAGY, C.; FÜLÖP, L.; FERENC, R. ; GYIMÓTHY, T.. **Faultbuster: An automatic code smellrefactoring toolset**. In: SCAM'15, p. 253–258, 2015.

[22] FERNANDES, E.; OLIVEIRA, J.; VALE, G.; PAIVA, T. ; FIGUEIREDO, E.. **A review-based comparative study of bad smell detection tools**. In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING (EASE), p. 18:1–18:12, 2016.

[23] LANZA, M.; MARINESCU, R. ; DUCASSE, S.. **Object-Oriented Metrics in Practice**. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[24] MUNRO, M.. **Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code**. 11th IEEE International Software Metrics Symposium (METRICS'05), p. 15–15, 2005.

[25] FERREIRA, M.; BARBOSA, E.; MACIA, I.; ARCOVERDE, R. ; GARCIA, A.. **Detecting architecturally-relevant code anomalies: a case study of effectiveness and effort**. In: PROCEEDINGS OF THE 29TH ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, p. 1158–1163. ACM, 2014.

[26] CARBONELL, J. G.; MICHALSKI, R. S. ; MITCHELL, T. M.. **An overview of machine learning**. In: MACHINE LEARNING, p. 3–23. Elsevier, 1983.

[27] HOZANO, M.; ANTUNES, N.; FONSECA, B. ; COSTA, E.. **Evaluating the accuracy of machine learning algorithms on detecting code smells for different developers**. In: PROCEEDINGS OF THE 19TH INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS, p. 474–482, 2017.

[28] KHOMH, F.; VAUCHER, S.; GUÉHÉNEUC, Y. G. ; SAHRAOUI, H.. **A bayesian approach for the detection of code and design smells**. In: QUALITY SOFTWARE, 2009. QSIC'09. 9TH INTERNATIONAL CONFERENCE ON, p. 305–314. IEEE, 2009.

[29] MAIGA, A.; ALI, N.; BHATTACHARYA, N.; SABANE, A. ; GUEHENEUC, YANN-GAEL ANDAIMEUR, E.. **SMURF: A SVM-based Incremental Anti-pattern Detection Approach**. 2012 19th Working Conference on Reverse Engineering, p. 466–475, Oct. 2012.

[30] FONTANA, F. A.; MÄNTYLÄ, M. V.; ZANONI, M. ; MARINO, A.. **Comparing and experimenting machine learning techniques for code smell detection**. Empirical Software Engineering, June 2015.

[31] ALSHAYEB, M.. **Empirical investigation of refactoring effect on software quality**. Information and software technology, 51(9):1319–1326, 2009.

[32] MOHA, N.; GUEHENEUC, Y.-G.; DUCHIEN, L. ; LE MEUR, A.-F.. **Decor: A method for the specification and detection of code and design smells**. IEEE Transactions on Software Engineering, 36(1):20–36, 2009.

[33] PALOMBA, F.; PANICHELLA, A.; DE LUCIA, A.; OLIVETO, R. ; ZAIDMAN, A.. **A textual-based technique for smell detection**. In: 2016 IEEE 24TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 1–10. IEEE, 2016.

[34] BAVOTA, G.; DE LUCIA, A. ; OLIVETO, R.. **Identifying extract class refactoring opportunities using structural and semantic cohesion measures**. Journal of Systems and Software, 84(3):397–414, 2011.

[35] PALOMBA, F.; ZAIDMAN, A.; OLIVETO, R. ; DE LUCIA, A.. **An exploratory study on the relationship between changes and refactoring**. In: 2017 IEEE/ACM 25TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 176–185. IEEE, 2017.

[36] OUNI, A.; KESSENTINI, M. ; SAHRAOUI, H.. **Search-based refactoring using recorded code changes**. In: 2013 17TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, p. 221–230. IEEE, 2013.

[37] WEISSGERBER, P.; DIEHL, S.. **Identifying refactorings from source-code changes**. In: 21ST IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE'06), p. 231–240. IEEE, 2006.

[38] TSANTALIS, N.; GUANA, V.; STROULIA, E. ; HINDLE, A.. **A multidimensional empirical study on refactoring activity**. In: 23RD AN-

NUAL INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND SOFTWARE ENGINEERING (CASCON), p. 132–146, 2013.

[39] FLURI, B.; WUERSCH, M.; PINZGER, M. ; GALL, H.. **Change distilling: Tree differencing for fine-grained source code change extraction**. IEEE Transactions on software engineering, 33(11):725–743, 2007.

[40] XING, Z.; STROULIA, E.. **Refactoring practice: How it is and how it should be supported-an eclipse case study**. In: ICSM'06, p. 458–468. IEEE, 2006.

[41] OLIVEIRA, J.; GHEYI, R.; MONGIOVI, M.; SOARES, G.; RIBEIRO, M. ; GARCIA, A.. **Revisiting the refactoring mechanics**. In: IST'19, 2019.

[42] OLIVEIRA, D.. **On the sensitivity of machine learning techniques to detect developer-sensitive smells**. In: TO BE SUBMITTED TO A JOURNAL.

[43] OLIVEIRA, D.. **Assessing machine learning techniques on code smelldetection**. In: TO BE SUBMITTED TO A JOURNAL.

[44] OLIVEIRA, D.. **How do developers customize refactoring in practice?** In: TO BE SUBMITTED TO A JOURNAL.

[45] OLIVEIRA, D.; BIBIANO, A. C. ; GARCIA, A.. **On the customization of batch refactoring**. In: 2019 IEEE/ACM 3RD INTERNATIONAL WORKSHOP ON REFACTORING (IWOR), p. 13–16. IEEE, 2019.

[46] FOWLER, M.. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley, Boston, MA, USA, 1999.

[47] HOZANO, M.; GARCIA, A.; ANTUNES, N.; FONSECA, B. ; COSTA, E.. **Smells are sensitive to developers!: On the efficiency of (un)guided customized detection**. In: PROCEEDINGS OF THE 25TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, ICPC '17, p. 110–120, Piscataway, NJ, USA, 2017. IEEE Press.

[48] KHOMH, F.; VAUCHER, S.; GUÉHÉNEUC, Y.-G. ; SAHRAOUI, H.. **Bdtex: A gqm-based bayesian approach for the detection of antipatterns**. J. Syst. Softw., 84(4):559–572, Apr. 2011.

[49] AMORIM, L.; COSTA, E.; ANTUNES, N.; FONSECA, B. ; RIBEIRO, M.. **Experience report: Evaluating the effectiveness of decision trees for detecting code smells**. In: PROCEEDINGS OF THE 2015 IEEE

26TH INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE), ISSRE '15, p. 261–269, Washington, DC, USA, 2015. IEEE Computer Society.

[50] HOZANO, M.; GARCIA, A.; FONSECA, B. ; COSTA, E.. **Are you smelling it? investigating how similar developers detect code smells**. Inf. Softw. Technol., 93(C):130–146, Jan. 2018.

[51] MARINESCU, R.. **Detection Strategies: Metrics-Based Rules for Detecting Design Flaws**. In: PROCEEDINGS OF THE 20TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, ICSM '04, p. 350–359, Washington, DC, USA, 2004. IEEE Computer Society.

[52] PALOMBA, F.; BAVOTA, G.; DI PENTA, M.; OLIVETO, R.; DE LUCIA, A. ; POSHYVANYK, D.. **Detecting bad smells in source code using change history information**. In: 2013 28TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 268–278. Ieee, Nov. 2013.

[53] MOHA, N.; GUÉHÉNEUC, Y.-G.; MEUR, A.-F. L.; DUCHIEN, L. ; TIBERGHIEN, A.. **From a domain analysis to the specification and detection of code and design smells**. Formal Aspects of Computing, 22(3):345–361, May 2009.

[54] MOHA, N.; GUEHENEUC, Y.-G.; DUCHIEN, L. ; A. F. LE MEUR. **DECOR: A Method for the Specification and Detection of Code and Design Smells**. IEEE Transactions on Software Engineering, 36(1):20–36, Jan. 2010.

[55] GOPALAN, R.. **Automatic detection of code smells in Java source code**. PhD thesis, Dissertation for Honour Degree, The University of Western Australia, 2012.

[56] MANEERAT, N.; MUENCHAISRI, P.. **Bad-smell prediction from software design model using machine learning techniques**. 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE), p. 331–336, May 2011.

[57] FONTANA, F. A.; ZANONI, M.; MARINO, A. ; MÄNTYLÄ, M. V.. **Code Smell Detection: Towards a Machine Learning-Based Approach**. 2013 IEEE International Conference on Software Maintenance, p. 396–399, sep 2013.

[58] WITTEN, I. H.; FRANK, E.. **Data Mining: Practical machine learning tools and techniques**. Morgan Kaufmann, 2005.

[59] SCHUMACHER, J.; ZAZWORKA, N.; SHULL, F.; SEAMAN, C. ; SHAW, M.. **Building empirical support for automated code smell detection**. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering andMeasurement - ESEM '10, p. 1, 2010.

[60] SANTOS, J. A. M.; DE MENDONÇA, M. G. ; SILVA, C. V. A.. **An exploratory study to investigate the impact of conceptualization in god class detection**. In: PROCEEDINGS OF THE 17TH INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING, EASE '13, p. 48–59, New York, NY, USA, 2013. ACM.

[61] FONTANA, F. A.; MARIANI, E.; MORNIOLI, A.; SORMANI, R. ; TONELLO, A.. **An Experience Report on Using Code Smells Detection Tools**. 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, p. 450–457, Mar. 2011.

[62] PALOMBA, F.; BAVOTA, G.; DI PENTA, M.; OLIVETO, R.; POSHYVANYK, D. ; DE LUCIA, A.. **Mining Version Histories for Detecting Code Smells**. IEEE Transactions on Software Engineering, 5589(c):1–1, 2014.

[63] MITCHELL, T. M.. **Machine learning**. McGraw-Hill series in computer science. McGraw-Hill, Boston (Mass.), Burr Ridge (Ill.), Dubuque (Iowa), 1997.

[64] STEINWART, I.; CHRISTMANN, A.. **Support vector machines**. Springer Science & Business Media, 2008.

[65] PLATT, J.. **Fast training of support vector machines using sequential minimal optimization**. In: Schoelkopf, B.; Burges, C. ; Smola, A., editors, ADVANCES IN KERNEL METHODS - SUPPORT VECTOR LEARNING. MIT Press, 1998.

[66] HOLTE, R.. **Very simple classification rules perform well on most commonly used datasets**. Machine Learning, 11:63–91, 1993.

[67] HO, T. K.. **Random decision forests**. In: DOCUMENT ANALYSIS AND RECOGNITION, 1995., PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON, volumen 1, p. 278–282. IEEE, 1995.

[68] COHEN, W. W.. **Fast effective rule induction**. In: TWELFTH INTERNATIONAL CONFERENCE ON MACHINE LEARNING, p. 115–123. Morgan Kaufmann, 1995.

[69] QUINLAN, R.. **C4.5: Programs for Machine Learning**. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[70] HALL, M.; FRANK, E.; HOLMES, G.; PFAHRINGER, B. ; REUTEMANN, PETER ANDWITTEN, I. H.. **The weka data mining software: an update**. ACM SIGKDD explorations newsletter, 11(1):10–18, 2009.

[71] KAMPSTRA, P.; OTHERS. **Beanplot: A boxplot alternative for visual comparison of distributions**. Journal of Statistical Software, Code Snippets, 28(1):1–9, 2008.

[72] WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C. ; REGNELL, BJÖORN ANDWESSLÉN, A.. **Experimentation in Software Engineering: An Introduction**. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[73] DI NUCCI, D.; PALOMBA, F.; TAMBURRI, D. A.; SEREBRENIK, A. ; DE LUCIA, A.. **Detecting code smells using machine learning techniques: Are we there yet?** 2018.

[74] LANZA, M.; MARINESCU, R.. **Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems**. Springer Science & Business Media, 2007.

[75] ARCOVERDE, R.; MACIA, I.; GARCIA, A. ; VON STAA, A.. **Automatically detecting architecturally-relevant code anomalies**. In: RECOMMENDATION SYSTEMS FOR SOFTWARE ENGINEERING (RSSE), 2012 THIRD INTERNATIONAL WORKSHOP ON, p. 90–91. IEEE, 2012.

[76] MACIA, I.; GARCIA, A.; CHAVEZ, C. ; VON STAA, A.. **Enhancing the detection of code anomalies with architecture-sensitive strategies**. In: SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), 2013 17TH EUROPEAN CONFERENCE ON, p. 177–186. IEEE, 2013.

[77] DANIEL; OTHERS. **Um estudo para avaliar a eficiência de têcnicas de aprendizagem de maquina para detectar code smells sensiveis a desenvolvedores**. Undergraduate thesis. Federal University of Alagoas, 2018.

[78] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects**. p. 465–475, 2017.

[79] SANTOS, J. A. M.; MENDONÇA, M. G.; DOS SANTOS, C. P. ; NOVAIS, R. L.. **The problem of conceptualization in god class detection: agreement, strategies and decision drivers**. Journal of Software Engineering Research and Development, 2:1–33, 2014.

[80] BIBIANO, A. C.; FERNANDES, E.; OLIVEIRA, D.; GARCIA, A.; KALINOWSKI, M.; FONSECA, B.; OLIVEIRA, R.; OLIVEIRA, A. ; CEDRIM, D.. **A Quantitative Study on Characteristics and Effect of Batch Refactoring on Code Smells**. In: 13TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–11, 2019.

[81] TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L. M.; MAZINANIAN, D. ; DIG, D.. **Accurate and efficient refactoring detection in commit history**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '18, p. 483–494, New York, NY, USA, 2018. ACM.

[82] MEANANEATRA, P.. **Identifying refactoring sequences for improving software maintainability**. In: ASE'12, p. 406–409, 2012.

[83] TERRA, R.; VALENTE, M. T.; MIRANDA, S. ; SALES, V.. **JMove: A novel heuristic and tool to detect move method refactoring opportunities**. J. Syst. Softw. (JSS), 138:19–36, 2018.

[84] LIN, Y.; PENG, X.; CAI, Y.; DIG, D.; ZHENG, D. ; ZHAO, W.. **Interactive and guided architectural refactoring with search-based recommendation**. In: 24TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 535–546, 2016.

[86] BORGES, H.; VALENTE, M. T.. **What's in a GitHub star? Understanding repository starring practices in a social coding platform**. J. Syst. Softw. (JSS), 146:112–129, 2018.

[87] CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How does refactoring affect internal quality attributes? A multi-project study**. In: PROCEEDINGS OF THE 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 74–83, 2017.

[88] ALOMAR, E. A.; MKAOUER, M. W.; OUNI, A. ; KESSENTINI, M.. **On the impact of refactoring on the relationship between quality attributes and design metrics**. In: 2019 ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–11. IEEE, 2019.

[89] BAVOTA, G.; LUCIA, A. D.; PENTA, M. D.; OLIVETO, R. ; PALOMBA, F.. **An experimental investigation on the innate relationship between quality and refactoring**. J. Syst. Softw. (JSS), 107:1–14, 2015.

[90] GOUTTE, C.; GAUSSIER, E.. **A probabilistic interpretation of precision, recall and F-score, with implication for evaluation**. In: 27TH EUROPEAN CONFERENCE ON INFORMATION RETRIEVAL (ECIR), p. 345–359, 2005.

[91] BIEGEL, B.; SOETENS, Q. D.; HORNIG, W.; DIEHL, S. ; DEMEYER, S.. **Comparison of similarity metrics for refactoring detection**. In: PROCEEDINGS OF THE 8TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 53–62. ACM, 2011.

[92] PRETE, K.; RACHATASUMRIT, N.; SUDAN, N. ; KIM, M.. **Template-based reconstruction of complex refactorings**. In: 2010 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, p. 1–10. IEEE, 2010.

[93] MACIA, I.; ARCOVERDE, R.; GARCIA, A.; CHAVEZ, C. ; VON STAA, A.. **On the relevance of code anomalies for identifying architecture degradation symptoms**. In: 2012 16TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, p. 277–286. IEEE, 2012.

[94] CEDRIM, D.. **Understanding and Improving Batch Refactoring in Software Systems**. PhD thesis, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), 2018.

[95] PINTO, G. H.; KAMEI, F.. **What programmers say about refactoring tools?: An empirical investigation of stack overflow**. In: WRT'13, p. 33–36. ACM, 2013.

[96] FOSTER, S. R.; GRISWOLD, W. G. ; LERNER, S.. **WitchDoctor: IDE support for real-time auto-completion of refactorings**. In: ICSE'12, p. 222–232. IEEE, 2012.

[97] CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How does refactoring affect internal quality attributes?** In: SBES'17, p. 74–83, 2017.

[98] ROLIM, R.. **Learning syntactic program transformations from examples**. In: PROCEEDINGS OF ICSE 2017.

[99] DE SOUZA, C. S.. **The semiotic engineering of human-computer interaction**. MIT press, 2005.