FEDERAL UNIVERSITY OF CEARÁ

CAMPUS QUIXADÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO (PCOMP)

ACADEMIC MASTER'S DEGREE IN COMPUTING

DARWIN DE OLIVEIRA PINHEIRO

MEASURING TRIVIAL AND NON-TRIVIAL REFACTORINGS: A PREDICTIVE ANALYSIS AND INDEX PROPOSAL

QUIXADÁ

2024

DARWIN DE OLIVEIRA PINHEIRO

MEASURING TRIVIAL AND NON-TRIVIAL REFACTORINGS: A PREDICTIVE ANALYSIS AND INDEX PROPOSAL

Dissertation presented to the Academic Master's Course in Computing of the Programa de Pós-Graduação em Computação (PCOMP) of the Campus Quixadá of the Federal University of Ceará, as a partial requirement for obtaining a master's degree in Computer Science. Concentration Area: Computer Science

Advisor: Profa. Dra. Carla Ilane Moreira Bezerra

Co-advisor: Prof. Dr. Anderson Gonçalves Uchôa

QUIXADÁ

2024

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P718m Pinheiro, Darwin de Oliveira.
Measuring trivial and non-trivial refactorings: a predictive analysis and index proposal / Darwin de Oliveira Pinheiro. – 2024.
204 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Campus de Quixadá, Programa de Pós-Graduação em Computação, Quixadá, 2024.
Orientação: Profa. Dra. Carla Ilane Moreira Bezerra.
Coorientação: Prof. Dr. Anderson Gonçalves Uchôa.

1. Refactoring. 2. Software Maintenance. 3. Software Quality. 4. Supervised Learning. 5. Machine Learning. I. Título.

CDD 005

DARWIN DE OLIVEIRA PINHEIRO

MEASURING TRIVIAL AND NON-TRIVIAL REFACTORINGS: A PREDICTIVE ANALYSIS AND INDEX PROPOSAL

Dissertation presented to the Academic Master's Course in Computing of the Programa de Pós-Graduação em Computação (PCOMP) of the Campus Quixadá of the Federal University of Ceará, as a partial requirement for obtaining a master's degree in Computer Science. Concentration Area: Computer Science

Approved in:

Examination Committee

First and foremost, to God, for always guiding and blessing my path. To my wife, Ariane, for her unconditional support and encouragement in every challenge. To my parents, for believing in me and inspiring me to keep moving forward. And to myself, for all the effort and dedication that led me to achieve this accomplishment.

ACKNOWLEDGEMENTS

I thank God, who has been a constant source of strength and inspiration, helping me to overcome the challenges of this academic journey and providing guidance and direction along the way.

To my parents, Kermedy and Dalva, I am eternally grateful for being examples of wisdom and inspiration in my life.

To my wife, Ariane, I thank you for your love, patience, and unwavering support throughout this journey, helping me to overcome obstacles. Your presence and understanding were essential for me to successfully complete this stage.

To Profa. Dra. Carla Ilane Moreira Bezerra, my advisor, I am deeply grateful for your insightful guidance, valuable teachings, and for the opportunity to become part of the scientific community. Your mentorship was crucial in the development of this Master's dissertation.

To Prof. Dr. Anderson Gonçalves Uchôa, whose co-advising was indispensable, I appreciate your valuable insights and best practices that helped enhance the quality of the scientific research.

To the distinguished examining committee, composed of esteemed Prof. Dr. Marco Túlio Valente, Prof. Dr. Lincoln Souza Rocha, and Prof. Dr. Regis Pires Magalhães, I express my sincere gratitude for the time and attention given to evaluating this Master's dissertation. Your insightful comments and constructive criticism were essential in raising the quality of the work, providing valuable insights, and guiding it to a higher level. I deeply appreciate your commitment and generosity in sharing your knowledge, significantly contributing to my academic development.

To the esteemed faculty of the Graduate Program in Computer Science at the Federal University of Ceará in Quixadá, with whom I have had the privilege of interacting since the beginning of my undergraduate studies, I am immensely grateful for the dedication and excellence in the courses offered. The quality of teaching and the support received were crucial for my development and the completion of this program.

To all who, directly or indirectly, contributed to the completion of this Master's dissertation, I extend my deep and sincere gratitude.

“Science can never solve one problem without raising ten more problems”

(George Bernard Shaw)

RESUMO

A refatoração altera a estrutura interna do código sem modificar seu comportamento externo, melhorando a qualidade, manutenibilidade e legibilidade, além de reduzir a dívida técnica. Estudos indicam a necessidade de aprimorar a detecção e correção de refatorações, recomendando o uso de aprendizado de máquina para investigar motivações, dificuldades e melhorias no software. Esta dissertação tem como objetivo identificar a relação entre refatorações triviais e não triviais, além de propor uma métrica que avalia a trivialidade da implementação de refatorações. Inicialmente, utilizamos modelos classificadores de aprendizado supervisionado para examinar o impacto das refatorações triviais na predição das não triviais. Analisamos três conjuntos de dados, com 1.291 projetos de código aberto e aproximadamente 1,9M de operações de refatoração, utilizando 45 métricas de código. Foram utilizados os 5 modelos de classificação, em diferentes configurações do dataset. Em segundo lugar, propomos também uma métrica baseada em ML para avaliar a trivialidade da refatoração, considerando complexidade, velocidade e risco. O estudo examinou como a priorização de 58 featuers, apontadas por 15 desenvolvedores, afetou a eficácia de sete modelos de regressão. Analisou a eficácia dos de 7 modelos de regressão e ensemble. Além disso, verificou-se o alinhamento entre as percepções de 16 desenvolvedores experientes e os resultados dos modelos. Nossos resultados são promissores: (i) Algoritmos como Random Forest, Decision Tree e Neural Network tiveram melhor desempenho ao usar métricas de código para identificar oportunidades de refatorações; (ii) Separar refatorações triviais e não triviais melhora a eficiência dos modelos, mesmo em diferentes conjuntos de dados; (iii) Usar todas as features disponíveis supera a priorização feita pelos desenvolvedores nos modelos preditivos; (iv) Modelos ensemble, como Random Forest e Gradient Boosting, superam os modelos lineares, independentemente da priorização de features; e (v) Há forte alinhamento entre as percepções dos especialistas e os resultados dos modelos. Em resumo, esta dissertação contribuiu com o processo de refatoração, um apoio importante para os desenvolvedores, pois pode influenciar a decisão de aplicar ou não uma refatoração. Além de destacar insights, desafios e oportunidades para trabalhos futuros.

Palavras-chave: refactoring; feature extraction; code metrics; software maintenance; software quality; supervised learning; machine learning.

ABSTRACT

Refactoring changes the internal structure of the code without changing its external behavior, improving quality, maintainability, and readability, in addition to reducing technical debt. Studies indicate the need to improve the detection and correction of refactorings, recommending the use of machine learning to investigate motivations, difficulties, and improvements in software. This Master's dissertation aims to identify the relationship between trivial and non-trivial refactorings, in addition to proposing a metric that evaluates the triviality of implementing refactorings. Initially, we use supervised learning classifier models to examine the impact of trivial refactorings on the prediction of non-trivial ones. We analyzed three datasets, with 1,291 open source projects and approximately 1.9M refactoring operations, using 45 code metrics. The 5 classification models were used, in different dataset configurations. Second, we also propose an ML-based metric to evaluate the triviality of refactoring, considering complexity, speed, and risk. The study examined how the prioritization of 58 features, identified by 15 developers, affected the effectiveness of seven regression models. The effectiveness of 7 regression and ensemble models was analyzed. In addition, the alignment between the perceptions of 16 experienced developers and the results of the models was verified. Our results are promising: (i) Algorithms such as Random Forest, Decision Tree and Neural Network performed better when using code metrics to identify opportunities for refactorings; (ii) Separating trivial and non-trivial refactorings improves the efficiency of the models, even on different datasets; (iii) Using all available features outperforms the prioritization made by developers in predictive models; (iv) Ensemble models, such as Random Forest and Gradient Boosting, outperform linear models, regardless of feature prioritization; and (v) There is strong alignment between the perceptions of experts and the results of the models. In summary, this Master's dissertation contributed to the refactoring process, an important support for developers, as it can influence the decision of whether or not to apply a refactoring. In addition, it highlights insights, challenges and opportunities for future work.

Keywords: refactoring; feature extraction; code metrics; software maintenance; software quality; supervised learning; machine learning.

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODE

LIST OF ABBREVIATIONS AND ACRONYMS

ML	Machine Learning
IDE	Integrated Development Environment
LOC	Lines of Code
TP	True Positive
FP	False Positive
TN	True Negative
FN	False Negative
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
R^2	Coefficient of Determination
TI	Triviality Index
SS	Score of Similarity
FOC	Frequency of Commit
AST	Abstract Static Tree

CONTENTS

1 INTRODUCTION

In this chapter, we present the main motivations for carrying out this Master's dissertation, which aims to identify the relationship between trivial and non-trivial refactorings and propose a metric that evaluates the triviality of implementing refactorings. This chapter is organized as follows: (i) Section 1.1, we contextualize the topic of refactoring, address the main related themes and the motivation for this Master's dissertation; (ii) Section 1.2, we present the objectives, as well as the research questions; (iii) Section 1.3 we detail the methodological process used in this research; and (iv) Section 1.5 we present in detail the general organization of the chapters of this Master's dissertation.

1.1 Contextualization

Software maintenance is one of the most expensive activities in software engineering effort (Zarnekow; Brenner, 2005). According to Bertrand (1994), maintenance represents a total cost of 70% of a system. The main factor for this high cost is poor software quality (Dehaghani; Hajrahimi, 2013). Good quality software will generate lower costs and effort in maintenance activities (Kaur; Singh, 2019). Thus, the software industry considers that giving importance to software quality generates benefits for maintainability and system reliability (Malhotra; Jain, 2019).

During software maintenance, it is possible that developers introduce codes with poor structural quality in an unintentional or unintentional way (Ouni *et al.*, 2015). Over time, these poor-quality codes end up degrading the code quality, which can lead to failures in the future (Yamashita; Moonen, 2012; Uchôa *et al.*, 2020). One solution that can solve this problem is applying transformations to the source code, a very common type of transformation that meets this objective is software refactoring (Silva *et al.*, 2016).

Opdyke (1992) introduced the term refactoring, but this term only became popular with the book by Fowler (2018). Refactoring is defined as a transformation that changes the internal structure of the source code without changing the external behavior (Fowler, 2018). Maintaining the external behavior means that after applying the refactoring activity, the software should produce the same output as before.

Some benefits of using refactoring techniques in software are: (i) increased overall software quality; (ii) decreased maintenance costs; and (iii) increased developer productivity (Fowler, 2018; Moser *et al.*, 2007). In this way, the benefits extend to internal quality attributes such as coupling and cohesion and external software quality attributes. They can significantly improve the reusability and readability of systems (Mens; Tourwé, 2004; Bavota *et al.*, 2015; Malhotra; Chug, 2016).

Researchers have investigated different perspectives for the use of refactoring (Mens; Tourwé, 2004; Azeem *et al.*, 2019; Sobrinho *et al.*, 2018; Bibiano *et al.*, 2023), such as: (i) solutions that recommend refactorings to developers (Bavota *et al.*, 2015; Tsantalis *et al.*, 2018; Almogahed *et al.*, 2023; Bibiano *et al.*, 2024; Nikolaidis *et al.*, 2024); (ii) machine learning-based refactoring detection (Aniche *et al.*, 2020; AlOmar *et al.*, 2021; Nyamawe, 2022; Tan *et al.*, 2024); (iii) developer motivation to refactor code (Silva *et al.*, 2016; Palomba *et al.*, 2017), and (iv) obstacles to refactoring (Kim *et al.*, 2014; Sharma *et al.*, 2015; Liu *et al.*, 2024). However, even though refactoring has been investigated as a provider of benefits for software quality, whether by improving internal or external attributes, some studies indicate that in some cases, refactorings can negatively affect software maintenance.

The application of ML predictive models to aid developers in identifying refactoring opportunities for design improvement is a relatively recent field of research (Azeem *et al.*, 2019). Several studies have employed ML through unsupervised learning for identifying refactoring possibilities (Alkhalid *et al.*, 2010; Bryksin *et al.*, 2018; Tan *et al.*, 2024) while others have explored the use of supervised learning techniques to detect such opportunities (Aniche *et al.*, 2020; AlOmar *et al.*, 2021; Nyamawe, 2022; Alomar *et al.*, 2022).

Although many studies have explored how ML can be utilized to enhance refactoring techniques (Alkhalid *et al.*, 2010; Bryksin *et al.*, 2018; Aniche *et al.*, 2020; AlOmar *et al.*, 2021; Nyamawe, 2022; Panigrahi *et al.*, 2020; Tan *et al.*, 2024), few have focused on strategies to improve the accuracy of refactoring predictions made by these models. According to Kumar *et al.* (2019), software metrics play a crucial role in estimating the likelihood of refactoring at the class level, among other approaches.

Azeem *et al.* (2019) and Baqais and Alshayeb (2020) emphasize the need for further studies on how ML can identify refactoring opportunities, as well as the development of automated solutions involving refactoring. Kaur and Singh (2019) highlights that many refactorings commonly practiced in the industry are still unexplored. Moreover, the lack of a clear assessment of the triviality of refactorings creates challenges in selecting the ideal technique and ensuring that the system's behavior is not affected (Akhtar *et al.*, 2022). Additionally, developers tend to prefer manual, time-consuming, and risky refactorings, avoiding automated tools to minimize the risk of introducing new bugs (Abid *et al.*, 2022; Silva *et al.*, 2016). Consequently, our motivation is based on the identified needs, proposing a solution based on ML that can be automated and provides developers with reliable information to decide whether to apply automated refactoring.

Additionally, during our literature review, we observed that the classification of refactorings is a commonly adopted approach in existing studies. Some studies classify or group refactorings according to their general purpose (Fernandes *et al.*, 2020; Sellitto *et al.*, 2021; Smiari *et al.*, 2022). Others simplify them into a binary classification: refactored and non-refactored (Eposhi *et al.*, 2019; Nyamawe, 2022). AlOmar *et al.* (2021) classifies its refactorings into: internal, external, fix bug, and fix smell. In contrast, Sellitto *et al.* (2021), to achieve the objective of the study, groups refactorings into: composing methods, moving resources, organizing data, simplifying method calls, dealing with generalizations, and others. Thus, to achieve the objective of this work, in our first study, we classify the types of refactoring into trivial and non-trivial. This classification is based on the number of changes made to the source code. A better contextualization is presented in Section 2.1.1. In the second section, we adopt another line of thought, which is to identify the triviality of each refactoring operation 5.1. In this context, despite several studies being carried out to investigate the benefits or challenges involving refactorings, it is still necessary to carry out more empirical studies addressing the topic to fill in the gaps discovered.

1.2 Objective and research questions

After highlighting the importance of refactoring for software quality and the efforts of researchers to investigate solutions, motivations, difficulties, and improvements in this practice. In addition, to the need to use machine learning to solve design problems, the main objective of this Master's dissertation is to identify the relationship between trivial and non-trivial refactorings and propose a metric that evaluates the triviality of implementing a refactoring. To this end, we first seek to identify and classify refactorings into trivial and non-trivial, using machine learning algorithms to improve software quality and maintainability. We also propose a metric that evaluates the triviality of implementing refactoring operations, considering simplicity, speed and risk. The main research questions, along with sub-questions, are presented below:

RQ$_1$: How effective are trivial refactorings in predicting non-trivial refactorings?

RQ$_{1.1}$: What is the performance of ML algorithms to predict trivial and nontrivial refactorings?

RQ$_{1.2}$: How effective is the inclusion of trivial refactorings to predict non-trivial refactorings?

RQ$_{1.3}$: How effective are data balancing techniques in the prediction of trivial and non-trivial refactorings?

RQ$_{1.4}$: Can the best models be carried over to different contexts?

RQ$_{2}$: How can an effective refactoring triviality index be developed using a machine learning approach from a developer's perspective?

RQ$_{2.1}$: Which code metrics are considered most relevant by developers to determine the triviality of a refactoring operation?

RQ$_{2.2}$: How do different Machine Learning (ML) techniques behave in predicting the code refactorings triviality index?

RQ$_{2.3}$: What is the impact of prioritizing features ranked by developers on the effectiveness of triviality index prediction models?

RQ$_{2.4}$: To what extent is the proposed triviality index aligned with the developers' perception regarding the triviality of applying refactorings?

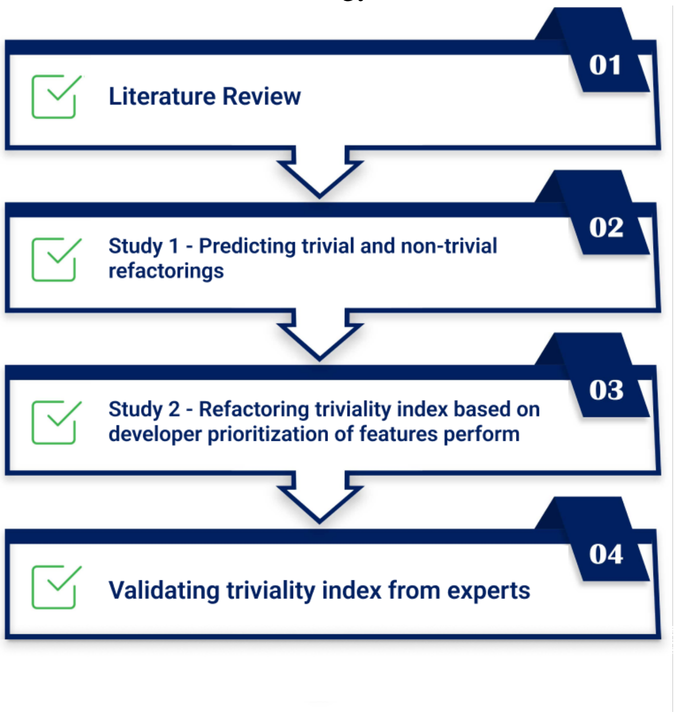
From this, we can list some specific objectives: (i) Identify the performance of machine learning algorithms in predicting trivial and non-trivial refactorings; (ii) Analyze the effectiveness of predictive models in different refactoring data domains, identifying the influence that trivial refactorings have on non-trivial refactorings; (iii) Investigate the relationship between feature prioritization by developers and the performance of machine learning models in predicting the refactorings triviality index; and, (iv) Evaluate the accuracy of predictive models regarding developers' perception of the refactoring's triviality, identifying areas of agreement and divergence.

1.3 Research methodology

This section presents the proposed methodology to explore and develop the triviality index and describe each step. Figure 1 presents the methodology used to execute the Master's dissertation.

1. **Literature review:** In this phase, we conducted an ad-hoc literature review to investigate opportunities, tools used, deficiencies, and needs for refactoring, as well as using machine learning to solve design problems focused on refactoring.

Figure 1 – Overview of the research methodology


01
Literature Review
02
Study 1 - Predicting trivial and non-trivial refactorings
03
Study 2 - Refactoring triviality index based on developer prioritization of features perform
04
Validating triviality index from experts

Source: Prepared by the author.

2. **Study 1 - Predicting trivial and non-trivial refactorings:** After identifying gaps in the literature review and research opportunities, we conducted a study based on machine learning techniques. The study is described in Chapter 4 and aims to investigate how trivial refactorings affect the prediction of non-trivial refactorings. In this step, we combined different types of refactorings in different contexts to investigate the influence caused by supervised learning algorithms on the classification problem. With this, we obtained efficient models based on *Random Forest* and *Decision Tree* to be used in a more practical solution for developers.
3. **Study 2 - Refactoring triviality index based on developer prioritization of features perform:** With the results found in the previous study, we present in Chapter 5 a new study that addresses the use of software refactoring through the analysis and proposal of a metric called "Triviality Index", which evaluates the degree of difficulty of implementing a refactoring operation from the point of view of software developers, considering its complexity, speed and risk. The metric is generated from code metrics with and without feature prioritization by developers and trained with supervised learning algorithms for the regression problem.
4. **Validating triviality index from experts:** After defining the triviality index, we checked

the agreement between the effectiveness of the models and the perceptions of developers with experience in refactoring activities. To do this, we first selected a set of code examples containing different types of refactoring operations. We then sent a survey to capture developers' perceptions of their experience in applying refactorings. The feedback provided by the experts helped refine and validate the index, ensuring its effectiveness in practical scenarios.

1.4 Study Replicability

An essential practice in any scientific research is ensuring that studies are replicable. This means that every manuscript must provide detailed information on how to reproduce the study. In this context, the open science movement aims to make all research artifacts publicly available, thereby increasing transparency and reproducibility in the scientific process (Mendez *et al.*, 2020). To support open science, the replication package for each study in this Master's dissertation is available on Zenodo[1] and Github[2], an online repository hosted by CERN. Table 1 details the location of each package. For each replication package, we provide all collected data, metric definitions, survey results, predictive models, and scripts.

Table 1 – Replication Package Available

Replication Package	**Host**
On the Effectiveness of Trivial Refactorings in Predicting Non-trivial Refactorings	https://doi.org/10.5281/zenodo.6800385, https://doi.org/10.5281/zenodo.7820168
Towards an effective refactoring triviality index: A Machine Learning Approach from a Developer's Perspective	https://doi.org/10.5281/zenodo.13766290 https://github.com/d4rwln/TrivialityIndex

Source: Prepared by the author.

1.5 Organization

This chapter presented the context, motivation, objectives, research questions, and methodology used in this Master's dissertation. The rest of the work is organized as follows: in Chapter 2, the main concepts about refactoring used in this work are presented, as well as the concepts about machine learning and metrics used to measure models. In Chapter 3, we discuss and compare the main related works focused on both the refactoring activity and the

[1] https://zenodo.org/
[2] https://github.com/

use of learning techniques to predict refactorings. In Chapter 4, we present a study that aims to investigate the influence that trivial refactorings at the class level have on the prediction of non-trivial refactorings using supervised machine learning. In Chapter 5 we present a study that aims to develop an index that evaluates the triviality of refactoring operations based on the following aspects: simplicity, speed and risk. Finally, in Chapter 6 we detail the considerations and main results of this Master's dissertation, some publications and future work aimed at extending this work.

2 BACKGROUND

This chapter presents the key concepts used in this Master's dissertation. Section 2.1 overviews software refactorings. Section 2.2 presents software metrics and their use in software engineering studies. Finally, Section 2.3 outlines the key concepts of machine learning techniques.

2.1 Refactoring

Refactoring is a key software maintenance activity aimed at enhancing the internal structure of a system without changing its external behavior. The concept was first introduced in the PhD thesis of Opdyke (1992), which focused on restructuring at the code-level, but gained popularity through the work of Fowler (2018). Fowler defines refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without altering its observable behavior". In his book, Fowler introduced a catalog of 72 refactoring techniques. In the latest edition (Fowler, 2018), he updated this catalog to reflect current trends and modern software development practices. The catalog is now available and continuously maintained online[1], providing an explanatory outline for each refactoring technique.

There are refactorings beyond the catalog proposed by Fowler (2018), including those integrated into Integrated Development Environment (IDE) tools and others manually employed by developers. Refactoring is frequently used to address code smells (Murphy-Hill *et al.*, 2012) and can enhance various quality attributes, such as readability and maintainability. It also plays a crucial role in modernizing legacy systems (Lacerda *et al.*, 2020). Additionally, refactoring is widely considered the most effective strategy for reducing technical debt (Avgeriou *et al.*, 2016; Pérez *et al.*, 2021).

In summary, applying refactorings provides benefits to the software development process, including: (i) improving software design and preventing its deterioration; (ii) making the code easier to understand, reducing the time other developers need to grasp it; (iii) accelerating the development of new features; (iv) facilitating bug detection through improved code comprehension; (v) aiding in the analysis and maintenance of legacy systems; and (vi) enhancing overall software quality by reinforcing key quality attributes, among other advantages (Fowler, 2018).

[1] https://refactoring.com/catalog/

We illustrate an example of a refactoring operation and its benefits as follows. The Listing 2 provides an example of the *Extract Method* refactoring operation applied to Listing 1.

```java
1 public void printAll() {
2     printBanner();
3 
4     // Print details.
5     System.out.println("name:  " + name);
6     System.out.println("amount:  " + getOutstanding());
7 }
```

Source Code 1 – Java code example before refactoring

```java
1 public void printAll() {
2     printBanner();
3     printDetails(getOutstanding());
4 }
5 
6 public void printDetails(double outstanding) {
7     System.out.println("name:  " + name);
8     System.out.println("amount:  " + outstanding);
9 }
```

Source Code 2 – Example of Java code refactored using the technique *Extract Method*

The *Extract Method* operation is used when multiple lines of code can be grouped because they are the same concern. The solution involves extracting a code fragment into a new method or function and replacing the extracted code in the original method with a call to this new function. In the source code illustrated in Listing 1, the code that prints details to the program's console is located on lines 5 and 6, with a code smell comment on line 4 indicating what is to be done. The *Extract Method* operation first extracts the code from lines 5 and 6 into a new method. Next, the comment on line 4 is used to name the newly created method `printDetails(double outstanding)`, as shown in the Listing 2. This process is also strongly recommended when the *Long Method* code smell is detected (Fowler, 2018).

Refactorings can vary in their levels and purposes. Fowler (2018) observed that for each refactoring operation, there is a logically inverse operation. For instance, the *Extract Method* operation, which extracts a code fragment into a separate method, has the *Inline Method* refactoring as its inverse. *Inline Method* simplifies code by replacing a method call with the body of that method (Fowler, 2018).

2.1.1 Trivial and Non-Trivial Refactorings

Rura (2003) suggests that refactorings be grouped according to a sequence of steps. Fowler (2018) suggests that refactorings can be categorized based on a combination of factors—some having a substantial impact on enhancing code design, while others consist of useful techniques that offer more general improvements.

Currently, there is no classification or grouping of refactorings in the literature that is considered a standard. Fowler (2018) describes cases where the same refactoring can be either trivial or non-trivial, often depending on changes to the code's scope. However, he neither outlines nor defines rules for classifying a refactoring as trivial or not. Thus, we extend the focus on this topic in our work, aiming to identify the triviality of refactorings. Initially, we define trivial refactorings as those involving minor changes to the code, while non-trivial ones result in more significant modifications. Subsequently, we conducted a study to assess the triviality of refactorings based on various criteria.

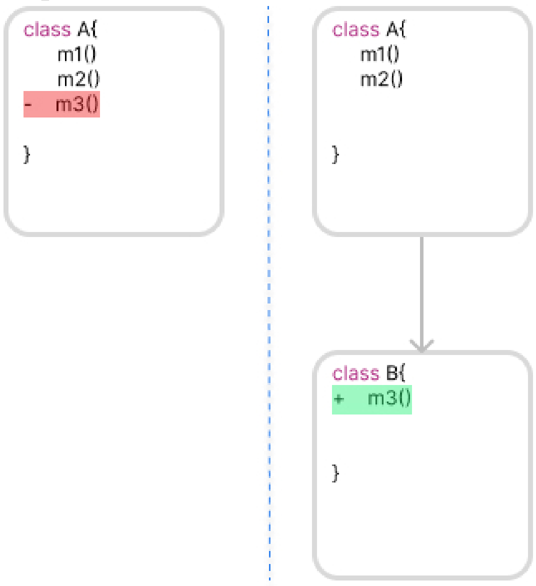
2.1.1.1 Non-Trivial

Our choice was driven by the lack of studies focusing on the main refactorings used in the industry (Murphy-Hill *et al.*, 2012; Khanam, 2018; Liu *et al.*, 2012). Thus, we considered the following refactorings as non-trivial refactorings: *Extract Class*, *Extract Superclass*, *Extract Subclass*, *Inline Class*, *Move Class*, *Move and Rename Class*, *Move Method*, *Extract Method*, *Inline Method*, *Extract Variable*, and *Inline Variable*.

Figure 2 illustrates an example of refactoring that separates the responsibilities of a class. The process involves creating a new class and moving the relevant attributes and methods to it. This refactoring can be applied when: (i) a class lacks a clear responsibility, and (ii) a subset of attributes and methods appear to form a distinct group (Fowler, 2018). This refactoring is considered non-trivial because it makes significant changes to the code's design. Additionally, it can involve other refactorings cataloged by Fowler (2018), such as *Move Method*, *Move Field*, and *Change Reference to Value*.

Extract Class

Figure 2 – Refactoring example *Extract Class*



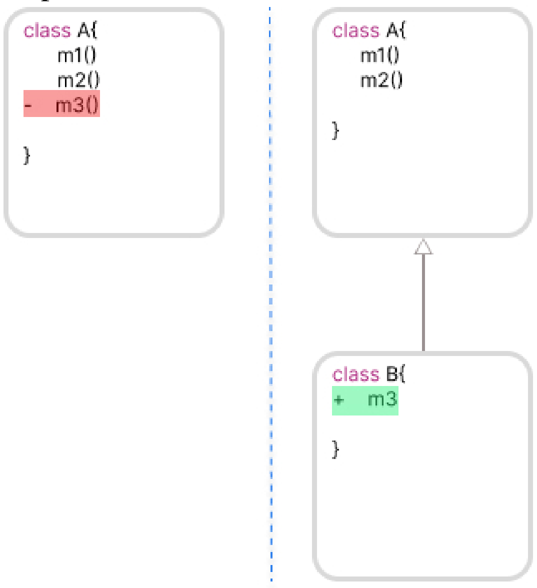
Source: Prepared by the author.

Figure 3 presents an example of a refactoring that isolates data used only in specific cases. The procedure involves creating a subclass and moving the relevant attributes and methods to it. This refactoring can be applied when a class contains fields that are not commonly used. It is considered Non-Trivial because it introduces a significant change to the *code design*. This refactoring may also incorporate other refactorings cataloged by Fowler (2018), such as: *Push Down Method*, *Push Down Field*, *Replace Constructor With Factory Method*, and *Replace Conditional with Polymorphism*.

Extract Subclass

Figure 4 illustrates an example of refactoring that merges similar tasks found in different classes. The procedure involves using basic inheritance to extract common attributes and methods from similar classes and place them into a superclass. This refactoring can be applied when similar tasks are identified across two or more classes. An alternative to inheritance for consolidating duplicate behavior is delegation, which can be combined with *Extract Class* (Fowler, 2018). It is also considered Non-Trivial because it introduces significant changes to the code design. This refactoring may incorporate other refactorings cataloged by Fowler (2018), such as *Pull Up Method*, *Pull Up Field*, *Pull Up Constructor Body*, and *Change Signature*.

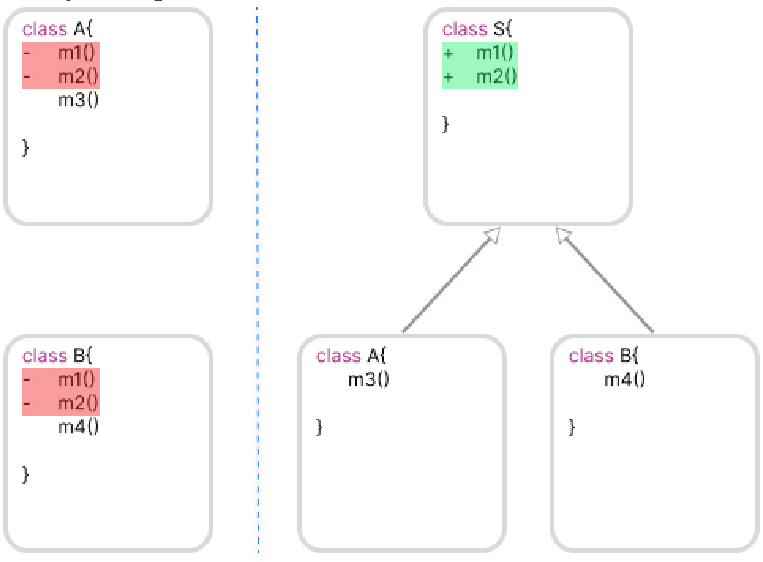
Figure 3 – Refactoring example *Extract Subclass*



Source: Prepared by the author.

Extract Superclass

Figure 4 – Refactoring example *Extract Superclass*

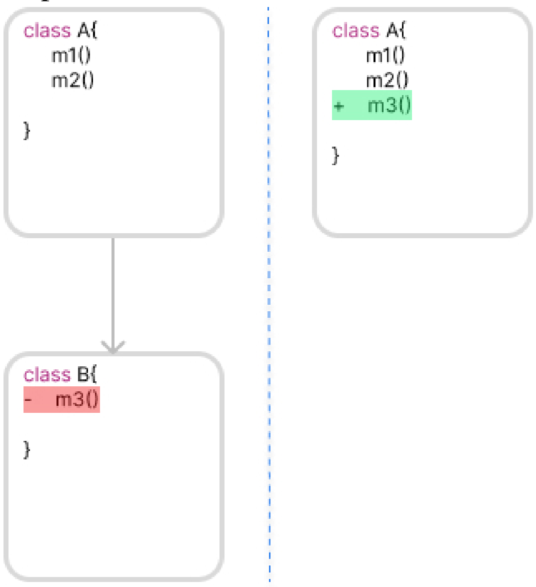


Source: Prepared by the author.

Inline Class

Figure 5 shows an example of *Inline Class*, also known as *Merge Class*, which is used to combine data and behaviors into a single class. The procedure consists of moving the attributes and methods of the original class to the final class, then deleting the original class. We can perform this refactoring when: (i) we identify that it is no longer worth having a class; and (ii) we want to refactor a pair of classes with different resource allocation. It is also considered Non-Trivial because it makes a larger change to the code design. This refactoring can make use

Figure 5 – Refactoring example *Inline Class*

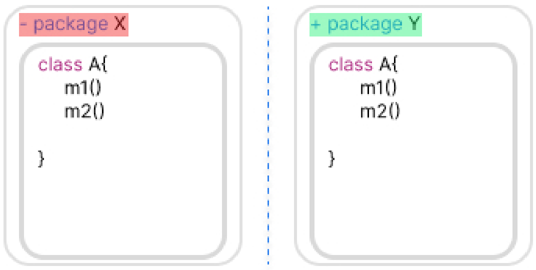


Source: Prepared by the author

of other refactorings cataloged by Fowler (2018) such as: *Move Method*, *Move Field* and *Inline Method*.

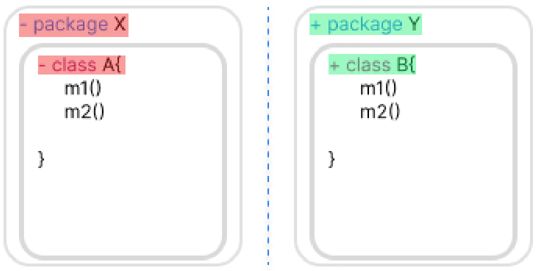
Move Class

Figure 6 – Refactoring *Move Class*



Source: Prepared by the author.

Figure 7 – Refactoring *Move and Rename Class*



Source: Prepared by the author.

Figure 6 shows the *Move Class* refactoring that is used to move a class to a more appropriate location. The procedure consists of moving a class to a package more related to the code, removing files generated during compilation in the old location, and modifying the necessary references. We can perform this refactoring when: (i) we identify that the class is next to classes that are not significantly related to its form or function; (ii) the package of the class in question is too large; and (iii) we want to refactor to avoid future dependency problems (Fowler, 2018). Even though it is a little simpler than the refactorings mentioned above, it is also considered Non-Trivial because it can make a larger change to the code design. This refactoring can use other refactorings cataloged by Fowler (2018) such as: *Extract Package*. Figure 7 shows a variation of this refactoring in which it is necessary to rename the original class.

2.1.1.2 Trivial

Trivial refactorings are easy to identify because they change little to the design of the code. The following are trivial refactorings: *Add Class Annotation*, *Add Class Modifier*, *Change Access Modifier*, *Modify Class Annotation*, *Remove Class Annotation*, *Remove Class Modifier*, *Rename Class*, *Rename Method*, and *Rename Variable*. Table 2 describes each trivial refactoring.

Table 2 – Group of trivial refactorings used in the first study

Refactoring	Description
Add Class Annotation	Used when it is necessary to add an annotation to a class.
Add Class Modifier	Adds a modifier (final, static or abstract) to the class.
Change Access Modifier	Changes the access modifier to default, private, protected, or public.
Modify Class Annotation	Change the class annotation.
Remove Class Annotation	Removes a class annotation.
Remove Class Modifier	Removes one of the modifiers: final, static or abstract.
Rename Class	Changes the class name.
Rename Method	Changes the method name.
Rename Variable	Changes the variable name.

Source: Prepared by the author.

2.1.2 Refactoring detection tools

In recent years, different techniques and tools for automatic refactoring detection have emerged in the literature. All of these solutions have their advantages and disadvantages. They are generally based on rules or machine learning. Rule-based solutions have better results but require a higher level of knowledge to define the rules, which is not a trivial task. Machine learning-based solutions, on the other hand, require a large amount of data as input to train classification algorithms.

Several automatic tools currently perform the task of detecting and applying refactorings. Lacerda *et al.* (2020) presents in its study the tools most used by developers that support refactorings. The most used tools are presented below:

– **JDeodorant**[2]: is an Eclipse plugin that suggests Java code refactorings for some code smells: God Class, Large Class, Feature Envy, Swwitch Statement/Type Check and Long Method.
– **TrueRefactor**[3]: is an automated refactoring tool that significantly improves the understandability of legacy systems.
– **Eclipse Refactoring**: A feature of the IDE itself, which has more than 20 refactorings.
– **IntelliJ IDEA Refactoring**: The IDE itself implements more than 40 refactorings, using a lexical analyzer.
– **Wrangler**[4]: It is a tool that supports interactive refactoring of Erlang programs. It is integrated with Emacs and also with Eclipse, through the ErlIDE plugin.

In addition to the previously mentioned tools, state-of-the-art tools detect refactorings applied to commit history, such as Refactoring Miner (Tsantalis *et al.*, 2020) and ReffDiff (Silva *et al.*, 2021). The first detects refactorings in projects developed in the Java language, while the second, in addition to Java, supports the JavaScript and C programming languages. However, the tool used in this work was Refactoring Miner[5] because it presents more accurate results in detecting refactorings, with an average precision of 99.6% and *recall* of 94% (Tsantalis *et al.*, 2020). In addition, the chosen tool detects low-level refactorings, can be easily implemented in automation projects through its API, has support for the Java language, and can be used as a dependency of Maven projects. Furthermore, Refactoring Miner is, on average, 2.6 times faster than ReffDiff. Refactoring Miner has a Chrome browser extension that allows detecting refactoring in the GitHub URL[6]; it is free and has already been used in several works (AlOmar *et al.*, 2021; Sellitto *et al.*, 2021; Aniche *et al.*, 2020; Alomar *et al.*, 2022; Nyamawe, 2022).

The use of the Refactoring Miner tool is also important, as it helps detect refactorings that have been applied even if they have not been documented. At the time of our study, this

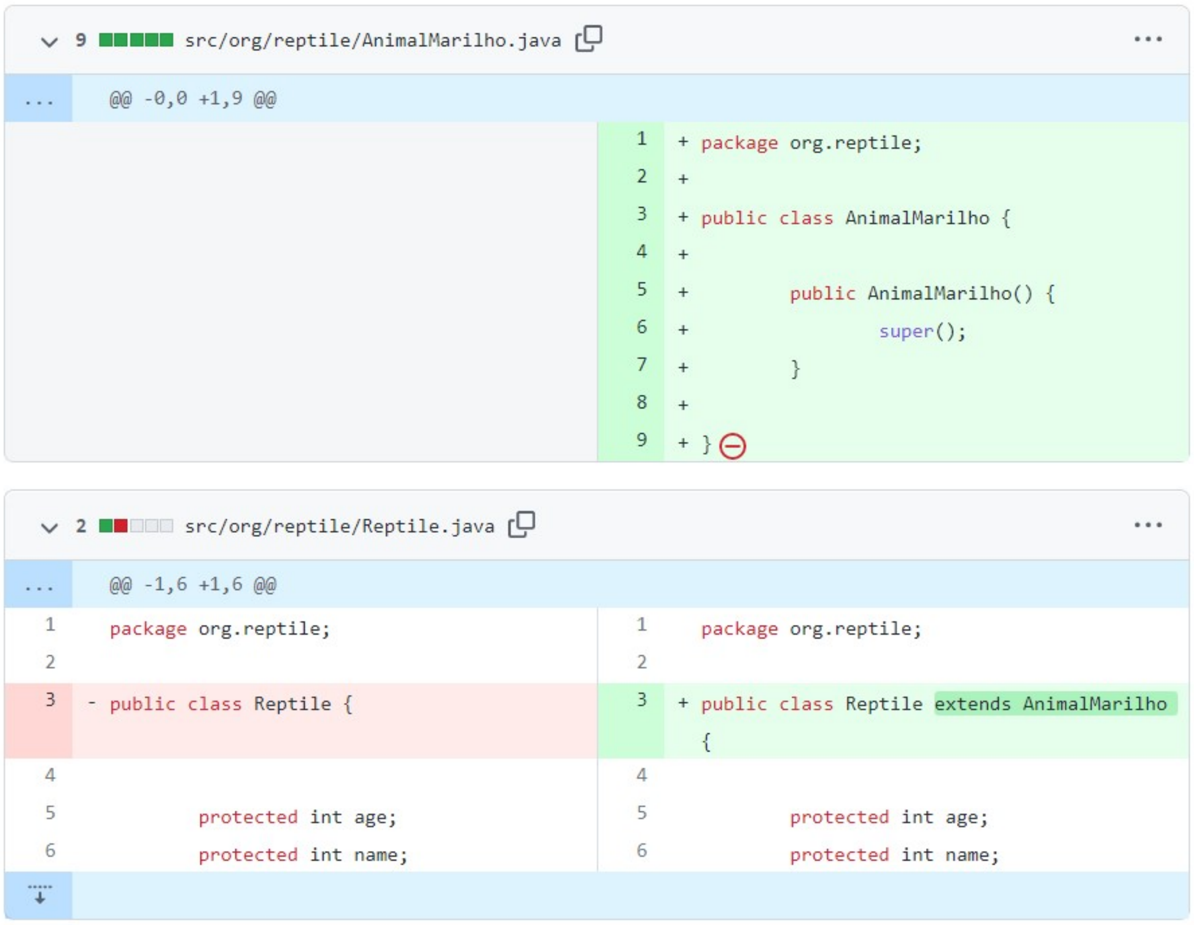
[2] https://github.com/tsantalis/JDeodorant
[3] https://github.com/ramon1/TrueRefactor
[4] http://refactoringtools.github.io/wrangler/overviesummary.html
[5] https://github.com/tsantalis/RefactoringMiner
[6] https://github.com/user/project/commit/id

tool could detect 87 different types of refactorings: *Extract Method, Inline Method, Rename Method, Move Method, Move Attribute, Pull Up Method, Pull Up Attribute, Push Down Method, Push Down Attribute, Extract Superclass, Extract Interface, Move Class, Rename Class, Extract and Move Method, Rename Package Change Package (Move, Rename, Split, Merge), Move and Rename Class, Extract Class, Extract Subclass, Extract Variable, Inline Variable, Parameterize Variable, Rename Variable, Rename Parameter, Rename Attribute, Move and Rename Attribute, Replace Variable with Attribute, Replace Attribute (with Attribute), Merge Variable, Merge Parameter, Merge Attribute, Split Variable, Split Parameter, Split Attribute, Change Variable Type, Change Parameter Type, Change Return Type, Change Attribute Type, Extract Attribute, Move and Rename Method, Move and Inline Method, Add Method Annotation, Remove Method Annotation, Modify Method Annotation, Add Attribute Annotation, Remove Attribute Annotation, Modify Attribute Annotation, Add Class Annotation, Remove Class Annotation, Modify Class Annotation, Add Parameter Annotation, Remove Parameter Annotation, Modify Parameter Annotation, Add Variable Annotation, Remove Variable Annotation, Modify Variable Annotation, Add Parameter, Remove Parameter, Reorder Parameter, Add Thrown Exception Type, Remove Thrown Exception Type, Change Thrown Exception Type, Change Method Access Modifier, Change Attribute Access Modifier, Encapsulate Attribute, Parameterize Attribute, Replace Attribute with Variable, Add Method Modifier (final, static, abstract, synchronized), Remove Method Modifier (final, static, abstract, synchronized), Add Attribute Modifier (final, static, transient, volatile), Remove Attribute Modifier (final, static, transient, volatile), Add Variable Modifier (final), Add Parameter Modifier (final), Remove Variable Modifier (final), Remove Parameter Modifier (final), Change Class Access Modifier, Add Class Modifier (final, static, abstract), Remove Class Modifier (final, static, abstract), Move Package, Split Package, Merge Package, Localize Parameter, Change Type Declaration Kind (class, interface, enum), Collapse Hierarchy, Replace Loop with Pipeline, Replace Anonymous with Lambda, Merge Class, Inline Attribute.*

The tool receives two commits as input, analyzes the *diff* of the files, and returns a list of refactorings applied between the commits. In Figure 8, we can see the *diff* of a commit that is analyzed by the tool. In this figure, it is possible to see that the Reptile class previously had no inheritance and then had AnimalMarilho as a superclass. Then, the AnimalMarilho class is created and implemented. In the example, the tool can analyze and detect the *ExtractClass* refactoring.

Finally, this work uses the Refactoring Miner tool to extract the refactorings from

Figure 8 – Commit diff before and after refactoring



9 src/org/reptile/AnimalMarilho.java

```
@@ -0,0 +1,9 @@
+ package org.reptile;
+
+ public class AnimalMarilho {
+
+         public AnimalMarilho() {
+                 super();
+         }
+
+ }
```

2 src/org/reptile/Reptile.java

```
@@ -1,6 +1,6 @@
  package org.reptile;

- public class Reptile {
+ public class Reptile extends AnimalMarilho {

          protected int age;
          protected int name;
```

Source: Prepared by the author.

the commit history, serving as a basis for training the supervised learning models.

2.2 Code quality metrics

Measuring code quality may not be an easy task. ISO/IEC 25010 provides a model that standardizes some external software characteristics by which we can evaluate quality (INTERNATIONAL ELECTROTECHNICAL COMMISSION, 2011). These characteristics are also known as quality attributes. Among these attributes, we have reliability, which measures the probability of the system failing. Another characteristic is maintainability, which measures the difficulty of changing the system over time. With this, it is clear that several abstract factors are requirements for evaluating the quality of a system.

However, we can analyze software through software metrics that will measure some internal attributes, such as coupling. This attribute indicates the degree of interdependence between system classes, which can affect an external attribute (maintainability). There are other internal quality attributes such as cohesion, inheritance, size, and complexity. Measuring from this internal perspective becomes easier by using values that can be measured using tools that statistically analyze source code (INTERNATIONAL ELECTROTECHNICAL COMMISSION, 2011).

Software metrics are used to measure and understand the structure of software. Once extracted, they provide meaningful information to evaluate internal quality attributes. Software metrics have been used and studied for a long time. Lines of Code (LOC) was one of the first metrics, first cited in the late 1960s, and is the most common for measuring software size to estimate time and cost (Lorenz; Kidd, 1994). Chidamber and Kemerer (1994) proposed a suite of code metrics for software that adopts the object-oriented paradigm. Chidamber and Kemerer (1994) being one of the precursors, several works have been carried out using this suite (Li; Henry, 1993; Padhy *et al.*, 2015; Aggarwal *et al.*, 2006; Malhotra[1]; Chug, 2012). McCabe (1976) studied and provided metrics based on cyclomatic complexity.

For this Master's dissertation, we selected a set of metrics from the suite proposed by Chidamber and Kemerer (1994), and Lorenz and Kidd (1994) in addition to different attributes of the code elements, such as: number of methods, number of returns and number of variables. We extracted the metrics from the CK (Aniche, 2015) and PMD[7] tools to use as features of the prediction models. The metrics and attributes used in each Master's dissertation study are found in Tables 7 and 15, respectively.

2.3 Machine learning

Currently, companies and public entities use the internet to offer various services to customers and users. These services generate a massive amount of data, making it important to study them to obtain some information or knowledge. For example, streaming platforms that capture user preferences through choices registered in the system. When the amount of data is small, it is possible to perform analysis manually. However, when there is a large mass of data, this becomes a difficult task for human capacity. Therefore, researchers seek computational solutions with greater computational power to solve this problem by processing and extracting information from this data.

An example of a computational solution used for this purpose is Machine Learning (ML) (Mitchell; Mitchell, 1997; Mitchell *et al.*, 2013). This solution consists of a field of artificial

[7] https://pmd.github.io/

intelligence that focuses on developing algorithms and statistical models that allow computers to learn from data and make predictions or decisions without being explicitly programmed (Samuel, 1959; Zhou, 2021). It can also be understood as the ability of computers to process data input and predict outputs through computational algorithms (Mitchell; Mitchell, 1997). For example, when entering user data from the streaming platform, the machine learning algorithm will learn and then return a list of suggestions related to the user's profile. Therefore, it is necessary to provide a set of data for the algorithm to train so that it can then predict.

ML is a constantly evolving area and is considered a sub-area of Artificial Intelligence (AI) (Mitchell *et al.*, 2013). It provides several types of algorithms to solve problems; however, there is no single algorithm to solve all types of problems. Thus, the type of algorithm will depend on the type of problem added to other factors. There are mainly four main types of learning algorithms: (i) supervised learning, (ii) unsupervised, (iii) semi-supervised, and (iv) reinforcement learning (Mahesh, 2020; Kaelbling *et al.*, 1996).

In the field of ML, supervised learning is a fundamental technique where a model is trained using a labeled dataset, allowing it to learn to map inputs to outputs based on previous examples (Mitchell; Mitchell, 1997; Bishop; Nasrabadi, 2006; Cord; Cunningham, 2008; Zhou, 2021; James *et al.*, 2023). In this approach, the correct output value is already known, and the dataset can be divided into two subsets: training and testing. The testing subset validates the model that must have been trained on the other subset. This technique is widely used in various applications such as pattern recognition, forecasting, and classification (Jordan; Mitchell, 2015; James *et al.*, 2023). Supervised learning can be further divided, depending on the problem, into Classification and Regression (Cord; Cunningham, 2008).

Classification is used when the problem wants to provide a categorical or binary variable as output. At the same time, Regression seeks to predict a continuous variable as output (Mitchell *et al.*, 2013), usually a numerical range (James *et al.*, 2023). It aims to predict continuous results by fitting a curve or line to the data points, allowing the estimation of the dependent variable based on the independent variables(James *et al.*, 2023). For example, we can use ML to predict the price of homes. To do this, we can provide a dataset with data about several homes, such as: price, location, size, number of floors, number of bedrooms, number of bathrooms, etc. Since we want to predict the price and it is a continuous variable, we have a problem that Regression algorithms will be more suitable to solve. Another example would be trying to predict the type of car based on a set of data, such as: color, number of wheels, size, etc.

In this case, the problem should be solved through classification algorithms.

In contrast, unsupervised learning does not use input-output pairs. However, the data is related to each other to form clusters. The algorithm based on this approach will be able to recognize similar characteristics between the data and separate them into groups, specifying what these groups are (Berry *et al.*, 2019). Semi-supervised learning is a combination of supervised and unsupervised learning. This type of approach allows the analysis performed on the dataset to accept labeled or unlabeled data.

Finally, reinforcement learning is an approach in which the algorithm is trained based on reinforcement. Reinforcement consists of rewards when the model gets it right and penalties when it gets it wrong. It is expected that after the results obtained in training, the model will generate better results than when it started (Kaelbling *et al.*, 1996). For this Master's dissertation, we chose to use supervised learning. In the first study, we focused our efforts on solving a classification problem, while in the second study, we addressed a regression problem.

2.3.1 Supervised learning algorithms

2.3.1.1 Classification

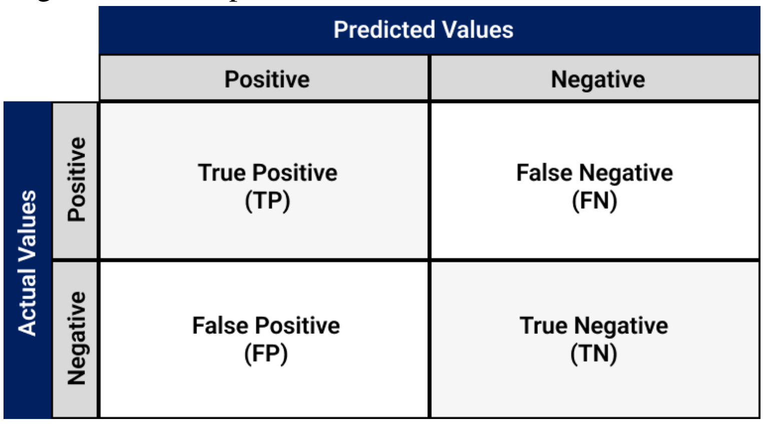
There are several ML techniques for the classification problem. However, for our first study, we used only five of them: (i) Decision Tree, (ii) Logistic Regression, (iii) Naive Bayes, (iv) Neural Network, and (v) Random Forest.

– ***Decision Tree***: A technique that builds the classification model as a tree structure. It uses if-then rules that are equally exhaustive and mutually exclusive. These rules are learned sequentially. Some advantages of this technique are: (i) simplicity in visualizing the decision process; (ii) requires little data preparation; (iii) handles categorical and numerical data simultaneously; (iv) speed in making predictions; (v) addresses most problems; and (vi) after training, the cost of maintaining the model is low. However, sometimes the algorithm can create complex, inefficient or unstable trees (Quinlan, 1993).

– ***Logistic Regression***: A technique that uses one or more independent variables to determine a result. To do so, statistical and probabilistic concepts are used to perform a binary classification. Some of its advantages include: (i) the ability to quantitatively explain the factors that lead to a classification; and (ii) understanding how a set of independent variables can affect a result. However, it is limited to binary variables, its biggest disadvantage. It can be divided into three models: binominal logistic regression, ordinal logistic regression, and multinomial logistic regression.

- ***Naive Bayes***: A technique based on Bayes' theorem that assumes independence between features. Therefore, regardless of whether one feature depends on another, all will contribute to the probability of classification independently (Jordan; Mitchell, 2015). It stands out in some aspects, such as: (i) it is simple to build; (ii) it is useful for large data sets; (iii) it requires a small amount of training data; and (iv) it is quick to make predictions. However, its accuracy is low compared to other classifiers.
- ***Neural Network***: It is a computing system involving neurons organized in interconnected layers. Each neuron receives input information, applies a function, usually non-linear, and generates an output sent to another layer. It is worth mentioning that its advantages include: (i) the ability to solve an infinite number of problems; (ii) tolerance to noisy data; (iii) classifying untrained patterns; and (iv) being used for deep learning. However, it requires high computational power, making it difficult to explain the learning process performed (Jin *et al.*, 2000).
- ***Random Forest***: A technique that can create several independent decision trees in which samples of observations and variables are assigned. It adds the predictions of each tree to determine an overall prediction of the forest. Its advantages include: (i) it can be used for both classification and regression; (ii) it tends to have high accuracy rates; and (iii) it can perform overfitting to improve the result; however, if it becomes too dense it may take a long time to predict a value (Cutler *et al.*, 2012).

Figure 9 – Example of confusion matrix



		Predicted Values	
		Positive	Negative
Actual Values	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

Source: Adapted from (Stehman, 1997).

To evaluate the performance of ML algorithms, it is necessary to apply metrics to

Table 3 – Metrics for evaluating supervised learning models for classification problems

ML Metrics	Equation
Accuracy	$\frac{TP+TN}{TP+FP+TN+FN}$
Precision	$\frac{TP}{TP+FP}$
Recall	$\frac{TP}{TP+FN}$
F1-score	$2\times\frac{Precision\times recall}{Precision+recall}$
AUC	$\int_0^1 \mathrm{TPR}(\mathrm{FPR})^{-1}\mathrm{dFPR}$

Source: Prepared by the author

measure the quality of a model. The metrics use the values extracted from the confusion matrix (Figure 9): True Positive (TP), False Positive (FP), True Negative (TN) and False Negative (FN). For this Master's dissertation, we used the following metrics: (i) Accuracy, (ii) Precision, (iii) Recall, (iv) F1-score and (v) AUC metric.

- **Accuracy**: This is the proportion of correctly classified observations among the total number of observations. Accuracy indicates the overall performance of the model (Davis; Goadrich, 2006). Table 3 presents its calculation equation.
- **Precision**: This is the proportion of correctly classified positive observations among the predicted positive observations. Precision is measured when FPs are considered more harmful than FNs. (Carvalho *et al.*, 2019). Table 3 presents its calculation equation.
- ***Recall***: This is the proportion of correctly classified positive observations among the true positive observations. *recall* is measured when FNs are considered more harmful than FPs (Carvalho *et al.*, 2019). Table 3 presents its calculation equation.
- **F1-Score**: It is the harmonic mean between precision and *recall* (Chicco; Jurman, 2020). Table 3 presents its calculation equation.
- **AUC**: Area Under the Curve is a commonly used metric to evaluate the quality of a binary classification model. Represents the area under the ROC (Receiver Operating Characteristic) curve, which is a graph that shows the relationship between the True Positive Rate (TPR) and the False Positive Rate (FPR). Is an indicator of the model's ability to distinguish between positive and negative classes (Hanley; McNeil, 1982; MuschelliIII, 2020).

2.3.1.2 Regression

Similarly, there are several ML techniques for the regression problem. However, for our first study we used only seven of them: (i) Linear Regression, (ii) Ridge Regression, (iii) Elastic Net, (iv) Decision Tree, (v) Random Forest, (vi) Gradient Boosting, and (vii) XGBoost.

– **Linear Regression:** Linear Regression is a statistical method used to model the relationship between a continuous dependent variable and one or more independent variables, assuming that a straight line can describe this relationship. The model is estimated by minimizing the sum of the squared errors, which ensures that the fitted line is the best possible representation of the observed data. The simplicity and interpretability of Linear Regression make it a widely used tool in several areas (Rawlings *et al.*, 1998; Su *et al.*, 2012).

– **Ridge Regression:** Ridge Regression is an extension of Linear Regression that includes a penalty called L2, that is, it adds a regularization term proportional to the square of the magnitude of the coefficients. This approach is particularly useful when there is multicollinearity between the independent variables since the penalty reduces the variance of the estimators, making the model more robust. Ridge regression is commonly used in situations where the number of predictors is large relative to the number of observations, which can cause overfitting (McDonald, 2009)

– **Elastic Net:** Elastic Net combines the L1 used in Lasso Regression (Ranstam; Cook, 2018) and L2 used in Ridge Regression(McDonald, 2009) penalties, offering a flexible approach to variable selection and regularization. This method is particularly effective in situations where there is a high correlation between predictors, as it can select groups of correlated variables instead of choosing a single variable. Elastic Net is widely used in high-dimensional applications, such as computational biology and genomic data, where the ability to handle many predictors is crucial (Zou; Hastie, 2005).

– **Decision Tree:** Decision Tree is a supervised learning algorithm that can be used for both regression and classification. It segments the input space into non-linear regions, creating a decision tree based on simple split conditions at each node. The main advantage of decision trees is their interpretability, as the decisions made at each node can be easily visualized and understood. However, decision trees tend to suffer from overfitting, especially in problems with many predictors (Breiman, 2017).

– **Random Forest:** Random Forest is an ensemble method that combines several decision trees to improve the accuracy and robustness of the model. It uses the bagging technique (bootstrap aggregating) to train several trees on different subsets of the training data, and the final predictions are made by averaging, in the case of regression, the predictions of all trees. This approach reduces the variance of the model and improves its generalization, making Random Forest a popular choice for many regression problems (Breiman, 2001).

– **Gradient Boosting:** Gradient Boosting is another ensemble method that builds the regression model sequentially by fitting new models to correct the residual errors of the previous models. Each new model is trained to minimize the loss function using gradient descent, which allows the algorithm to improve the model's performance iteratively. Gradient Boosting is known for its high accuracy but can also be susceptible to overfitting if not regularized properly (Friedman, 2001).
– **XGBoost:** XGBoost (Extreme Gradient Boosting) is an optimized version of the Gradient Boosting algorithm that incorporates several improvements, such as L1 and L2 regularization, the use of parallelism in tree construction, and pruning techniques to avoid overfitting. It is widely used in machine learning competitions and is recognized for its high performance on a wide range of regression problems. XGBoost's efficiency and fine-tuning capabilities make it a powerful tool for predictive analysis on large datasets (Chen; Guestrin, 2016).

To evaluate the performance of ML algorithms, we can use metrics to measure the quality of a model. A brief description of each metric is provided below, and Table 4 presents the equations used for their calculation:

Table 4 – Metrics for evaluating supervised learning models for regression problems

ML Metrics	Formula
MSE	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
RMSE	$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$
MAE	$\frac{1}{n}\sum_{i=1}^{n}\lvert y_i - \hat{y}_i\rvert$
MAPE	$\frac{100\%}{n}\sum_{i=1}^{n}\left\lvert\frac{y_i-\hat{y}_i}{y_i}\right\rvert$
R^2	$1 - \frac{\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{\sum_{i=1}^{n}(y_i-\bar{y})^2}$
R_A^2	$1 - \frac{(1-R^2)(n-1)}{n-p-1}$

Source: Prepared by the author

– **Mean Squared Error (MSE):** is a metric that calculates the mean of the squares of the errors, where the error is the difference between the real value and the value predicted by the model. We should consider lower MSE values as more accurate models to interpret this metric. Since the errors are squared, the metric gives greater weight to outliers (Montgomery *et al.*, 2021). The equation used to calculate MSE is:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Breakdown:

- n: Total number of observations or data points.
- y_i: Actual value of the i-th data point.
- $\hat{y}_i$: Predicted value of the i-th data point by the model.
- $(y_i - \hat{y}_i)^2$: Squared difference between the actual and predicted values for each data point.

– **Root Mean Squared Error (RMSE):** is the metric that calculates the square root of the MSE. The interpretation of this metric is similar to the MSE, it offers a more direct interpretation of the errors, since it is expressed in the same unit as the dependent variable (Chai; Draxler, 2014). The equation used to calculate RMSE is:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Breakdown:

- The only difference in interpreting the equation in relation to MSE is the application of the square root.

– **Mean Absolute Error (MAE):** is the metric that calculates the average of the absolute differences between the actual and predicted values without squaring the errors. It measures the average magnitude of the prediction errors, being less influenced by outliers compared to the MSE (Chai; Draxler, 2014). The equation used to calculate MAE is:

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Breakdown:

- $|y_i - \hat{y}_i|$: Absolute difference between the actual value and the predicted value for each data point.

– **Mean Absolute Percentage Error (MAPE):** is the MAE metric expressed as a percentage of the actual value. It helps understand the error in percentage terms but can be misleading if there are actual values very close to zero (Kim; Kim, 2016). The equation used to calculate MAPE is:

$$\frac{100\%}{n}\sum_{i=1}^{n}\left|\frac{y_i-\hat{y}_i}{y_i}\right|$$

Breakdown:

- $\frac{100\%}{n}$: Measures the average difference in percentage terms, expressed as a percentage.
- $\left|\frac{y_i-\hat{y}_i}{y_i}\right|$: Absolute percentage difference between the actual value and the predicted value for each data point.

– **Coefficient of Determination (R²):** is the metric that calculates the proportion of variability in the data explained by the regression model. In interpreting this metric, 1 indicates that the model explains 100% of the variation in the data, while 0 indicates that the model explains no variation. Although rare and usually indicating a problem, negative values can occur when the model fits worse than the mean of the values. This suggests a very poor fit (Montgomery *et al.*, 2021). The equation used to calculate R² is:

$$1-\frac{\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{\sum_{i=1}^{n}(y_i-\bar{y})^2}$$

Breakdown:

- $\bar{y}$: Mean of the actual values
- $\sum_{i=1}^{n}(y_i-\hat{y}_i)^2$: Sum of squared errors (Sum of Squared Residuals).
- $\sum_{i=1}^{n}(y_i-\bar{y})^2$: Sum of squared differences between actual values and the mean (Total Sum of Squares).

– **Adjusted Coefficient of Determination (R² Adjusted):** is a modified version of R² that considers the number of predictors in the model and the sample size. The difference is that it penalizes models that include unnecessary predictors. This metric is useful for comparing models with different numbers of predictors (Srivastava *et al.*, 1995). The equation used to calculate R_A^2 is:

$$1-\frac{(1-R^2)(n-1)}{n-p-1}$$

Breakdown:

- n: Total number of observations or data points.
- p: Number of independent variables in the model.
- R^2: Coefficient of determination.

2.4 Conclusion

This chapter presented the main concepts that will be used in this Master's dissertation. The main objective of this work is to explore and improve the software refactoring technique with a focus on the triviality of operations. We identify and classify refactorings as trivial or non-trivial, using machine learning algorithms to increase software quality and facilitate maintenance. In addition, we propose the creation of an index that measures the triviality of refactorings, evaluating them based on simplicity, speed and risk. Thus, we present the concepts related to the refactorings used to compose the refactoring triviality index Section 2.1 and their categories in Section 2.1.1. We also present three tools used in the study for feature detection and extraction (Section 2.1.2). Finally, we describe the machine learning algorithms used to train the models that will serve as the basis for predicting this triviality in Section 2.3.

3 RELATED WORK

This chapter describes the related work found in the literature during the development of this Master's dissertation. For better understanding, we grouped the related works into three topics: (i) Literature review on refactoring (Section 3.1); (ii) Grouping refactorings into distinct opportunities (Section 3.2); and, (iii) Use of ML techniques as an automated solution for refactoring detection (Section 3.3). In Section 3.4, a comparison of the works related to the proposal of this work is made. Finally, Section 3.5 presents the chapter's conclusions.

3.1 Literature reviews on refactoring

In recent years, several researchers have made efforts to study aspects related to code design refactoring. However, several studies address the subject in isolation and others in conjunction with *code smells*. In this chapter, we present some studies focused on code design refactoring, not limited to code smells refactoring (Singh; Kaur, 2018; Kaur; Singh, 2019; Azeem *et al.*, 2019; Baqais; Alshayeb, 2020; Lacerda *et al.*, 2020; Agnihotri; Chug, 2020; Naik *et al.*, 2023).

Singh and Kaur (2018) conducted a systematic literature review covering 238 primary studies. The authors sought to answer questions related to the current status of code smell refactoring, the approaches used for code smell detection and removal, the tools used to remove anti-patterns, the datasets used by the authors, and the main smells found by researchers. As a result of this study, the authors point out that: (i) after introduced the term refactoring (Opdyke, 1992) and detailed by (Fowler, 2018), few studies were revealed during the following decade, returning to being the object of study from 2001 onwards; (ii) semi-automatic and automatic approaches using code metrics are the most used by the authors, appearing to be easier and more suitable for measuring the internal quality of the system; (iii) the tools can be automatic, with JDeodorant standing out, which is an Eclipe plug-in, has an interface and is easy to find, but has limitations in the number of detectable smells; (iv) open source projects are the most used in data sets, including: Xerces, JFreeChart, ArgoUML, GanttProject, Eclipse, Jedit, Azureus and Log4j; and (v) the most frequently found smells are *God Class* and *Feature Envy*, while the least frequently found smell in the studies was *Parent Bequest*. Based on the results, the authors suggest that further studies be carried out to enable the development of automatic solutions, expanding the refactoring operations involved and optimizing datasets to involve more industrial data.

Kaur and Singh (2019) systematically mapped 142 primary studies in 2 decades, covering studies up to December 2017. The authors sought to investigate issues related to refactoring operations' effect on software quality. Among these issues, refactoring operations and code smells, software quality measures used to measure the impact of refactoring operations, tools used to predict or evaluate the impact of this activity, and datasets chosen to conduct the empirical study were investigated. The results found indicate that: (i) 154 different refactoring operations were identified, of which 70 are in the 72 activities proposed by (Fowler, 2018), with *Extract Class*, *Extract Method* and *Move Method* being the most used; (ii) the quality attributes found in the studies were product, process and resource, with cohesion, coupling and complexity being the most used; (iii) the metrics found and most used were Lines of Code, Number of Methods, Lack of Method Cohesion, Coupling between Objects; (iv) few studies reported tools to predict or evaluate the effect of refactoring on software quality, which the authors still highlight as an open area; and (v) approximately half of the studies considered only one data set, the other half did not exceed 27. Another result found by the authors is that there will not always be an improvement in all quality attributes with refactoring activities. Based on the results, the authors suggest that more studies be carried out on refactoring activities unexplored by researchers, including the most used in the industry, such as: Field Renaming, Class Renaming, Package Renaming, among others. Another open problem identified is the development of automatic solutions, such as tools aimed at refactoring operations.

Even with an exhaustive search, only one systematic literature review conducted to analyze research using ML as an automated solution was found. Azeem *et al.* (2019) conducted a systematic literature review with 15 studies on the use of machine learning for code smell detection between 2000 and 2017. Several aspects were investigated by the authors, among them: (i) the configuration used in the machine learning approach to code smell detection; (ii) how these approaches were evaluated; (iii) the design of the evaluation strategies; and (iv) and a meta-analysis of the performance of the models proposed in the studies. The results found by the authors suitable for our study indicate that: (i) *God Class* and other code smells were the most found smells; (ii) the CK metric set was the most used in the studies; (iii) Three types of classification were most used: binary, probability-based, and severity-based; (iv) the vast majority of studies used tree-based algorithms *Decision Tree*, with *Random Forest* being the most effective; and (v) the choice of independent variables impacts the model's performance at the time of prediction. Based on the results found in the literature review, the authors highlight the lack of studies investigating how machine learning can be used to detect design problems.

Baqais and Alshayeb (2020) conducted a systematic literature review covering 41 studies to describe the state of the art regarding code refactoring between 1996 and 2015. The authors investigated several aspects, including: the level of application of refactoring; methods used to automate refactoring; performance and validation of each method used in refactoring automation; and, the trend in the use of automatic refactoring. The results show that: (I) most studies apply refactoring at the code level, but are not limited to this level. Refactorings are also applied to other software artifacts such as UML diagrams and process models; (ii) manual validation is less used because it takes more time, with software metrics or quality metrics being preferable to measure software quality; and (iii) code levels are the most addressed, with a higher occurrence for class and method levels. One of the problems identified by the authors is that most correction works did not verify the preservation of behavior after refactoring, still having the precondition as the only means of verification. However, the authors pointed out that few works provide tools such as automated solutions in their results, generating a lack of support and that more work needs to be done aiming to improve the detection and correction process.

Lacerda *et al.* (2020) conducted a tertiary systematic literature review covering 40 studies published between 1992 and 2018. The authors investigated the most studied code smells and refactorings, the most used refactoring tools, and the main techniques. The results found by the authors show that: (i) *Duplicated Code* and *God Class* were the most cited and most referenced smells in the studies; (ii) the most cited refactoring operations were those involving extraction (class, method, and variable); and (iii) CCFinder and JDeodorant were the most used tools for detecting code smell and refactoring, respectively. Furthermore, the authors identified that refactorings affect quality attributes more than code smells, indicating that it is not just a code smell-refactoring relationship but rather that the relationship's origin is quality.

Agnihotri and Chug (2020) conducted a literature review to investigate studies based on applying refactoring operations to remove code smells and their effect on software quality. The authors selected 68 studies that were published between 2001 and 2019. The results found by the authors indicate that: (i) *Extract Class* is among the most used techniques in the selected studies; (ii) refactoring is mainly performed automatically, with the help of tools; (iii) tools such as SourceMeter, JDeodorant were the most used in the study; and (iv) complexity, cohesion and size metrics were the most used. Thus, based on the results, the authors pointed out the need for researchers to conduct studies proposing automatic tools for detecting and applying refactorings to improve software quality.

Naik *et al.* (2023) conducted a detailed review on the use of deep learning (DL) techniques in code refactoring, analyzing 17 studies published between 2016 and 2022. The objective was to discover which DL techniques are applied in refactoring, evaluate the performance of these approaches, and identify the limitations and challenges faced in the area. The findings revealed that the Java language predominated in the research and that variable-level refactoring presented the best performance. At the same time, neural networks based on Multilayer Perceptrons (MLP) led in effectiveness, unlike Convolutional Neural Networks (CNN), which had the worst performance. In addition, method-level refactoring obtained inferior results, while class-level refactoring was slightly more promising. Finally, the authors highlight that, despite significant advances, important gaps persist, especially in the exploration of post-refactoring themes, in the expansion to other programming languages, and in the availability of robust models and datasets for new investigations.

Based on the results found in the literature review on refactoring, it is possible to identify some deficiencies and needs for this area, such as: (i) conducting studies that develop new automated solutions involving refactoring activities; (ii) conducting more studies with refactoring activities commonly used in the industry; and (iii) conducting studies addressing machine learning as a solution to code design problems. Therefore, and knowing the importance of filling these gaps, this work proposes to explore a semi-automated solution to classify the refactorings most used by developers, covering those involved in the industry and based on models trained by supervised learning algorithms.

3.2 Grouping refactorings into different opportunities

Eposhi *et al.* (2019) conducted a study that investigated 1,468 classes from 2 systems implemented in C# to investigate the impact of refactoring activity on design problems, density, and symptom diversity. For the authors, the grouping work was based on refactorings performed and not performed. However, the types of refactorings used were not specified. The results indicate that the refactored classes have a higher density of code smell, coupling, and complexity when compared to other classes in the system. Another result identified is that the refactorings did not positively impact the density and diversity of any type of code smell.

Peruma *et al.* (2020) conducted a study on 800 Java projects to identify the influence that data type changes have on the structure and meaning of a renaming refactoring. Given this, the authors did not perform any grouping of the refactorings. The results found by the authors indicate that some developers with little experience in projects tend to perform only renaming refactorings rather than other types of refactorings. Thus, the authors seek to offer improvements to the support for renaming recommendations.

Fernandes *et al.* (2020) conducted a study on 23 projects with 29,303 refactoring operations, of which half were re-refactoring operations, to investigate how refactoring and re-refactoring operations affect internal quality attributes such as: cohesion, complexity, coupling, inheritance, and size. The authors grouped the refactorings according to their granularity, which can be: class, method, and variable. They combined descriptive analysis and statistical tests to deeply understand the effect of refactoring and re-refactoring on each attribute. The results show that 90% of the refactoring operations and 100% of the re-refactorings were applied to code elements with at least one critical attribute. Another result identified is that 65% of the operations improved the attributes associated with the type of refactoring applied, while 35% remained unchanged.

Sellitto *et al.* (2021) conducted a study to investigate the impact of refactoring on developers' understanding of source code. The authors mined refactorings from 156 software projects and related them to readability metrics to verify the positive and negative impacts. The refactorings were grouped based on their general purpose, such as: Composing Methods, Moving Resources, Organizing Data, Simplifying Method Calls, Dealing with Generalizations, and Others. In addition, the authors provided initial information on the relationship between refactoring and program understanding through statistical models that could generalize logistic regression to multi-class problems. However, the authors point out that refactoring will not always have positive impacts, and they also point out some cases of negative impact on code understanding.

Smiari *et al.* (2022) conducted a study on a single case of a company that develops embedded applications in the medical field to investigate how refactorings are applied. The refactorings were also separated according to their general purpose, such as: Composing Methods, Simplifying Conditionals, Moving Methods, Simplifying Method Calls, Optimizing Data, Dealing with Generalizations, and Large Refactorings. In addition, the results found by the authors indicated that quality attributes such as Maintainability and Reusability motivate the refactoring activity. Another result identified was the most frequent refactorings, such as:

Extract Method; Replace Magic Number with Constant; and *Remove Parameter* with the first one presenting 80% applicability.

Ferreira *et al.* (2023), proposed a definition to order dependencies between refactorings and developed an algorithm to detect these relationships, using a vast dataset with 1,457,873 refactorings recommended in 9,595 open source projects. The authors classified refactorings into trivial and non-trivial graphs, representing collections of operations with their respective application dependencies. They investigated the accuracy of detecting these dependencies and compared how these refactoring graphs impacted code quality attributes. The results showed a 100% correct detection rate, revealing that refactorings are often involved in dependent relationships and can rarely be applied in isolation. Thus, the authors conclude that reasoning about refactorings should be collective, considering the mutual impact between operations, and not individually.

The studies present different approaches to grouping refactorings, examining their impacts on code quality and software comprehension. For example, the study by Eposhi *et al.* (2019) makes a simple grouping between classes that underwent refactorings and classes that did not, without specifying the types of refactorings applied. On the other hand, the studies by Sellitto *et al.* (2021) and Smiari *et al.* (2022) share an approach of grouping refactorings based on their general objectives, such as "Method Composition" or "Call Simplification", and both investigate the impact of these refactorings on attributes such as readability and code quality. Peruma *et al.* (2020) and Fernandes *et al.* (2020) differ in their approaches: Peruma *et al.* (2020) focuses on renaming refactorings without performing formal groupings, while Fernandes *et al.* (2020) groups refactorings by granularity (class, method, variable), allowing a more detailed analysis of their effects on internal attributes. In a broader perspective, Ferreira *et al.* (2023) addresses the interdependence between refactorings, grouping them into trivial and non-trivial graphs, suggesting that refactorings should be considered together, given their impact on code quality. Thus, the present work presents two groupings and explores their relationship. Then, it seeks to identify, through an index, to which group a given refactoring may belong.

3.3 Using machine learning techniques as an automated solution for refactoring detection

Aniche *et al.* (2020) conducted a large-scale empirical study on 11,149 projects to verify the effectiveness of machine learning algorithms in predicting refactoring recommendations. The authors found that supervised algorithms are effective in predicting refactorings, and the results indicate that the resulting models achieved accuracy above 90

Alomar *et al.* (2022) conducted a study to analyze the relationship between documentation and refactoring operations by comparing two different approaches. In this way, the authors performed text mining of commit messages, extracting keywords that best represent the type of refactoring. Subsequently, models that approach a *multi-class* classification were trained to predict the types of refactoring. The results indicate that each type of refactoring operation has a different complexity for prediction. The *Rename Method* and *Extract Method* operations were considered the best documented, while *Pull-up Method* and *Push-down Method* were the most difficult to identify through textual descriptions. However, they involve a few of the main refactorings in the industry and produce a model capable of predicting refactorings.

Nyamawe (2022) used machine learning with commit history to predict refactorings. The authors implemented a binary classifier to predict the need for refactorings and a multi-label classifier to recommend refactoring. The results suggest that leveraging commit messages significantly improved the accuracy of refactoring recommendations. However, few of the significant refactorings in the industry are considered in the study, and only a binary classifier is produced as the output.

Panigrahi *et al.* (2020) conducted a study in which they proposed *Naive Bayes (Gaussian, Multinomial and Bernoulli)* classifier-based models to predict software refactorings at the method level. In addition, the authors used techniques such as *SMOTE*, *UPSAMPLE* and *RUSBOOTS* for data balancing. The results indicate that, among the *Naive Bayes* classifiers, *Bernaulli* is the one that provides the highest accuracy compared to the others used in the study. However, it only involves refactorings at the industry method level.

AlOmar *et al.* (2021) conducted a study using 800 projects to understand what motivates developers to apply refactoring. The study was based on the commit comments made by developers. To this end, the authors used supervised machine learning with multi-class models defining categories for types of refactorings. The authors identified that developers' motivation to refactor is to correct code smells and errors, change requirements, optimize the design structure, and improve quality attributes. In addition, the authors identified the most commonly used textual patterns in refactorings. However, they provide an automated solution to detect refactorings and investigate in the study few of the main refactorings used in the industry.

Panigrahi *et al.* (2022) conducted a study using 125 software metrics calculated in the design phase from the class diagrams of object-oriented systems, that is, before development. The objective was to create a predictive refactoring model, using machine learning-based frameworks, focusing on ensemble techniques. The authors performed a multi-stage filtering to select the most relevant features, prioritizing attributes that characterize inheritance, size, coupling, cohesion, and complexity. The study used a diverse set of classifiers, such as Decision Tree, Logistic Regression, and Extreme Learning Machine (ELM), among others. To evaluate the accuracy of the models, metrics such as mean absolute error (MAE), mean magnitude of relative error (MMRE), root mean square error (RMSE), and standard error of the mean (SEM) were used. The results indicated that an approach based on a heterogeneous model of classifiers outperformed the performance of individual classifiers. In addition, data balancing techniques, such as random sampling, upsampling, and downsampling, were tested, with upsampling standing out as the most effective strategy.

In the studies carried out by Aniche *et al.* (2020) and AlOmar *et al.* (2021), even considering many refactorings, they still cover few of the main refactorings in the industry. Although they trained several supervised learning models as an automated solution, the authors did not introduce any new concept or solution to measure the refactoring technique to support developers. The studies carried out by Nyamawe (2022), Alomar *et al.* (2022), Panigrahi *et al.* (2022) and Panigrahi *et al.* (2020) use machine learning, but consider a small number of refactorings. In addition, they also do not introduce a new solution to measure refactoring. Thus, this study aims to fill the gaps left by previous works by exploring and improving the measurement of refactoring triviality. The solution will be based on machine learning models capable of detecting the triviality of refactoring (Section 2.1.1).

3.4 Comparison of related work

In the Table 5, we can view the main works related to the present work, considering similar factors expressly columned. For this, some aspects were considered to better understand the different works. Among these factors, we have: (i) Models types, the number of models types used in the study based on machine learning; (ii) Dataset, presents the number of software projects involved in the study; (iii) Industry Refactorings, a factor that corresponds to the number of the main industry refactorings described in Section 2.1.1 that the work covers (integer number if informed and "Not specified" if not informed by the author); (iv) Learning Techniques, presents the techniques used in the study to achieve the objective of the work; (v) Grouping, presents the categories created during the study to achieve the objective of the work (sequence of integers with the name of the category or grouping used in the study, or the expression "Not specified" if no category is informed by the author during the study). An important point noted is that none of the related works involving machine learning consider all the main refactorings in the industry. Furthermore, although literature reviews (Baqais; Alshayeb, 2020; Kaur; Singh, 2019; Lacerda *et al.*, 2020; Agnihotri; Chug, 2020; Naik *et al.*, 2023) have pointed out the need to produce automated solutions capable of improving the refactoring technique, none of the works attempt to measure the triviality of the refactoring technique. Thus, this work proposes to configure a solution based on machine learning that calculates a triviality index that serves as a metric to measure the application of each refactoring operation validated by experts.

Table 5 – Comparison of related work

Work	Models Types	Datasets	Industrial Refactorings	Techniques	Clustering
Author's proposal	12	1,291	10	Supervised Learning	1-Trivial, 2-Non-Trivial, Index 0-1
(Eposhi *et al.*, 2019)	0	2	(Unspecified)	Statistical Testing	1-Refactored, 2-Other
(Fernandes *et al.*, 2020)	0	23	4	Descriptive analysis, Statistical tests	1-Class, 2-Method, 3-Field
(Peruma *et al.*, 2020)	0	800	5	Detection Automatic	(Unspecified)
(Panigrahi *et al.*, 2020)	3	5	1	Supervised Learning	1-Refactored, 2-Unrefactored
(Sellitto *et al.*, 2021)	0	156	10	Statistical Models	1-Composing Methods, 2-Moving Resources, 3-Organizing Data, 4-Simplifying Method Calls, 5-Dealing with Generalizations, and 6-Others
(AlOmar *et al.*, 2021)	6	800	9	Supervised Learning	1-Internal, 2-External, 3-Fix Bug, 4-Fix Smell
(Aniche *et al.*, 2020)	6	11,149	9	Supervised Learning	1-Refactored, 2-Unrefactored
(Alomar *et al.*, 2022)	9	800	4	Supervised Learning	1-Extract, 2-Inline, 3-Move, 4-Pull Up, 5-Push Down, 6-Rename
(Nyamawe, 2022)	6	65	6	Supervised Learning	1-Refactored, 2-Unrefactored
(Smiari *et al.*, 2022)	0	1	4	Search, Interview, Artifact Analysis	1-Composing Methods, 2-Simplifying conditionals, 3-Moving methods, 4-Simplifying Calls, 5-Optimized, 6-Generalizations e 7-Major Refactorings
(Panigrahi *et al.*, 2022)	11	4	(Not specified)	Supervised Learning	1-Refactored, 2-Unrefactored
(Ferreira *et al.*, 2023)	0	9,595	9	Statistical Tests	1-Trivial Refactoring graph, 2-Non-Trivial Refactoring graph

Source: Prepared by the author

3.5 Conclusion

The papers presented in this chapter report the importance of the code refactoring technique, highlighting its positive and negative points and identifying gaps that have not yet been addressed. Most of them focus on specific aspects, such as the effect of refactoring on code smells or software quality metrics. At the same time, few explore more complex scenarios, such as dependencies between refactoring operations. In addition, there is a notable lack of tools capable of evaluating the triviality or importance of refactorings, which would help developers make more informed decisions during the refactoring process. Overall, the papers make significant contributions to understanding the refactoring activity and its impact on software quality, in addition to emphasizing the development of automated solutions to detect and apply these operations. One of the approaches that has received considerable attention from researchers is the use of machine learning to support refactoring. Although this technique has shown great potential, especially for its accurate predictions and the ability to automate parts of the refactoring process, it still requires deeper and more comprehensive studies. The use of machine learning algorithms has been presented in several areas and for various problems. Software refactoring appears to be a promising approach, but it still requires more research to maximize its effectiveness and coverage.

The papers cover a wide range of topics, including literature reviews on code refactoring, clustering of refactoring opportunities, and the use of machine learning techniques for refactoring detection. However, despite the existing contributions, many questions remain open, especially regarding the use of refactoring operations, the impact on software quality attributes, and the lack of reliable automated tools capable of performing refactoring more efficiently and comprehensively. In view of this, this paper proposes a solution for refactoring activity based on machine learning to identify the triviality of refactorings. This approach also includes validation by experts and developers in the area of software engineering, allowing a combination of automatic predictions and specialized human knowledge. Thus, the proposal of this work aims to fill the gaps identified in previous research, offering a solution to improve the application of refactorings and providing predictive models capable of identifying their triviality. Finally, it contributes to a more accurate and effective evaluation of the operations performed, aligning with the industry's needs and trends in software engineering. In the next chapter, we present a study investigating the influence of trivial refactorings on the prediction of non-trivial refactorings using supervised machine learning.

4 ON THE EFFECTIVENESS OF TRIVIAL REFACTORINGS IN PREDICTING NON-TRIVIAL REFACTORINGS

An earlier version of the work in this chapter appears in
the Proceedings of the 16th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS) (Pinheiro *et al.*, 2022) and Journal of Software Engineering Research and Development (JSERD) (Pinheiro *et al.*, 2024).

Refactoring is the process of restructuring source code without changing the external behavior of the software. Refactoring can bring many benefits, such as removing code with poor structural quality, avoiding or reducing technical debt, and improving maintainability, reuse, or code readability. Although there is research on how to predict refactorings, there is still a clear lack of studies that assess the impact of operations considered less complex (trivial) to more complex (non-trivial). In addition, the literature suggests conducting studies that invest in improving automated solutions through detecting and correcting refactoring. This study aims to identify refactoring activity in non-trivial operations through trivial operations accurately. For this, we use classifier models of supervised learning, considering the influence of trivial refactorings and evaluating performance in other data domains. To achieve this goal, we assembled 3 datasets totaling 1,291 open-source projects, extracted approximately 1.9M refactoring operations, collected 45 attributes and code metrics from each file involved in the refactoring and used the algorithms Decision Tree, Random Forest, Logistic Regression, Naive Bayes and Neural Network of supervised learning to investigate the impact of trivial refactorings on the prediction of non-trivial refactorings. For this study, we contextualize the data and call context each experiment configuration in which it combines trivial and non-trivial refactorings. Our results indicate that: (i) Tree-based models such as Random Forest, Decision Tree, and Neural Networks performed very well when trained with code metrics to detect refactoring opportunities. However, only the first two were able to demonstrate good generalization in other data domain contexts of refactoring; (ii) Separating trivial and non-trivial refactorings into different classes resulted in a more efficient model. This approach still resulted in a more efficient model even when tested on different datasets; (iii) Using balancing techniques that increase or decrease samples may not be the best strategy to improve models trained on datasets composed of code metrics and configured according to our study.

4.1 Introduction

During software maintenance, developers can introduce low-quality code intentionally or unintentionally (Ouni *et al.*, 2015; Mello *et al.*, 2022). Over time, this low-quality code can deteriorate the overall code quality and lead to crashes in the future Yamashita and Moonen (2012). Refactoring is a solution that can be used to address this problem by applying transformations to the source code (Silva *et al.*, 2016). Refactoring is a term introduced by Opdyke (1992) but only became widely known after the publication of Martin Fowler's book (Fowler, 2018). Refactoring refers to a transformation that changes the internal structure of the source code without changing its external behavior (Fowler, 2018). In other words, the software should produce the same output after the refactoring activity as it did before.

Researchers have investigated different perspectives for the use of refactoring (Mens; Tourwé, 2004; Azeem *et al.*, 2019; Sobrinho *et al.*, 2018; Bois *et al.*, 2004; Cassell *et al.*, 2011; Bavota *et al.*, 2010; Alkhalid *et al.*, 2011; Dallal, 2012; Bibiano *et al.*, 2023). Among them: (i) solutions that recommend refactorings for developers (Bavota *et al.*, 2015; Tsantalis *et al.*, 2018); (ii) challenges in applying refactoring (Sharma *et al.*, 2015; Kim *et al.*, 2014); (iii) developers' motivation to refactor the code (Silva *et al.*, 2016; Palomba *et al.*, 2017; Paixão *et al.*, 2020); and (iv) machine learning-based refactoring detection (Aniche *et al.*, 2020; Nyamawe, 2022; AlOmar *et al.*, 2021). The utilization of machine learning predictive models (ML) to assist developers in identifying refactoring opportunities to improve design is a relatively new area. (Azeem *et al.*, 2019). Some studies use ML to detect refactoring opportunities through supervised learning (Aniche *et al.*, 2020; AlOmar *et al.*, 2021; Nyamawe, 2022; Alomar *et al.*, 2022; Rish, 2001). Others investigate refactoring opportunities using unsupervised learning (Alkhalid *et al.*, 2010; Bryksin *et al.*, 2018).

Despite many studies investigating how ML can be leveraged as a way to improve refactoring techniques (Alkhalid *et al.*, 2010; Bryksin *et al.*, 2018; Aniche *et al.*, 2020; Nyamawe, 2022; AlOmar *et al.*, 2021; Panigrahi *et al.*, 2020), few studies investigate strategies on how to improve the prediction of refactorings by these models. Kumar *et al.* (2019) states that software metrics are the most important factors in helping to estimate the propensity for refactoring at the class level among the main possible approaches. Azeem *et al.* (2019) conducts a literature review and points out that there is room for studies to investigate how ML can detect refactoring opportunities. In our literature review, we observed that refactoring classification is a widely used strategy. Thus, we decided to incorporate this approach into our research, adopting a classification based on refactoring triviality.

Therefore, our motivation arises from the scarcity of studies on strategies to improve refactoring prediction, especially with the introduction of a methodology not yet explored in the literature, which addresses refactoring triviality. Furthermore, we aim to provide predictive models for automated tools in order to increase their efficiency in predicting refactorings. Automated tools with better refactoring prediction increase developer productivity and help maintain clean, efficient code. In addition, these improvements make the tools more attractive to developers, increasing their market adoption, and allow project managers to more efficiently allocate resources to the most critical areas of the code.

In our previous study (Pinheiro *et al.*, 2022), we investigated the impact of trivial refactorings on classification model prediction. Non-trivial refactorings are operations that generate changes in the design of system, while trivial refactorings do not significantly change the system design. The models were trained using the algorithm: Decision Tree, Random Forest, Logistic Regression, Naive Bayes, and Neural Network in 884 open-source systems. We identified contexts in which trivial refactorings can positively impact the prediction of non-trivial refactorings. We analyzed: (i) the performance of ML algorithms to predict refactorings; (ii) the effect of trivial operations on the prediction of non-trivial ones; and (iii) the use of balancing techniques to improve the predictions.

This article is an extension of our previous study (Pinheiro *et al.*, 2022), in which we investigated the effectiveness of trivial refactorings in predicting non-trivial ones. Furthermore, we used classifier models of supervised learning, taking into account the influence of trivial refactorings. We also evaluated the performance of these models in other datasets. For this study, we: (i) added a new research question (RQ_4) to assess whether the ML models trained with the code metrics and attributes of the dataset used in our previous study (Pinheiro *et al.*, 2022) can generalize to two other datasets selected in this new study; (ii) increased the number of projects used to compose each dataset, totaling 407 new projects (207 from the Apache community and 200 from the Eclipse community) in comparison to the previous study (Pinheiro *et al.*, 2022); (iii) expanded the data extraction process to include refactorings, files, commits, and code metrics, to save all the necessary data for training the machine learning models; (iv) implemented a balancing technique called Synthetic Minority Oversampling Technique (SMOTE), which uses an approach to deal with unbalanced datasets through oversampling of minority classes (Chawla *et al.*, 2002); and, (v) used the Area Under the ROC (AUC) metric in all models of this new study, a widely used metric to measure the classification of ML models (Hanley; McNeil, 1982).

As additional contributions to this article, we claim that ML with tree-based models such as Random Forest and Decision Tree performed extremely well and demonstrated good generalization in other data domains related to refactoring. Additionally, separating trivial and non-trivial refactorings into distinct classes resulted in a more effective model, even when tested on different datasets. However, altering the data balancing technique may lead to a comparable or worse outcome compared to the unbalanced model. This extended version of our study makes the following contributions:

– Our results show that tree-based machine learning models, such as Random Forest and Decision Tree, have shown excellent performance when trained with code metrics to detect refactoring opportunities.
– We identified that separating trivial and non-trivial refactorings into different classes resulted in a more efficient model, suggesting that this approach may improve the accuracy of automated solutions based on ML.
– We observed that sampling balancing techniques might not be the best strategy to improve models trained on datasets composed of code metrics and configured according to the study at hand.
– Finally, we observed that models trained with code attributes and metrics demonstrate good generalization in other data domain contexts.

The remainder of this article is organized as follows. Section 4.2 presents our study settings. Section 4.3 presents our main findings, followed by a discussion. Section 4.4 discusses the main threats to validity. Finally, Section 4.5 concludes the article and suggests future work.

4.2 Study Settings

This section describes the settings of our study. Section 4.2.1 introduces the study goal and research questions. Section 4.2.2 describes each study step and procedure, from data collection to data analysis.

4.2.1 Goal and Research Questions

This study aims to investigate the influence of trivial refactorings at the class level in predicting non-trivial refactorings. To this end, we used models based on ML algorithms trained with 45 code metrics. By understanding how trivial refactorings affect the prediction of non-trivial refactorings, we will be able to discover strategies to improve the prediction of refactorings through supervised learning. Furthermore, we investigated whether the trained models are generalized to other contexts. We describe our research questions (RQ$_s$) as follows.

RQ$_1$: What is the performance of ML algorithms to predict trivial and non-trivial refactorings? – RQ$_1$ aims to investigate the performance of 5 ML algorithms(Random Forest, Decision Tree, Logistic Regression, Naive Bayes and Neural Network) to predict trivial and non-trivial refactorings together. By answering **RQ$_1$**, we can identify which algorithms produce the best results for our different sets of contexts, considering each context as different set that combines trivial and non-trivial refactorings.

RQ$_2$: How effective is the inclusion of trivial refactorings to predict non-trivial refactorings? – RQ$_2$ aims to compare the performance of trained models to predict non-trivial refactorings by considering different sets combining trivial and non-trivial refactorings. By answering **RQ$_2$**, we can compare and evaluate which combination of trivial and non-trivial refactorings presents better results.

RQ$_3$: How effective are data balancing techniques in the prediction of trivial and non-trivial refactorings? – RQ$_3$ aims to evaluate the effectiveness of the data balancing technique applied to our different sets of contexts. By answering **RQ$_3$**, we can identify whether there is an imbalance in our data, as well as find the data balancing technique that performs best in our models with our configuration.

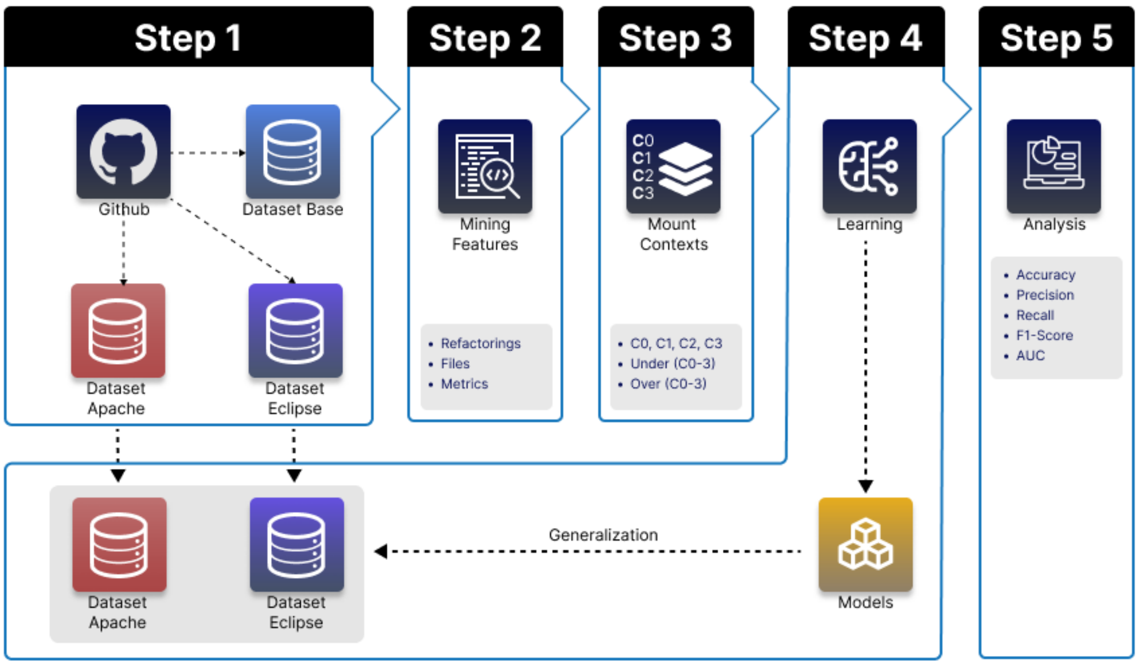
RQ$_4$: Can the best models be carried over to different contexts? – RQ$_4$ aims to understand whether the best models should be trained for a given context and whether it generalizes enough to different contexts. By answering **RQ$_4$**, we can reduce the cost that a new training can bring. In addition, we identified the ability to handle large volumes of data and avoid the cost of identifying complex patterns.

4.2.2 Study Steps and Procedures

Figure 10 overviews the sequence of five-step that we have followed to answer our RQs: (1) Selection and analysis of open source systems; (2) Detect refactoring opportunities and features mining; (3) Contexts Selection; (4) Training and testing the models; and (5) Evaluation Results. We describe each step as follows.

Step 1: Selection and analysis of open-source software systems. The first step

Figure 10 – Overview of the research methodology



Source: Prepared by the author.

consisted of selecting a set of open-source software systems. For our study, we needed to gather a large number of open-source projects to allow the study replication. For this, we have built three sets of data. We used the dataset used in the last article plus two similar datasets in order to minimize any bias produced by just one dataset. The new ones have the same characteristics and are compatible with the used tools. The first dataset (D1), namely in this study as the base dataset used in the last study (Pinheiro *et al.*, 2022). We selected 884 software projects from a dataset of engineering software projects from different authors. The second dataset (D2) is composed of 207 projects from the Apache ecosystem. Finally, the third dataset (D3) is composed of 200 projects from the Eclipse ecosystem. These projects were chosen because the authors observed evidence of solid software engineering practices, including collaboration, continuous integration, quality, maintainability, sustained evolution, project management, responsibility, and unit testing. All projects were extracted from GitHub by our Python scripts developed. Table 6 summarizes the data for the selected software systems. The first column contains the name of each ecosystem in the dataset, followed by the number of projects, commits, and refactorings. The replication package of the previous study[1] and extension[2] contains their detailed information.

Step 2: Detect refactoring opportunities and features mining. In this step, we

[1] Available at https://doi.org/10.5281/zenodo.6800385.
[2] Available at https://doi.org/10.5281/zenodo.7820168.

Table 6 – Overview of the selected datasets

Ecosystem	# Projects	# Commits	# Refactorings
D1 (Base)	884	35,838	84,262
D2 (Apache)	207	272,096	1,144,365
D3 (Eclipse)	200	153,610	767,111
Total	**1,291**	**461.544**	**1,995,738**

Source: Prepared by the author.

have extracted the data about refactorings and code metrics (used as features) for all selected projects. To this end, we have performed three key activities: (1) extracting code refactorings; (2) tracking the modified files before and after refactorings, and (3) extracting code metrics to be used as features. We detailed each step as follows.

Activity 1: Code refactorings extraction. We detected refactorings for all selected projects. For this end, we chose RMiner, (version 2.0) as the tool to detect code refactorings due to its high accuracy (Tsantalis *et al.*, 2018). This tool is applied between two versions (commits) and returns the elements that changed from one version to another. It also returns the refactoring type associated with the change. The tool detected a total of 1.995.738 refactoring types used in our study (see Table 6). It is important to note that there may be some noise in the types of refactorings collected, as it was not verified whether they were performed intentionally by developers or unintentionally. After the code refactoring extraction, we divided the refactorings into two groups, trivial and non-trivial refactorings, as described in Section 2.1.

Activity 2: Tracking the modified files before and after refactorings. To analyze the prediction of trivial and non-trivial refactorings, we need to track the modified files before and after the refactoring application. Thus, we tracked the version before and after each file undergoing trivial and non-trivial refactoring. To track the modified files, we utilized Pydriller (Spadini *et al.*, 2018) and the Jupyter Notebook (Kluyver *et al.*, 2016) to process the data. Thus, we tracked the version before and after each file undergoing trivial and non-trivial refactoring. A total of 39,423,447 files involved in refactoring operations were analyzed.

Activity 3: Extracting code metrics for tracked files. In this activity, we extracted the code metrics and some attributes of code elements to be used as features in our study. To this end, we have used the CK tool (Aniche, 2015) to extract each metric and attribute. Additionally, we created a Python script to automatize summarizing the file outputs provided by the CK tool in a single file. For all fields calculated by the tool, after previous data analysis, we decided to use only 45 metrics and attributes as features for our datasets. They can be seen in Table 7.

Table 7 – Metrics and attributes of code elements used in this work (Lorenz; Kidd, 1994; Chidamber; Kemerer, 1994)

Field	**Description**
	Metrics
cbo	Coupling between objects. Counts the number of dependencies a class has.
wmc	Weight Method Class or McCabe's complexity. It counts the number of branch instructions.
noc	Number of Children. It counts the number of immediate subclasses that a particular class has.
rfc	Response for a Class. Counts the number of unique method invocations in a class.
lcom	Lack of Cohesion of Methods a normalized metric that computes the lack of cohesion of class
nosi	Number of static invocations. Counts the number of invocations to static methods
loc	Lines of code. It counts the lines of the count, ignoring empty lines and comments
	Code elements attributes
totalMethodsQty	Counts the number of all methods.
staticMethodsQty	Counts the number of static methods.
publicMethodsQty	Counts the number of public methods.
privateMethodsQty	Counts the number of private methods.
protectedMethodsQty	Counts the number of protected methods.
defaultMethodsQty	Counts the number of default methods.
visibleMethodsQty	Counts the number of visible methods.
abstractMethodsQty	Counts the number of abstract methods.
finalMethodsQty	Counts the number of final methods.
synchronizedMethodsQty	Counts the number of synchronized methods.
totalFieldsQty	Counts the number of all fields
staticFieldsQty	Counts the number of static fields
publicFieldsQty	Counts the number of public fields
privateFieldsQty	Counts the number of private fields
protectedFieldsQty	Counts the number of protected fields
defaultFieldsQty	Counts the number of default fields
finalFieldsQty	Counts the number of final fields
synchronizedFieldsQty	Counts the number of synchronized fields
returnQty	The number of return instructions
loopQty	The number of loops like for, while, do while and enhanced for
comparisonsQty	The number of comparisons == and !=
tryCatchQty	The number of try/catches
parenthesizedExpsQty	The number of expressions inside parenthesis
stringLiteralsQty	The number of string literals
numbersQty	The number of numbers literals int, long, double, float
assignmentsQty	The number of same or different comparisons
mathOperationsQty	The number of math operations (times, divide, remainder, plus, minus, left shit, right shift)
variablesQty	The number of declared variables
maxNestedBlocksQty	The highest number of blocks nested together
anonymousClassesQty	The quantity of anonymous classes
innerClassesQty	The quantity of inner classes
lambdasQty	The quantity of lambda expressions
uniqueWordsQty	The algorithm basically counts the number of words in a class, after removing Java keywords
typeAnonymous	Boolean indicating whether is an anonymous class
typeClass	Boolean indicating whether is a class
typeEnum	Boolean indicating whether is an enum
typeInnerclass	Boolean indicating whether is an inner class
typeInterface	Boolean indicating whether is an interface
Total: 45	

Source: Prepared by the author.

Step 3: Contexts Selection. In this step, with the datasets defined (**D1**, **D2** and **D3**), we separate and combine each dataset by type of refactoring (trivial and non-trivial) and state of refactoring (before and after the activity occurred). Each separation and combination in this study we call *context*. Furthermore, we subdivide each context into two classes depending on the type and state of the refactoring. In this study, we call one class **0** and the other **1**. The Table 8 presents all the divisions.

The base context (**C0**) is defined by separating features from files that have undergone non-trivial refactoring activity. Class 0 is for features before the activity is executed, while class 1 is for features after the activity is executed.

The next context (**C1**) is defined by file features that have undergone the trivial and non-trivial refactoring activity. Class 0 is for features after the trivial refactoring activity is performed, while class 1 is for features after the non-trivial refactoring activity is performed.

Context two (**C2**) is defined by file features that have undergone trivial and non-trivial refactoring activity. Class 0 is for features before the trivial and non-trivial refactoring activity is performed, while class 1 is for features after the trivial and non-trivial refactoring activity is performed.

Context three (**C3**) is defined by file features that have undergone trivial and non-trivial refactoring activity. Class 0 is for features before the trivial refactoring activity is performed, while class 1 is for features after the non-trivial refactoring activity is performed.

It is important to highlight that most contexts included refactorings since we sought to investigate how they can affect the prediction of non-trivial refactorings. The number of instances of each context can be seen in Table 8.

Table 8 – Instance numbers of contexts

Context	**D1**		**D2**		**D3**	
class	0	1	0	1	0	1
C0	251,416	258,010	1,004,983	992,991	625,274	649,768
C1	232,468	258,010	1,265,956	992,991	446,880	649,768
C2	364,015	377,879	1,633,881	1,630,119	845,395	876,527
C3	112,599	258,010	628,828	992,991	220,121	649,768

Source: Prepared by the author.

Step 4: Training and testing the models. In this step, we used the datasets constructed by the combinations **C1**, **C2** and **C3** created in the previous step to predict refactorings. All contexts have been tested, with some changes to the processing pipeline. Thus, the data from each dataset was split into two datasets: 80% for the training set (used to train the model) and 20% for the test set (used to validate and test the model). The trained models formed binary classifiers based on supervised ML algorithms: Random Forest, Decision Tree, Logistic Regression, Naive Bayes, and Neural Network. The first four algorithms were used through the Scikit-learn library[3], while the Neural Network was used through TensorflowKeras[4]. After

[3] https://scikit-learn.org/stable/
[4] https://www.tensorflow.org/api_docs/python/tf/keras

training, each generated model was validated by predicting the refactorings of the features in the test set. For ML training, a machine with Windows 10 (64-bits) operating system was used, with 16 GB of RAM, 4 CPU cores and 1000 GB of disk available for data collection.

Furthermore, we consider that the prediction of ML algorithms can be negatively affected by an unbalanced dataset (number of different samples between classes). Therefore, we applied two balancing techniques for each combination: Random Under Sampler and SMOTE (Chawla *et al.*, 2002). These balancing techniques were chosen because they are commonly used in recent studies (Moreo *et al.*, 2016; Hasanin; Khoshgoftaar, 2018; Tabassum *et al.*, 2023) and because they facilitate the comparison of efficiency when used together (Mohammed *et al.*, 2020). In addition to unbalanced contexts, the balancing techniques were applied individually by context **C0**, **C1**, **C2** and **C3**, creating eight more combinations: **C0 Under**, **C1 Under**, **C2 Under**, **C3 Under**, **C1 over**, **C1 Over**, **C2 Over** and **C3 Over**.

We optimized the hyperparameters for each model through a GridSearch to find the best combination for the model, utilizing stratified cross-validation with two splits. For each algorithm, we sought the best configuration among parameters. For the **Random Forest**, we tuned parameters such as `max_depth`, `max_features`, `criterion`, `n_estimators`, `min_samples_split`, and `bootstrap`. **Logistic Regression** was optimized by adjusting `max_iter`, `C`, and `solver`. For the **Naive Bayes** classifier, we focused on `var_smoothing`. In the case of **Decision Trees**, the hyperparameters `max_depth`, `max_features`, `min_samples_split`, `splitter`, and `criterion` were fine-tuned.

Furthermore, for the **Neural Network**, a feedforward architecture with dense layers and dropout with `Keras` was used. It includes dense layers with 128, 64, and 64 units using `ReLU activation`, and two dropout layers with a 20% rate to prevent overfitting. The output layer has one unit with `sigmoid activation`, ideal for binary classification. The model uses `binary_crossentropy` as the loss function and the `Adam optimizer`. Training occurs over `10 epochs` with `batches` of 128, and performance is evaluated on the test data.

After training, we tested the models to predict refactoring activity on the datasets. Then, we tested the generated models from the base dataset of the context that obtained better results in the other datasets.

Step 5: Evaluation Results. Finally, we calculated accuracy, precision, recall, F1-score, and Area Under the Curve metrics to evaluate the trained models and compared the results by context. We decided to use the mean as we needed a value to represent the data.

Next, we observed: (i) whether the presence of trivial refactorings affects the prediction of other refactorings; (ii) which algorithm obtained better results; (iii) whether data balancing techniques had any effect; and, (iv) whether the models were able to generalize to other contexts. With this, it was possible to answer our research questions. We present the results in the next section.

4.3 Results and Discussions

In this section, we describe our results. We present an overview of calculating metrics for the contexts mentioned in Table 9 and Table 11. The choice of AUC and F1-score metrics is supported by the need to comprehensively assess the performance of machine learning models in classification tasks. AUC provides a robust measure of the model's discriminative ability in binary scenarios, while the F1-score balances precision and recall, proving particularly useful in cases of class imbalance. When used together, these metrics offer a more comprehensive analysis, enhancing the reliability and validity of the presented results.

We also present the generalization of the models. In the following subsections, we answer each of the RQs.

4.3.1 Performance of ML algorithms to predict trivial and non-trivial refactorings (RQ$_1$)

To answer **RQ**$_1$, we combined several datasets and evaluated the performance of the ML algorithm in predicting trivial refactorings (present in all contexts). Table 9 presents the performance of each ML algorithm by ML metric and context specified in this study.

In summary, our results indicate that the Random Forest algorithm achieved the best performance indices, considering the general average in all contexts, with an average of 0.71, 0.72, 0.74, 0.73 and 0.70 for the accuracy metrics, precision, recall, F1-score, and AUC, respectively. This Random Forest algorithm stood out in the first context, with a remarkable balance (see Table 9). For the context **C1**, the performance was 0.84, 0.86, 0.84, 0.85, and 0.84 for the metrics of accuracy, precision, recall, F1-score, and AUC, respectively. For **C1** with undersampling balancing, the performance was 0.84, 0.86, 0.82, 0.84, and 0.84 for the metrics of accuracy, precision, recall, F1-score, and AUC, respectively.

While for **C1** with oversampling balancing, 0.84, 0.86, 0.83, 0.84, and 0.84 were obtained for the metrics of accuracy, precision, recall, F1-score, and AUC, respectively. Furthermore, in the **C3** context, the models showed even better results with 0.88, 0.89, 0.94, 0.91, and 0.84 for the metrics of accuracy, precision, recall, F1-score, and AUC without balancing the data, respectively. With undersampling balancing, the performance was 0.85, 0.86, 0.84, 0.85, and 0.85. With the data oversampling, the values reached 0.88, 0.89, 0.86, 0.87, and 0.88, respectively.

Finding 1: Models based on the Random Forest algorithm were the best in general contexts, with highlighting for the contexts **C1** and **C3**.

In Table 9, we also observed that the Decision tree in the **C3**, without balancing data, achieved equivalent results of the Random Forest. The indices were (accuracy, precision, recall, F1-score, and AUC) 0.88, 0.90, 0.93, 0.91, and 0.85, against 0.88, 0.89, 0.94, 0.91, and 0.84 of the Random Forest. With both balancing techniques, the results remained equivalent.

Finding 2: Decision tree and Random Forest were the algorithms that achieved better results in the **C3** context, using or not the balancing technique.

As shown in Table 9, the model built by a Neural Network using a balancing technique with data oversampling showed the best results, considering the area under the ROC curve (**AUC**). This model achieved a significant AUC index of 0.91, followed by Random Forest and Decision Tree.

Finding 3: The Neural Network was the algorithm that created the best classification model, considering the model's ability to distinguish between classes, regardless of the chosen cutoff point.

Table 11 presents the values of the metrics obtained from the models trained with data from datasets D2 and D3 in the **C3** context, as well as from the **base dataset** models applied to datasets D2 and D3 in the same context, represented in the Table 11 by D1_D2 and D1_D3.

The models based on **Random Forest**, **Neural Network**, and **Decision Tree** showed great results both in dataset D2 and D3. In D2 the **Decision Tree** model obtained a F1-score and AUC of 95% and 94%, respectively. In the case of **Neural Network**, a F1-score and AUC of 86% and 90% were obtained, respectively. For the **Random Forest** model was obtained a F1-score and AUC of 95% and 93%, respectively.

Similarly, in D3, the performance was also optimistic, in which the **Decision Tree** model obtained a F1-score and AUC of 95% and 90%, respectively. The **Neural Network** model obtained a F1-score and AUC of 89% and 87%, respectively. Finally, the **Random Forest** model obtained a F1-score and AUC of 95% and 89%, respectively. On the other hand, the models based on **Logistic Regression** and **Naive Bayes** algorithms presented inferior results in both datasets. In D2, the **Logistic Regression** model obtained a F1-score and AUC of 75% and 52%, respectively, while with **Naive Bayes** resulted in a F1-score and AUC of 74% and 51%, respectively.

Finding 4: Tree-based and neural network models tend to be more efficient regardless of the dataset.

Table 9 – Results of the different ML models after trained and tested

	None					**Under**					**Over**				
Alg	M	C0	C1	C2	C3	M	C0	C1	C2	C3	M	C0	C1	C2	C3
	acc	0.52	0.52	0.58	0.88	**acc**	0.52	0.84	0.58	0.86	**acc**	0.53	0.85	0.58	0.88
	pre	0.53	0.53	0.59	0.90	**pre**	0.53	0.86	0.60	0.87	**pre**	0.53	0.86	0.60	0.89
Decision	**rec**	0.51	0.48	0.58	0.93	**rec**	0.45	0.83	0.50	0.85	**rec**	0.44	0.83	0.50	0.87
	f1	0.52	0.51	0.58	0.91	**f1**	0.48	0.84	0.54	0.86	**f1**	0.48	0.84	0.55	0.88
	auc	0.52	0.52	0.58	0.85	**auc**	0.52	0.84	0.58	0.86	**auc**	0.53	0.85	0.58	0.88
	acc	0.50	0.52	0.51	0.81	**acc**	0.50	0.61	0.50	0.60	**acc**	0.50	0.61	0.50	0.61
	pre	0.50	0.60	0.51	0.81	**pre**	0.50	0.59	0.50	0.58	**pre**	0.50	0.59	0.50	0.59
Logistic	**rec**	0.93	0.82	0.92	0.97	**rec**	0.49	0.71	0.36	0.71	**rec**	0.48	0.67	0.44	0.74
	f1	0.65	0.69	0.65	0.82	**f1**	0.49	0.65	0.42	0.64	**f1**	0.49	0.63	0.47	0.65
	auc	0.50	0.61	0.50	0.54	**auc**	0.50	0.61	0.50	0.60	**auc**	0.50	0.61	0.50	0.61
	acc	0.49	0.49	0.49	0.68	**acc**	0.50	0.52	0.50	0.52	**acc**	0.49	0.52	0.50	0.52
	pre	0.50	0.50	0.52	0.70	**pre**	0.50	0.51	0.50	0.51	**pre**	0.49	0.51	0.50	0.51
Navie	**rec**	0.07	0.12	0.06	0.95	**rec**	0.06	0.93	0.08	0.92	**rec**	0.06	0.93	0.08	0.92
	f1	0.12	0.20	0.11	0.80	**f1**	0.11	0.66	0.14	0.65	**f1**	0.11	0.66	0.14	0.66
	auc	0.49	0.50	0.50	0.51	**auc**	0.50	0.52	0.50	0.52	**auc**	0.49	0.52	0.50	0.52
	acc	0.50	0.76	0.50	0.82	**acc**	0.50	0.76	0.51	0.75	**acc**	0.49	0.76	0.52	0.82
	pre	0.50	0.74	0.74	0.81	**pre**	0.50	0.74	0.52	0.73	**pre**	0.49	0.73	0.78	0.80
Neural	**rec**	0.97	0.82	0.50	0.95	**rec**	0.97	0.78	0.28	0.80	**rec**	0.95	0.82	0.06	0.81
	f1	0.66	0.78	0.60	0.87	**f1**	0.66	0.76	0.36	0.76	**f1**	0.65	0.78	0.11	0.82
	auc	0.50	0.85	0.51	0.86	**auc**	0.50	0.85	0.51	0.84	**auc**	0.51	0.86	0.51	0.91
	acc	0.53	0.84	0.58	0.88	**acc**	0.53	0.84	0.58	0.85	**acc**	0.53	0.84	0.58	0.88
	pre	0.54	0.86	0.59	0.89	**pre**	0.54	0.86	0.60	0.86	**pre**	0.54	0.86	0.59	0.89
Random	**rec**	0.55	0.84	0.61	0.94	**rec**	0.50	0.82	0.52	0.84	**rec**	0.49	0.83	0.55	0.86
	f1	0.54	0.85	0.60	0.91	**f1**	0.52	0.84	0.56	0.85	**f1**	0.51	0.84	0.57	0.87
	auc	0.53	0.84	0.58	0.84	**auc**	0.53	0.84	0.58	0.85	**auc**	0.53	0.84	0.58	0.88

Source: Prepared by the author.

Implications of RQ$_1$. Our findings indicate that the **Random Forest** and **Decision Tree** algorithms are the most effective. However, regarding the AUC metric, the **Neural Network**

created the best classifier.

In the context of our research, it is crucial to acknowledge that each metric addresses specific facets of the model's performance, and the declaration of an algorithm as the best according to a particular metric does not necessarily imply overall superiority.

For instance, when considering the performance of algorithms such as Neural Networks, which are identified as the best based on the AUC metric, it suggests an exceptional capability for classification in terms of discriminating between classes, as measured by the ROC curve. However, alternative metrics like the F1 score prioritize different aspects, such as precision and recall, potentially yielding disparate conclusions regarding the overall model performance. This discrepancy underscores the significance of selecting evaluation metrics aligned with the specific objectives of the given task.

Furthermore, using a balancing technique, either through undersampling or oversampling of the data, did not significantly improve the results. Therefore, the results obtained in **RQ**$_1$ can help data scientists and developers of automated refactoring tools to make more informed decisions about which algorithms to use when investigating refactorings based on metrics and code attributes.

4.3.2 The effectiveness of including trivial refactorings to predict new refactorings (RQ$_2$)

To answer **RQ**$_2$, we performed several combinations of trivial and non-trivial refactorings, in which each combination corresponds to a context in our study. In total, four different contexts were created. Additionally, two balancing techniques were applied to each of them. The first context (**C0**) is the unique context in which trivial refactorings are not present in any of the classes. We compared the results obtained in other contexts with **C0** to evaluate the effectiveness of including trivial refactorings.

By looking at Table 9, we can see that the values obtained in **C0** are recurrently smaller than **C1**, **C2**, and **C3** even in the set of unbalanced data. To compare the performance of the classification models, we used the values of the metrics F1-Score and AUC. Thus, the **Decision Tree** model obtained an increase of 39% and 33% for F1-Score and AUC, respectively, when including trivial refactorings in the configuration of **C3**. In the same line, the **Logistic Regression** model obtained an increase of 17% (F1-Score) and 4% (AUC) in the configuration of **C3**. The model based on **Naive Bayes** showed a significant increase in the F1-score of 68% and 2% in AUC. **Neural Network** model obtained an increase of 21% and 36% for F1-score and

AUC, respectively. Finally, models based on **Random Forest** achieved similar scores to those based on **Decision Tree**, with increases of 37% (F1-Score) and 31% (F1-Score).

Finding 5: Adding trivial refactorings to different classes along with non-trivial refactorings resulted in a more effective model. This suggests that including trivial refactorings is important for improving the prediction of new refactorings.

For **C2**, For C2, we added file features before passing through a trivial or non-trivial refactoring in one class of the machine learning algorithm, while in the other class, we kept the features of the corresponding files with the refactoring already performed. The dataset has grown by 45%, adding 232,468 rows.

Considering the F1-score and AUC, we obtained similar results. The combination **C2** showed an increase of only 6% for both metrics, using the **Decision Tree** model. We also can observe that no increase in the metrics was observed for the model based on **Logistic Regression**. The same happened with the model based on **Naive Bayes**, but with a loss in the F1-score of 1%. The Neural Network model lost 6% of F1-score and gained 1% of AUC. Finally, Random Forest had a slight increase of 6% in both F1-score and 5% in AUC.

Finding 6: Combining trivial and non-trivial refactorings in the same class does not change the results significantly. This indicates that the presence of trivial refactorings to be positive for refactoring prediction will depend on how they are combined in the dataset.

Implications of RQ$_2$. Trivial refactoring operations can impact the result of predicting new refactorings, which can be positive or negative. In the first case, an increase in accuracy was observed when partially combining the trivial refactorings in the **C1** and **C3** contexts, compared to the context without trivial refactorings (**C0**). In the second case, when combining all trivial and non-trivial refactorings and separating them into before and after refactoring, some cases did not show significant values and even worsened the indices. Therefore, trivial refactorings can improve the models' prediction by choosing the appropriate configuration.

4.3.3 Effectiveness of data balancing techniques in predicting trivial and non-trivial refactorings (RQ_3)

To improve the performance between the models and reduce the outliers between classes, we have evaluated the Effectiveness of two well-known data balancing techniques: *Random Under Sampler* and *Oversampling with SMOTE*. In summary, we observed a significant increase only in contexts that received trivial refactorings. The all outliers are presented with gray highlight in Table 10.

By applying the Undersampling and Oversampling balancing techniques in the **C0** context, which does not have trivial refactorings, we observed that the results obtained were little significant or negative in all algorithms. In Table 10, we highlight that the technique of Undersampling had a significant negative impact on the F1-score of the **Logistic Regression** model, with a worsening of 16%. In the other models, the variation of worsening was from 0% to 4% in the F1-score. In the same way with the Oversampling technique, we obtain the same negative value for models based on **Logistic Regression** in F1-score, with a negative variation between 1% and 5%.

In the other contexts - **C1**, **C2**, and **C3** - we have observed a worsening in almost all algorithms. The context **C2** stood out negatively, using the **Neural Network** algorithm, with a loss of 24% in the F1-score using Undersampling and 49% in the Oversampling of the data.

On the other hand, the model based on the **Decision Tree** algorithm stood out positively in the **C1** context, with F1-score increasing by 33% in both Undersampling and Oversampling. Similarly, AUC (Area Under the Curve) also increased by 32% with Undersampling and 33% with Oversampling. Furthermore, Table 10 presents the model based on the **Naive Bayes** algorithm increased its F1-score by 46% with the applied techniques.

Finding 7: For our problem, balancing the dataset up or down usually keeps the same result or makes the model worse.

The algorithm **Navie Bayes** in the **C1** context obtained a lower result, with a recall of 12% and an F1-score of 20%. However, when applying the Undersampling and Oversampling balancing techniques in this context, the model obtained a significant improvement of 81% and 46% in recall and F1-score, respectively.

Furthermore, the models based on the **Naive Bayes** algorithm in the **C3** context showed good recall and F1-score indices, with 95% and 80%. respectively. However, we observed a worsening with the use of balancing techniques, in which the use of Undersampling and Oversampling resulted in a worsening of 15% and 14% in the F1-score, respectively.

Table 10 – Performance of algorithms in the contexts with balancing techniques

	C0		C1		C2		C3	
Alg	under	over	under	over	under	over	under	over
	0.00	0.01	0.32	0.33	0.00	-0.02	-0.02	0.00
	0.00	0.00	0.33	0.33	0.01	-0.03	-0.03	-0.01
decision	-0.06	-0.07	0.35	0.35	-0.08	-0.08	-0.08	-0.06
	-0.04	-0.04	0.33	0.33	-0.04	-0.03	-0.05	-0.03
	0.00	0.01	0.32	0.33	0.00	0.00	0.01	0.03
	0.00	0.00	0.09	0.09	-0.01	-0.01	-0.21	-0.20
	0.00	0.00	-0.01	-0.01	-0.01	-0.01	-0.23	-0.22
logistic	-0.44	-0.45	-0.11	-0.15	-0.56	-0.48	-0.26	-0.23
	-0.16	-0.16	-0.04	-0.06	-0.23	-0.18	-0.18	-0.17
	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.07
	0.01	0.00	0.03	0.03	0.01	0.01	-0.16	-0.16
	0.00	-0.01	0.01	0.01	-0.02	-0.02	-0.19	-0.19
navie	-0.01	-0.01	0.81	0.81	0.02	0.02	-0.03	-0.03
	-0.01	-0.01	0.46	0.46	0.03	0.03	-0.15	-0.14
	0.01	0.00	0.02	0.02	0.00	0.00	0.01	0.01
	0.00	-0.01	0.00	0.00	0.01	0.02	-0.07	0.00
	0.00	-0.01	0.00	-0.01	-0.22	0.04	-0.08	-0.01
neural	0.00	-0.02	-0.04	0.00	-0.22	-0.44	-0.15	-0.14
	0.00	-0.01	-0.02	0.00	-0.24	-0.49	-0.11	-0.05
	0.00	0.01	0.00	0.01	0.00	0.00	-0.02	0.05
	0.00	0.00	0.00	0.00	0.00	0.00	-0.03	0.00
	0.00	0.00	0.00	0.00	0.01	0.00	-0.03	0.00
random	-0.05	-0.06	-0.02	-0.01	-0.09	-0.06	-0.10	-0.08
	-0.02	-0.03	-0.01	-0.01	-0.04	-0.03	-0.06	-0.04
	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.04

Source: Prepared by the author.

Finding 8: The model with the worst results in the **C1** context obtained the best use of the balancing techniques.

We also observed that the model based on **Logistic Regression** obtained the worst results when applying data balancing techniques. In the **C0** context, using the undersampling technique resulted in a 44% and 16% reduction in recall and F1-Score, respectively. Similarly, using the oversampling technique resulted in a reduction of 45% and 16% in recall and F1-Score, respectively.

In the **C1** context, the reduction in recall and F1-Score were 11% and 4% with undersampling and 15% and 6% with Oversampling. In the case of the **C2** context, the reduction was even more significant, with a worsening of 56% and 23% in recall and F1-Score with the use of Undersampling and 48% (recall) and 18% (F1-Score) of worsening in the use of Oversampling. Finally, in the **C3** context, the reduction was 26% and 18% with undersampling and 23% and 17% with oversampling in the values of recall and F1-Score, respectively.

Finding 9: The Logistic Regression algorithm was the one that deteriorated the most with the use of balancing techniques.

Implications of RQ$_3$. The data balancing techniques' results varied in the different models, both by context and algorithm. In some cases, a complete rejection of the technique was observed since the use of the technique did not result in improvements or at least maintained the original results.

4.3.4 Generalization of the best model in other data context domain (RQ$_4$)

To answer RQ$_4$, we evaluated the best models obtained in **C3** context with the **base dataset** with respect to datasets with different named data domains (D2 and D3). These other datasets were configured in the same **C3** context, trained, and evaluated. Next, we evaluate the performance of the model in terms of predicting refactorings, we also compared it with the models trained using the **base dataset**. Results can be seen in the Table 11 and Table 12.

Table 12, shows the values of the differences of the metrics obtained from the model trained with data from the base dataset applied in dataset D2 and D3 with the values obtained from the models trained in the same data domain of D2 and D3. All models showed low or no variation, by presenting values between 0% and 5%, except for the **Neural Network** model which showed significant variation. Additionally, the values obtained from the variation of the **Neural Network** models trained in the base dataset and applied in D2 were 17% in the F1-score and 41% in the AUC metric.

Similarly, a variation was observed in D3, in which was obtained at 8% in F1-score and 37% in AUC for less. The values of the AUC metrics were the ones that most distanced themselves from the values obtained by the models when trained in the domain itself.

Finding 10: Most of the models trained by the base dataset obtained satisfactory results when generalized to other domain contexts.

Table 11 – Result of generalization in the best context

		None					**Under**					**Over**			
Alg	M	D2	D1_D2	D3	D1_D3	M	D2	D1_D2	D3	D1_D3	M	D2	D1_D2	D3	D1_D3
	acc	0.94	0.91	0.92	0.89	**acc**	0.94	0.90	0.91	0.85	**acc**	0.94	0.92	0.92	0.89
	pre	0.95	0.92	0.94	0.92	**pre**	0.94	0.90	0.91	0.85	**pre**	0.94	0.92	0.93	0.90
Decision	**rec**	0.95	0.94	0.95	0.94	**rec**	0.94	0.91	0.90	0.84	**rec**	0.94	0.92	0.92	0.89
	f1	0.95	0.93	0.95	0.93	**f1**	0.94	0.91	0.91	0.85	**f1**	0.94	0.92	0.92	0.89
	auc	0.94	0.91	0.90	0.85	**auc**	0.94	0.90	0.91	0.85	**auc**	0.94	0.92	0.92	0.89
	acc	0.62	0.62	0.75	0.75	**acc**	0.58	0.58	0.59	0.58	**acc**	0.58	0.57	0.59	0.59
	pre	0.62	0.62	0.75	0.75	**pre**	0.58	0.58	0.58	0.59	**pre**	0.58	0.56	0.58	0.57
Logistic	**rec**	0.96	0.97	0.99	0.99	**rec**	0.58	0.57	0.62	0.57	**rec**	0.58	0.58	0.66	0.68
	f1	0.75	0.76	0.85	0.85	**f1**	0.58	0.57	0.60	0.58	**f1**	0.58	0.61	0.62	0.62
	auc	0.52	0.52	0.51	0.51	**auc**	0.58	0.58	0.59	0.58	**auc**	0.58	0.57	0.59	0.59
	acc	0.60	0.61	0.74	0.74	**acc**	0.51	0.51	0.51	0.51	**acc**	0.51	0.51	0.51	0.51
	pre	0.61	0.61	0.75	0.74	**pre**	0.50	0.50	0.50	0.50	**pre**	0.50	0.50	0.50	0.50
Navie	**rec**	0.94	0.94	0.98	0.98	**rec**	0.94	0.94	0.96	0.96	**rec**	0.94	0.93	0.97	0.96
	f1	0.74	0.74	0.85	0.85	**f1**	0.65	0.65	0.66	0.66	**f1**	0.65	0.65	0.66	0.66
	auc	0.51	0.51	0.50	0.50	**auc**	0.51	0.51	0.51	0.51	**auc**	0.51	0.51	0.51	0.51
	acc	0.82	0.56	0.84	0.69	**acc**	0.91	0.49	0.77	0.49	**acc**	0.82	0.49	0.82	0.51
	pre	0.83	0.61	0.85	0.75	**pre**	0.91	0.49	0.77	0.49	**pre**	0.80	0.49	0.82	0.51
Neural	**rec**	0.89	0.81	0.94	0.88	**rec**	0.80	0.60	0.79	0.61	**rec**	0.84	0.48	0.82	0.51
	f1	0.86	0.69	0.89	0.81	**f1**	0.81	0.54	0.78	0.55	**f1**	0.82	0.49	0.82	0.51
	auc	0.90	0.49	0.87	0.50	**auc**	0.89	0.49	0.85	0.49	**auc**	0.90	0.49	0.90	0.50
	acc	0.94	0.91	0.92	0.89	**acc**	0.84	0.91	0.90	0.85	**acc**	0.94	0.92	0.92	0.90
	pre	0.95	0.92	0.94	0.91	**pre**	0.94	0.90	0.91	0.85	**pre**	0.94	0.92	0.93	0.90
Random	**rec**	0.95	0.94	0.95	0.95	**rec**	0.93	0.91	0.90	0.86	**rec**	0.94	0.92	0.91	0.89
	f1	0.95	0.93	0.95	0.93	**f1**	0.94	0.91	0.90	0.85	**f1**	0.94	0.92	0.92	0.89
	auc	0.93	0.91	0.89	0.84	**auc**	0.94	0.91	0.90	0.85	**auc**	0.94	0.92	0.92	0.90

Source: Prepared by the author.

Table 11 presents the values obtained from the application of the data balancing techniques in D2 and D3. We can observe that in D2, the models that underwent the Undersampling technique obtained the worst results. Models based on **Decision Tree** had a negative decrease of -1% in the F1-score metric and -1% in the AUC.

The **Logistic Regression** based models also had a negative decrease of -17% in the F1-score metric, but a 6% increase in AUC. Additionally, those based on **Naive Bayes** also had a negative decrease of -9% in the F1-score metric and maintained the same value in the AUC metric. Furthermore, the models based on **Neural Network** obtained a negative decrease of -5% in the F1-score metric and -1% in the AUC.

However, only the **Random Forest** based models showed even a small improvement with the balancing technique, with an increase in the AUC metric by +1%. Furthermore, still in D2, but with the Oversampling technique, the results were very similar, with a slight loss of the F1-score metric of -4% in the models based on **Neural Network**.

In D3, with the same balancing technique, the models obtained slightly different results when compared to D2. The **Decision Tree** based models obtained a negative decrease

Table 12 – Performance of generalization in the best context

	None		Under		Over	
Alg	**D1_D2**	**D1_D3**	**D1_D2**	**D1_D3**	**D1_D2**	**D1_D3**
	-0.03	-0.03	-0.04	-0.06	-0.02	-0.03
	-0.03	-0.02	-0.04	-0.06	-0.02	-0.03
decision	-0.01	-0.01	-0.03	-0.06	-0.02	-0.03
	-0.02	-0.02	-0.03	-0.06	-0.02	-0.03
	-0.03	-0.05	-0.04	-0.06	-0.02	-0.03
	0.00	0.00	0.00	-0.01	-0.01	0.00
	0.00	0.00	0.00	0.01	-0.02	-0.01
logistic	0.01	0.00	-0.01	-0.05	0.00	0.02
	0.01	0.00	-0.01	-0.02	0.03	0.00
	0.00	0.00	0.00	-0.01	-0.01	0.00
	0.01	0.00	0.00	0.00	0.00	0.00
	0.00	-0.01	0.00	0.00	0.00	0.00
navie	0.00	0.00	0.00	0.00	-0.01	-0.01
	0.00	0.00	0.00	0.00	0.00	0.00
	0.00	0.00	0.00	0.00	0.00	0.00
	-0.26	-0.15	-0.42	-0.28	-0.33	-0.31
	-0.22	-0.10	-0.42	-0.28	-0.31	-0.31
neural	-0.08	-0.06	-0.20	-0.18	-0.36	-0.31
	-0.17	-0.08	-0.27	-0.23	-0.33	-0.31
	-0.41	-0.37	-0.40	-0.36	-0.41	-0.40
	-0.03	-0.03	0.07	-0.05	-0.02	-0.02
	-0.03	-0.03	-0.04	-0.06	-0.02	-0.03
random	-0.01	0.00	-0.02	-0.04	-0.02	-0.02
	-0.02	-0.02	-0.03	-0.05	-0.02	-0.03
	-0.02	-0.05	-0.03	-0.05	-0.02	-0.02

Source: Prepared by the author.

of -4% in the F1-score metric and an increase of +1% in the AUC. Similarly, the **Logistic Regression** based models also obtained a negative decrease of -25% in the F1-score metric and an increase of +8% in the AUC. In the case of **Naive Bayes** based models, we also observed a negative decrease of -19% in the F1-score metric and an increase of +1% in the AUC metric. Furthermore, the **Neural Network** based models obtained a negative decrease of -11% in the F1-score metric and -2% in the AUC. Finally, those **Random Forest** based models obtained a -5% decrease in the F1-score metric and +1% increase in AUC.

We also observed that in the same dataset (D3), the Oversampling technique obtained similar results, with emphasis on the **Random forest** based models, which presented an increase in the AUC by +3% and a decrease of the F1-score to -3%. Additionally, in the case of **Neural Network** based models, we observed an increase in AUC of +3% and a decrease of F1-score to -7%.

Similarly, the balancing technique applied to the models trained with the base dataset and applied to the D2 and D3 datasets obtained little variation, between -6% and 7%. Except in the case of **Neural Network** based models.

In D2, the generalized models based on the Neural Network algorithm obtained a difference between the model of the domain itself of -17% for the F1-score (Table 12). This difference increased to -27% with the use of the Undersampling technique and to -33% with the Oversampling technique. Similarly, on D3, the values obtained for models based on Neural Network were -8% in F1-score. In the case of the Undersampling technique, was obtained -23% (F1-score), and Oversampling with -31% (F1-score).

Finding 11: The balancing techniques applied to models from other domains generated negative or little positive results.

Implications of RQ$_4$. In general, the generalization of the models trained with the base dataset was positive. Although refactoring data was extracted from multiple projects, the models were able to identify refactorings based on code attributes and metrics, regardless of the data domain.

4.4 Threats to Validity

This section discusses threats to the validity of the study according to the classification of (Wohlin *et al.*, 2012).

Internal validity. In our study, we used RefactoringMiner (Tsantalis *et al.*, 2018), a high-precision tool to detect refactoring opportunities in commits, Pydriller (Spadini *et al.*, 2018) to extract source code from files and CKTool (Aniche, 2015) to obtain code metrics of files involved in refactorings. Despite the high accuracy, these tools may still fail during the process of mining. To mitigate this problem, we have repeated some steps of the process when necessary. Additionally, to find out the impact that trivial refactorings have on the prediction of other refactorings, we grouped the features of a set of refactoring operations in the same class and this can cause a drop in the performance of the models.

External validity. Despite a large number of projects (1,291) and code refactorings analyzed (1,995,738), different results can be obtained depending on the domain of systems in terms of: (i) programming language, (ii) maintainability, (iii) used programming paradigm, or (iv) software quality. Regarding data balancing, we emphasize that the unbalance of instances and refactorings accounted in each dataset can negatively influence the result during the process of predicting refactorings. To mitigate this threat, we applied balancing techniques in different contexts: Oversampling strategy with SMOTE and Random Undersampling. However, there was an unbalance in the generalization, as the models trained on the base dataset had the lowest proportion and were compared to the models trained on the datasets with the highest proportions.

Construct validity. One threat to validity may be the size of the dataset chosen for the study. This size was chosen based on previous studies on refactorings (Aniche *et al.*, 2020; AlOmar *et al.*, 2021; Peruma *et al.*, 2020). However, we do not know if it is the right size to find the best solution to our problem. This, finding the solution with different dataset sizes may yield more efficient results. Another important threat refers to the metrics used to build the dataset. However, we have used well-known metrics in the literature: accuracy, precision, recall, F1-score, and Area Under the Curve metrics. In addition, it is necessary to investigate and systematize the choice of metrics based on the object-oriented paradigm.

Conclusion validity. We investigated the effect of trivial refactorings on the prediction of non-trivial ones. To identify how the former affects the latter, data from files involved in both types of refactorings are used and tested on the same and different classes in the prediction models. This relationship may cause some bias in the results at the prediction time. This may affect our conclusion.

4.5 Concluding Remarks

Our study investigated how trivial (class-level) refactorings can affect the prediction of non-trivial refactorings across attributes and code metrics. Our experiment was carried out on 1,291 open-source projects and used the following algorithms as a supervised learning technique to create classifiers: Decision Tree, Logistic Regression, Navies Bayes, Random Forest and Neural Network. Our study also used two data balancing techniques: Random Oversampling and SMOTE Oversampling. We grouped refactorings according to their triviality and proposed contexts based on combinations of refactoring types. In addition, we separated the datasets to identify possible generalizations of the models.

This study is useful for software engineering tool developers who can use these models to improve refactoring suggestions. It also allows researchers to optimize code refactoring by increasing the efficiency of the technique through the methodology used. In addition, the study serves as a basis for future research on new approaches in refactoring analysis and prediction.

Our main findings: (i) ML with tree-based models such as Random Forest, Decision Tree, and Neural network performed very well when trained with code metrics to detect refactoring opportunities. However, only the first two are able to reach a good generalization in other data domain contexts of refactoring; (ii) separating trivial and non-trivial refactorings into different classes still results in a more efficient model, even on different datasets; and (iii) using balancing techniques that increase or decrease samples may not be the best strategy to improve models trained by datasets composed of code metrics and configured according to our study. (iv) We understand that a possible explanation for the performance improvements when "trivial refactorings" are included is that the machine learning models have increased knowledge of what is not non-trivial refactoring, thus improving their prediction.

In future work, we intend to: (i) Create a triviality index that best defines a trivial refactoring operation and quantifies that triviality; (ii) identify other attributes and metrics that can produce more efficient results for predicting refactorings; (iii) perform an in-depth investigation of other algorithms that may perform better in predicting refactorings; and, (iv) investigate how models that predict trivial refactorings impact the detection of refactorings performed by automated solutions.

5 TOWARDS AN EFFECTIVE REFACTORING TRIVIALITY INDEX: A MACHINE LEARNING APPROACH FROM A DEVELOPER'S PERSPECTIVE

This chapter presents a study that addresses the use of software refactoring through the analysis and proposal of a metric called "Refactoring Triviality Index", which evaluates the degree of difficulty of implementing a refactoring operation from the point of view of software developers, considering its complexity, speed and risk. To this end, we identify the most relevant code metrics according to developers, as well as analyze the effectiveness of different ML algorithms, using seven predictive models to evaluate the performance of the triviality index in three large open-source project ecosystems: Apache, Eclipse, and Random. The study investigates how the prioritization of the features considered most important by developers affects the effectiveness of ML models in predicting the triviality index of refactorings. In addition, it analyzes the agreement of Triviality Index (TI) with the perception of developers with experience in refactoring activities.

The study contributes to the advancement of automated refactoring solutions by proposing a metric to evaluate the triviality of implementing one refactoring with its effectiveness aligned with the developers' perception, providing more precise detail on the use of the technique to improve software quality.

5.1 Introduction

Software refactoring is a crucial activity for maintaining and improving code quality. Mens and Tourwé (2004) emphasize that refactoring is an essential practice in agile software development, as it allows developers to maintain clean and adaptable code, which is essential for the continuous evolution of systems. According to Hutton (2009), refactoring can also be seen as a continuous and iterative practice that is part of the software development life cycle, contributing to reducing technical debt and improving the overall software quality.

Software maintenance and evolution are continuous processes that require regular updates and improvements to ensure the quality and longevity of the system (Lehman, 1980). In this context, knowing that refactoring plays a fundamental role in contributing to software maintainability and extensibility (Fowler, 2018; Opdyke, 1992) the assessment of the complexity and risk associated with each refactoring remains an underexplored challenge, highlighting the need to develop more accurate and reliable methods to predict the impact of these changes (Mens; Tourwé, 2004). Furthermore, despite the wide adoption of refactoring techniques, assessing the triviality of refactoring is still a challenge (Murphy-Hill; Black, 2008). Current tools offer only basic, often imprecise, indications, which can lead to risky refactorings or missed opportunities for code improvement (Kim *et al.*, 2014; Silva *et al.*, 2016).

In recent years, the scientific community has explored several approaches to improve the use of software refactoring. One line of research focuses on developing solutions that identify and recommend specific refactorings to developers, aiming to improve code quality and maintainability (Bavota *et al.*, 2015; Tsantalis *et al.*, 2018). Developers' motivation to perform refactorings has also been examined, revealing that, although the practice is recognized as beneficial, it is often avoided due to the perception of risk or lack of time (Silva *et al.*, 2016; Palomba *et al.*, 2017; Paixão *et al.*, 2020). Another area of study identifies the challenges faced by developers in applying these practices, highlighting the complexity and risk associated with refactorings, which often result in making undesirable decisions or underutilizing improvement opportunities (Kim *et al.*, 2014; Sharma *et al.*, 2015). At the same time, detecting refactoring opportunities through ML techniques has gained prominence. Recent studies demonstrate that the use of ML can effectively identify areas of code that would benefit from refactorings, making this approach promising for the development of software engineering (Azeem *et al.*, 2019; Aniche *et al.*, 2020; Nucci *et al.*, 2018; Khleel; Nehéz, 2023).

However, the application of ML to propose solutions that improve refactoring activity remains an underexplored area. The lack of an accurate assessment of refactoring triviality generates challenges in software maintenance, such as the difficulty in selecting the most appropriate refactoring technique for different projects and ensuring that refactoring does not change the external behavior of the system (Akhtar *et al.*, 2022). Without a reliable approach, developers prefer to spend a lot of energy on manual, time-consuming, and risky refactoring, hesitating to perform necessary refactorings through automated tools, overestimating the risk associated with them (Silva *et al.*, 2016; Abid *et al.*, 2022). This study addresses this gap by proposing a ML based solution to identify the triviality of implementing refactoring.

The main objective of this study is to propose to software developers a metric that can be implemented in an automated solution, which can generate more confidence in the implementation of refactoring by evaluating the triviality of the refactoring. The "Refactoring Triviality Index" is a composite metric reflecting the complexity, speed and risk of refactoring implementation. This index is expressed as a continuous value between 0 and 1, where higher values indicate greater triviality. Triviality does not depend on the specific context of the software, as it is constituted by code metrics. Furthermore, the proposal allows triviality to be weighted according to the needs of the developer or project manager. This provides flexibility to define, based on their assessment, to what extent a refactoring will be considered trivial or not. In this way, developers can make safer decisions and perform refactorings with greater confidence, improving the overall quality of software development.

The remainder of this article is organized as follows. Section 5.2 presents our study settings. Section 5.3 presents our main findings, followed by a discussion. Section 5.4 discusses the main threats to validity. Finally, Section 5.5 concludes the article and suggests future work.

5.2 Study Settings

5.2.1 Goal and Research Questions

To define the goal and research questions (*RQs*), we use the Goal-Question-Metric (GQM) model (Wohlin *et al.*, 2012). Our goal is to: **analyze** and propose an index that evaluates the triviality of refactoring; **with the purpose of** identify the degree of difficulty of its implementation; **in relation to** the aspects of complexity, speed, and risk; **from the point of view** of software developers; **in the context of** refactoring operations. We detail each *RQ* as follows:

RQ$_1$: Which code metrics are considered most relevant by developers to determine the triviality of a refactoring operation? - RQ$_1$ aims to identify the code metrics that developers consider most relevant, considering three aspects related to the definition of triviality used in this study: complexity, speed, and risk. This may include metrics such as cyclomatic complexity, number of code lines, coupling, cohesion, among others. To this end, we sent a questionnaire to capture the perceptions of internal developers (i.e., those who participated in the development of the projects), as well as developers external to the investigated projects. Additionally, we investigated whether there is a discrepancy in the developers' different perspectives regarding the relevance of the metrics. By answering **RQ$_1$**, we identified which metrics are most relevant to make it possible to create the triviality index.

RQ$_2$: How do different ML techniques behave in predicting the code refactorings triviality index? - RQ$_2$ investigates the effectiveness of different ML algorithms, both those based on Regression such as Logistic Regression and those based on ensemble trees such as Random Forest, in predicting the refactorings triviality index. The comparison is based on performance metrics such as mean squared error (MSE) and coefficient of determination (R^2). By answering **RQ_2**, we can identify which algorithms produce the best results for our different datasets.

RQ_3: What is the impact of prioritizing features ranked by developers on the effectiveness of triviality index prediction models? - RQ_3 investigates how prioritizing features considered most important by developers affects the effectiveness of ML models for predicting the refactorings triviality index. To do so, we compare the effectiveness of models trained with and without feature prioritization. By answering **RQ_3**, we can identify whether models with feature prioritization perform better than models without prioritization.

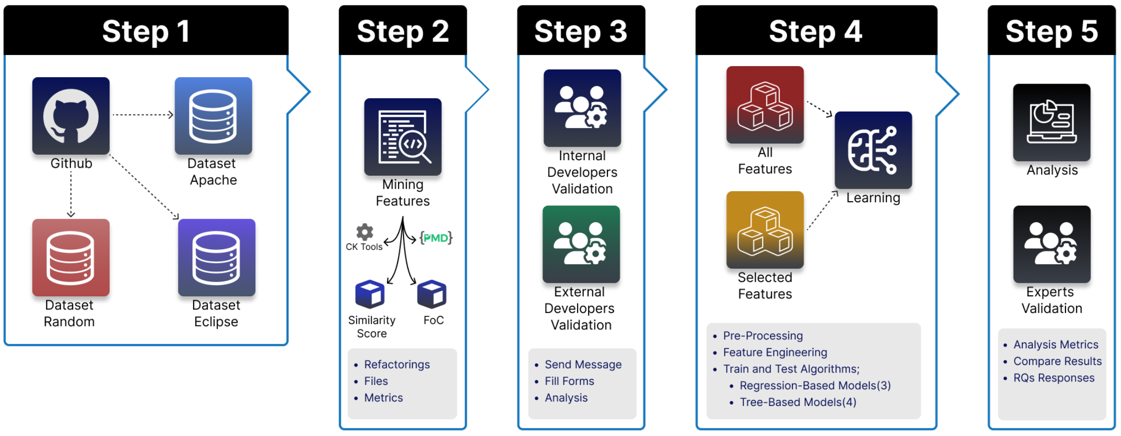
RQ_4: To what extent is the proposed triviality index aligned with the developers' perception regarding the triviality of applying refactorings? - RQ_4 verifies the agreement between the triviality index and the developers' perception. To do this, we initially defined a set of 12 code examples to be refactored with different types of refactoring operations, such as the Extract Method and Pull Up Method. Next, we sent a form to capture the perceptions of 16 developers with experience in applying refactorings. In this form, each developer evaluates the triviality of the aspects used in the construction of the triviality index: complexity, speed, and risk. By answering **RQ_4**, we can evaluate the agreement of the triviality index resulting from the ML model with the developers' perception regarding the triviality of the refactoring operation in each code example.

5.2.2 Study Steps and Procedures

Figure 11 presents an overview of the steps we followed to answer our RQs: (1) Selection and analysis of open source systems; (2) Detection of refactoring opportunities and features mining; (3) Validation with developers; (4) Application of the machine learning technique; and (5) Evaluation of the results. Next, we describe each of these steps.

Step 1: Selection and analysis of open-source systems. The first step of this study consisted of the selection and analysis of open-source systems. We selected datasets that met the following criteria: (i) systems implemented in Java and open source since they are supported by the tool adopted for the extraction of refactorings; (ii) systems that Git must version, necessary for the feature extraction tool; and, (iii) have a significant volume of refactoring operations to facilitate the training of ML algorithms. In addition, the selected datasets were used in our previous study, providing a solid basis for comparison. All projects were extracted from GitHub

Figure 11 – Overview of the research methodology.



Source: Prepared by the author.

using automated scripts developed for this purpose with the GitHub API, they were extracted and stored in JSON files containing information such as name, URL and size of the projects. The datasets totaled 1,259 projects and were distributed across three ecosystems: the Apache dataset with 214 projects, the Eclipse dataset with 194 projects, and the Random dataset with 851 projects from different authors, but they met the requested requirements. Table 13 summarizes the data from the selected software systems. The first column contains the name of the dataset, followed by the number of projects, commits, refactorings and high rerfactorings which are the class and method level refactorings. The replication package of this study[1] can also see it in Table 1.

Table 13 – Summary of data for selected software systems

Ecosystem	# Projects	# Commits	# Refactorings	# High Refactoring
Apache	214	2,148	83,033	30,693
Eclipse	194	2,370	101,635	43,827
Random	851	307	6,962	2,624
Total	**1,259**	**4,825**	**191,630**	**77,144**

Source: Prepared by the author.

Step 2: Detecting refactoring opportunities and mining features. In this step, we extracted data on refactorings and code metrics (used in this study as features) for all selected projects. To do so, we performed three main activities: (1) extracting code refactorings; (2) tracking modified files before and after refactoring; and (3) extracting code metrics to be used as features. We detail each activity below.

[1] Available at https://doi.org/10.5281/zenodo.13766290.

Activity 1: Extracting code refactorings. We detected refactorings in all selected projects using the Refactoring Miner tool (version 2.0), chosen due to its high accuracy, with an average precision of 99.6% and recall of 94% (Tsantalis *et al.*, 2020). This tool is applied between two versions (commits). It identifies the elements that changed from one version to the other, in addition to providing the type of refactoring associated with each change. For detection, we applied a filter to commits containing the term "refactor" in the commit text as an initial way to select commits where the developer expressed the intention to refactor by using this term, because our goal is root refactoring. This strategy has also been adopted in other studies (Silva *et al.*, 2016; AlOmar *et al.*, 2021; Nyamawe, 2022). The tool detected a total of 191,630 refactoring operations. Of these, we selected 77,144 as high refactorings, as those applied at the class and method level. These refactorings generally have a more significant impact on the structure and design of the code, compared to refactorings at the attribute or variable level. They often involve changes in the system architecture, which can improve the code's modularity, cohesion and reusability. Furthermore, they are more likely to present significant variations in the complexity, time and risk associated with their implementation, making these characteristics ideal for modeling the concept of triviality (see Table 14).

Activity 2: Tracking modified files before and after refactoring. To perform the analysis of the prediction of the triviality of refactorings, it was necessary to track the changes made to the code files before and after the application of the refactorings. This process involves identifying the files' versions before and after the refactoring to allow a detailed analysis of the modifications, including test files. We used Pydriller (Spadini *et al.*, 2018), a powerful tool for mining version control repositories, which allowed us to extract relevant information about the modifications in the code. Pydriller offers functionalities to access the commit history in Git repositories, allowing us to identify precisely which files were changed in each commit associated with a refactoring operation. In the scripts developed in Python, we started by loading data from a JSON file containing detailed information about the refactorings applied to different projects. We iterated over the commits involved for each project and refactoring, focusing on class and method level changes, which are critical for our analysis. We used the Pydriller tool to navigate through the commits and identify modified files, specifically filtering for files with the .java extension. For each Java file identified, we extracted the source code versions before and after the modifications and stored them in an organized directory system. This storage system was structured to separate the versions before and after the refactorings, allowing for

Table 14 – High Refactorings by ecosystem

Type of Refactoring	**# Apache**	**# Eclipse**	**# Random**
Add Class Annotation	710	1187	93
Add Class Modifier	342	260	51
Add Method Annotation	2153	7774	198
Add Method Modifier	743	751	39
Change Class Access Modifier	386	708	55
Change Method Access Modifier	2514	6422	228
Extract And Move Method	741	688	74
Extract Class	372	425	47
Extract Method	2113	2525	174
Extract Subclass	53	35	6
Extract Superclass	261	279	16
Inline Method	385	609	19
Merge Class	48	51	2
Merge Method	4	8	0
Modify Class Annotation	384	308	20
Modify Method Annotation	758	559	15
Move And Inline Method	141	163	7
Move And Rename Class	585	434	62
Move And Rename Method	614	629	58
Move Class	5930	4975	430
Move Method	2535	3244	239
Pull Up Method	1778	2384	98
Push Down Method	365	334	55
Remove Class Annotation	691	830	27
Remove Class Modifier	216	119	23
Remove Method Annotation	1150	1793	94
Remove Method Modifier	816	565	58
Rename Class	775	904	81
Rename Method	3077	4777	347
Replace Anonymous With Class	22	19	5
Split Class	17	13	3
Split Method	14	15	0
Total	**30,693**	**11,267**	**2,624**

Source: Prepared by the author.

more efficient access and analysis of the data. In addition, we used Jupyter Notebook (Kluyver *et al.*, 2016) to process and analyze the extracted data. It provided an interactive and flexible environment to explore the data, perform statistical analysis, and visualize the results effectively. This was important to validate the triviality predictions and understand the impact of refactorings on the code files.

Throughout this activity, we analyzed a total of 61,842 files that were involved in refactoring operations. This database allowed us to validate the proposed approach and provide valuable insights into the refactoring process.

Activity 3: Extracting code metrics. In this activity, we extract code metrics and some attributes of code elements to be used as features in our study. To this end, we used automated tools such as CK (Aniche, 2015) and PMD[2] to extract important metrics and elements from Java files (see Table 15).

First, we used the CK tool to extract several metrics of code complexity, coupling, cohesion, inheritance, and other relevant attributes from each code element. For each commit, we ran CK on source code files organized into "before" and "after" steps, storing the generated outputs in specific directories. The execution was automated through Python scripts, which facilitates large-scale data manipulation.

Then, we used the PMD tool employed to analyze problems and potential improvements in Java files. We use custom rules that promote good coding practices, ensure consistency in code style and target design and architecture issues. We also use rules that identify code patterns that can lead to runtime errors, detect concurrent code issues, highlight issues affecting code performance, and identify potential security vulnerabilities. The PMD tool was run on both the original and refactored code, counting the number of issues identified at each stage.

We implemented a Python script to calculate the Frequency of Commits (FoC), a metric that reflects the speed of code changes over time. FoC was calculated over different time intervals (7, 14, 21, and 28 days), representing the frequency with which refactoring commits were made within these periods. The formula used to calculate FoC is:

$$\text{FoC} = \frac{\text{totalCommits}}{\left(\frac{\text{total\_days}}{\text{sprintDays}}\right)}$$

Where **totalCommits** is the total number of refactoring commits, **totalDays** is the number of days between the first and last commits, and **sprintDays** represents the range of days (7, 14, 21, or 28) for which the frequency is being calculated.

Additionally, we created a Java program[3] to calculate the similarity score (based in Jaccard Similarity (Jaccard, 1901)) Java files across commits using the abstract syntax tree (AST) as the basis for the analysis. The similarity between code files is an important metric to evaluate the degree of changes made over time and serves for comparative analysis in terms of refactoring. We used the **org.eclipse.jdt.core to calculate the similarity.dom**[4] library to generate the AST of the Java files. The AST is a hierarchical representation of the code structure, allowing a detailed analysis of the structural changes between different file versions. Then, for each Java file under analysis, we generated the AST using the **ASTParser** class of the

[2] https://pmd.github.io/
[3] See Appendix C
[4] https://eclipse.dev/jdt/

Table 15 – Metrics and attributes of code elements used in this work

Field	Description	Source
	Complex Aspect	
ss	Similarity score based on Abstract Syntax Tree (AST).	Our
cbo	Coupling between objects. Counts a class's input and output dependencies	CKTool
fan-in	Counts how many classes depend on a given class	CKTool
fan-out	Counts how many classes a given class uses	CKTool
dit	Counts the number of parents of a class in the inheritance tree.	CKTool
noc	Number of Children counts a class's direct subclasses.	CKTool
nosi	Counts the number of invocations to static methods	CKTool
rfc	Counts the number of unique method invocations in a class.	CKTool
wmc	Weight Method Class. Counts the number of branch instructions in a class.	CKTool
lcom	Lack of Cohesion of Methods, computes the lack of cohesion of class	CKTool
totalMethodsQty	Counts the number of all methods.	CKTool
staticMethodsQty	Counts the number of static methods.	CKTool
publicMethodsQty	Counts the number of public methods.	CKTool
privateMethodsQty	Counts the number of private methods.	CKTool
protectedMethodsQty	Counts the number of protected methods.	CKTool
defaultMethodsQty	Counts the number of default methods.	CKTool
visibleMethodsQty	Counts the number of visible methods.	CKTool
abstractMethodsQty	Counts the number of abstract methods.	CKTool
finalMethodsQty	Counts the number of final methods.	CKTool
synchronizedMethodsQty	Counts the number of synchronized methods.	CKTool
totalFieldsQty	Counts the number of all fields	CKTool
staticFieldsQty	Counts the number of static fields	CKTool
publicFieldsQty	Counts the number of public fields	CKTool
privateFieldsQty	Counts the number of private fields	CKTool
protectedFieldsQty	Counts the number of protected fields	CKTool
defaultFieldsQty	Counts the number of default fields	CKTool
finalFieldsQty	Counts the number of final fields	CKTool
loopQty	The number of loops like for, while, do while and enhanced for	CKTool
comparisonsQty	The number of comparisons == and !=	CKTool
tryCatchQty	The number of try/catches	CKTool
parenthesizedExpsQty	The number of expressions inside parenthesis	CKTool
stringLiteralsQty	The number of string literals	CKTool
numbersQty	The number of numbers literals int, long, double, float	CKTool
assignmentsQty	The number of same or different comparisons	CKTool
mathOperationsQty	Counts math operations.	CKTool
variablesQty	The number of declared variables	CKTool
maxNestedBlocksQty	The highest number of blocks nested together	CKTool
anonymousClassesQty	The quantity of anonymous classes	CKTool
innerClassesQty	The quantity of inner classes	CKTool
lambdasQty	The quantity of lambda expressions	CKTool
uniqueWordsQty	The algorithm counts words in a class, excluding Java keywords.	CKTool
typeAnonymous	Boolean indicating whether is an anonymous class	CKTool
typeClass	Boolean indicating whether is a class	CKTool
typeEnum	Boolean indicating whether is an enum	CKTool
typeInnerclass	Boolean indicating whether is an inner class	CKTool
typeInterface	Boolean indicating whether is an interface	CKTool
returnQty	The number of return instructions	CKTool
	Velocity Aspect	
locc	Counts lines of code changed, excluding empty lines and comments.	CKTool
foc7	Frequency of commits to a refactoring repository over 7 days	Our
foc14	Frequency of commits to a refactoring repository over 14 days.	Our
foc21	Frequency of commits to a refactoring repository over 21 days.	Our
foc28	Frequency of commits to a refactoring repository over 28 days.	Our
	Risk Aspect	
cs	Code Smell related to design, cohesion and modularity.	PMD
dcc	Cyclomatic complexity difference to evaluate new bug risk.	CKTool
Issues of Error Prone	Set of detected problems based on code patterns that are prone to errors	PMD
Issues of Multithreading	Set of detected problems based on parallel programming and multithreading	PMD
Issues of Performance	Set of problems detected with a focus on improving code performance	PMD
Issues of Security	Identifies and corrects security vulnerabilities in the code.	PMD
Total: 58		

Source: Prepared by the author.

library. The resulting AST captures the complete syntactic structure of the code, including class declarations, methods, variables, and expressions. Next, we implement the **MyVisitor** class, which inherits from **ASTVisitor**, to traverse the AST and collect information. Systematically visiting nodes allows us to build a complete representation of the code structure. Finally, we use the Jaccard (1901) similarity algorithm to calculate the similarity between two ASTs generated from different file commits. The Jaccard similarity index is defined as:

$$\text{Jaccard Similarity Index} = \frac{|A \cap B|}{|A \cup B|}$$

Where A and B represent the sets of nodes collected from the ASTs of two distinct code files. The numerator $|A \cap B|$ represents the number of common nodes between the two ASTs, while the denominator $|A \cup B|$ represents the total number of unique nodes present in both ASTs. The formula captures the essence of the similarity between two structures, returning a value between 0 and 1, where 0 indicates no similarity and 1 indicates that the structures are identical regarding the analyzed nodes.

The outputs of the CK and PMD tools, as well as the similarity scores and FoC metrics, were centralized into CSV files, which serve as the datasets for our ML experiments. After an initial data analysis, we selected 58 distinct features that proved most relevant for building the predictive models. These data now form the basis for the next phase of the study, where we explore the impact of code metrics on predicting the refactorings triviality.

Step 3: Developers' perception of the most relevant metrics. In conducting this study, we sought to understand which metrics developers consider most relevant in code refactoring activities. To this end, we conducted a survey with developers who work both internally and externally on the projects in question. The goal was to identify perceptions and practices related to refactoring metrics, as well as to obtain suggestions for improvements to the process.

First, we performed an automated collection of commit data using the GitHub API. We collected all emails available in commits from developers. The result of this collection was used to identify developers who frequently perform refactorings, who were later invited to participate in a detailed survey. 2,143 emails were sent, of which only 15 were responded to, approximately 1%. The typical rate was lower than that found in questionnaire-based software engineering surveys (Shull *et al.*, 2007). The developers were contacted to participate in an online survey[5]. The developers' profile can be seen on Table 16. The email explained the purpose of the survey, highlighting the benefits of creating a refactoring triviality index, which could assist in planning and executing refactorings in software projects.

Afterward, we structured the survey to cover three main aspects of the refactoring process: complexity, speed, and risk. Each aspect was addressed through questions that assessed

[5] See Appendix A

Table 16 – Developers Profile

ID	Experience (years)	Devevoper Roles	Formal Education	Code Refactoring	Project Internal
D1	36	Backend Developer, Fullstack, Devops, Tech Leader, Quality Assurance (QA), Project Manager, Stakeholder	High School	Always	YES
D2	7	Backend Developer	Vocational School	Often	YES
D3	40	Infrastructure developer.	Master's Degree	Often	YES
D4	15	Backend Developer	Master's Degree	Often	YES
D5	29	Developer in a Scientific Team	Master's Degree	Rarely	YES
D6	2	Frontend Developer	Graduation	Sometimes	NO
D7	10	Backend Developer, Tech Leader	Doctorate Degree	Often	NO
D8	6	Tech Leader, Scrum Master, Project Manager	Specialization	Sometimes	NO
D9	6	Frontend Developer, UX/UI Designer	Graduation	Often	NO
D10	10	Backend Developer, Tech Leader, Scrum Master	Master's Degree	Always	NO
D11	1	Backend Developer, Fullstack	Graduation	Never	NO
D12	6	Frontend Developer, Backend Developer, Fullstack	Master's Degree	Always	NO
D13	10	Fullstack, Devops, Tech Leader, Project Manager	Master's Degree	Sometimes	NO
D14	10	Operational Support Analyst	Graduation	Never	NO
D15	2	Fullstack	Technical Course	Sometimes	NO

Source: Prepared by the author.

the relevance of different metrics related to that specific topic. For the Complexity aspect, developers were asked about metrics such as FAN-IN, FAN-OUT, CBO (Coupling Between Objects), DIT (Depth of Inheritance Tree), among others. The goal was to understand which complexity metrics are considered most critical in the decision to refactor. For the Speed aspect, we addressed the frequency of commits and the size of changes in terms of lines of code. Metrics such as LOCC (Lines of Code Changed) and FOC (Frequency of Commit) were analyzed to understand how the speed of refactoring operations is perceived. For the Risk aspect, we addressed the likelihood of refactorings causing new problems or flaws in the code. Developers rated the importance of issues related to design practices, error-proneness, security, and performance.

Participation in the survey was voluntary and targeted at developers with experience in code refactoring. Participants were asked to rate the importance of each metric on a Likert (1932) scale, where 1 represented "not at all important" and 5 "very important". In addition, we asked for suggestions for other metrics or factors that could be considered relevant for improving software quality. Finally, we analyzed the collected data to identify opportunities for improvement in the refactoring practice, aiming not only to identify the most valued metrics but also to obtain insights into how different aspects of the code influence the decision to refactor.

Step 4: Applying supervised learning to predict refactoring triviality. Next, to better explain the machine learning process, we present the activities of data preprocessing, feature engineering, data normalization, data balancing, model training and evaluation. We must highlight that we added the refactoring operations that each commit received in the dataset.

Activity 1: Data Import and Preprocessing. We performed the activities for the 3 datasets containing metrics of the project's code classes: Apache, Eclipse and Random. First, we imported the data into a Jupiter notebook and configured the panda's display options to facilitate exploratory analysis. Initially, we treated missing data by replacing the values with the mean of the respective columns and corrected corrupted values, replacing invalid entries with more appropriate values. To deal with categorical data, we applied the one-hot encoding technique. Columns that represented operations or class types were converted to binary columns. This allowed machine learning models to process this information efficiently. We removed columns that did not contribute to refactoring prediction due to redundancy or irrelevance in the context of the problem studied. This selection was made based on suggestions from developers (in a second step to compare the results) and correlation analysis between variables.

Activity 2: Feature Engineering and Data Normalization. We calculated two additional metrics, LOCC (Lines of Code Changed) and DCC (Diff McCabe's Complexity), which represent the change in lines of code and cyclomatic complexity before and after a refactoring, respectively. We included these metrics as features in the model. We performed data normalization on a scale from 0 to 1. For metrics where smaller values are desirable, we applied inverse normalization. This step is crucial to ensure that all features are on the same scale, preventing variables with larger values from dominating the model training process.

In addition, our study introduces the concept of the Triviality Index. This index aims to quantify how trivial (simple, fast, and low-risk) the refactoring implementation is in software projects. To do this, we use the conceptualization of it as a composite metric that ranges from 0 to 1, where values close to 0 indicate more complex and high-risk refactorings, while values close to 1 indicate trivial refactorings. It is calculated considering three main aspects: complexity, velocity, and risk. Complexity includes metrics such as cyclomatic complexity, number of methods, number of classes, and depth of the inheritance tree. These metrics help to understand how intricate the code is, influencing the ease or difficulty of modifying it. As mentioned previously, velocity was measured, in part, by the frequency of commits during periods equivalent to sprints. A greater number of commits in a time interval may indicate that the code is more easily adaptable and less complex, allowing for rapid changes. Risk encompasses metrics related to the number of code smells detected and execution problems. These metrics indicate the risk associated with the changes.

To calculate the Triviality Index, each aspect is evaluated based on the specified metrics (Table 15). The algorithm that calculates the index takes the weighted arithmetic average of the three aspects. In this study, we consider the weight to be equal. Still, depending on the context, it may be helpful to consider different weights for each aspect, depending on its relevance to triviality in the specific context. With the normalized and weighted metrics, we calculate the index using the following formula:

$$\text{Triviality Index} = \frac{w_1 \cdot C + w_2 \cdot V + w_3 \cdot R}{w_{total}}$$

In this formula, C represents the *Complexity* aspect of the refactoring, V indicates the *Speed* aspect, and R the *Risk* aspect. The parameters w_1, w_2, and w_3 are the weights assigned to each of these aspects, reflecting the relative importance of each factor in the evaluation. The sum of these weighted products follows the multiplication of each aspect by its respective weight. To normalize the result and obtain a standardized metric, this sum is divided by the total of the weights w_{total}. The resulting Triviality Index provides a continuous scale indicating the degree of triviality of the refactoring, with higher values indicating simpler, faster, and less risky refactorings. This calculation provides a quantitative indication of the triviality of implementing the code refactoring.

Activity 3: Data Balancing. In this study, we use the SMOGN (Synthetic Minority Over-sampling Technique for Regression with Gaussian Noise) technique, as described by Branco *et al.* (2017), to balance the distribution of the triviality index data. SMOGN is a variant of SMOTER that adds Gaussian noise to the generated synthetic examples, which helps to simulate the natural variability of the data better and improves the robustness of the model. Unlike SMOTE, which was initially proposed by Chawla *et al.* (2002) to solve imbalance problems in classification tasks, SMOTER is an extension adapted for regression tasks. SMOTER generates synthetic examples for underrepresented areas of the continuous data, while SMOGN goes further by introducing Gaussian noise to promote an even more uniform data distribution.

This balancing technique was chosen because it is highly regarded in recent studies, showing effectiveness in dealing with unbalanced data distributions in regression problems (Song *et al.*, 2022; Torgo *et al.*, 2015; Branco *et al.*, 2017). This balancing was necessary to prevent the model from becoming biased towards refactorings with a more common triviality index and poor distribution of feature values in the collected data. In this way, we sought to make the evaluation of the model robust and generalizable in relation to the entire data set. The data was divided into two subsets: 80% for the training set (used to train the model) and 20% for the test set (used to validate and test the model). The trained models were based on logistic regression algorithms, including Linear Regression, Ridge and ElasticNet, and tree-based algorithms: Decision Tree, Random Forest, Gradient Boosting and XGBoost. We used the Scikit-learn library[6] and XGBoost[7] to implement these algorithms.

To optimize the performance of the models, we applied GridSearch and use number of cross-fold validations as five, which allowed us to tune and configure the hyperparameters of each algorithm efficiently. For **Linear Regression**, we adjusted the `fit_intercept` with binary value. **Ridge** regressions involved tuning `alpha` with range of values `[0.1, 1.0, 10.0, 100.0]` and `fit_intercept` with binary value. In the case of **ElasticNet**, the parameters included `alpha` with range of values `[0.1, 1.0, 10.0]`, `l1_ratio` with range of values `[0.1, 0.5, 0.9, 1.0]` and `fit_intercept` with binary value. The **Decision Tree** was fine-tuned using `max_depth` with range of values `[None, 10, 20, 30]`, `min_samples_split` with range of values `[2, 5, 10]`, and `min_samples_leaf` with range of values `[1, 2, 4]`. For the **Random Forest**, we explored `n_estimators` with range of values `[100, 200, 300]`, `max_depth` with range of values `[None, 10, 20, 30]`, `min_samples_split` with range of values `[2, 5, 10]` and `min_samples_leaf` with range of values `[1, 2, 4]`. The **Gradient Boosting** hyperparameters included `n_estimators` with range of values `[100, 200, 300]`, `learning_rate` with range of values `[0.01, 0.1, 0.2]`, `max_depth` with range of values `[3, 5, 10]`, `min_samples_split` with range of values `[2, 5, 10]` and `min_samples_leaf` with range of values `[1, 2, 4]`. Finally, the **XGBoost** was optimized by tuning `n_estimators` with range of values `[100, 200, 300]`, `max_depth` with range of values `[3, 5, 10]`, `eta` with range of values `[0.01, 0.1, 0.3]`, `subsample` with range of values `[0.5, 0.7, 1.0]`, `colsample_bytree` with range of values `[0.5, 1.0]`, `learning_rate` with range of values `[0.01, 0.1, 0.2]` and `min_child_weight` with range of values `[1, 2, 4]`. This technique was essential to ensure that the models were well-tuned and that there was no overfitting on the training data. Finally, we tested the models to predict the triviality index on the datasets.

Step 5: Evaluation of agreement between experts and predictive models. In this

[6] https://scikit-learn.org/stable/
[7] https://xgboost.readthedocs.io/en/stable/

step, we calculated the MSE, RMSE, MAE, MAPE and adjusted R² metrics to evaluate the trained models (Kuhn; Johnson, 2013; James *et al.*, 2023), comparing the results between algorithms and databases, both with and without prioritization of features by developers. In addition, we sought to identify the agreement of the solution with developers in the code refactoring activity. To this end, we conducted a survey with developers specialized in refactoring, to identify their perceptions about the triviality of implementing the main refactoring operations applied (Silva *et al.*, 2016).

We contacted the experts by email and social networks, and they were introduced to the research and invited to participate in an online survey[8]. The profile of experts developers can be seen in the Table 17. We explained the study's objective, highlighting the benefits of creating a refactoring triviality index, which could assist in the planning and execution of refactorings in software projects.

The survey was structured to cover, for each refactoring operation addressed, the three aspects of the refactoring process: complexity, speed, and risk. Participation was voluntary and targeted at code refactoring experts. Source codes with different code smells related to the application of 12 of the most commonly used refactoring operations were presented. 16 participants rated the impact of each aspect on a Likert scale Likert (1932), where 1 represented "Very difficult" and 5 "Very easy". In addition, they were allowed to provide additional information about the refactoring aspects presented. Finally, we analyzed the collected data statistically and compared them with the results obtained by the best supervised learning models.

Finally, we observed: (i) which metrics are most relevant to enable the creation of the triviality index; (ii) which algorithms produce the best results for our different data sets; (iii) how effective models with feature prioritization compared to models without prioritization; and (iv) agreement between the triviality index resulting from the ML model and the developers' perception of the triviality of the refactoring operation in each code example. This allowed us to answer our research questions. We present the results in the following section.

5.3 Results and Discussion

In the following subsections, we will address each of the Research Questions (RQs) presented in Section 5.2.1, discussing the results obtained in detail.

[8] See Appendix B

Table 17 – Experts developers profile

ID	Gender	Age	Devevoper Roles	Formal Education	Experience (years)	Code Refactoring
DX1	Man	25-34	Backend Developer, Product Owner, Tech Leader, Scrum Master, Project Manager	Master's Degree	1-3	Weekly
DX2	Man	25-34	Fullstack, Devops	Master's Degree	7-14	Weekly
DX3	Man	18-24	Fullstack	High School	1-3	Daily
DX4	Man	25-34	Backend Developer, Devops	Doctorate Degree	7-14	Daily
DX5	Man	18-24	Frontend Developer, Backend Developer, Fullstack	Graduation	1-3	Weekly
DX6	Woman	25-34	Backend Developer	Doctorate Degree	7-14	Weekly
DX7	Man	25-34	Frontend Developer,	Graduation	4-6	Weekly
DX8	Man	18-24	Fullstack	Graduation	4-6	Weekly
DX9	Man	35-44	Frontend Developer, Backend Developer, Devops	Graduation	7-14	Daily
DX10	Man	35-44	Frontend Developer, Backend Developer	Graduation	7-14	Monthly
DX11	Man	18-24	Fullstack	Technical Course	1-3	Annually
DX12	Man	25-34	Frontend Developer	Graduation	1-3	Monthly
DX13	Man	25-34	Fullstack, Devops, Tech Leader, Project Manager	Master's Degree	7-14	Monthly
DX14	Man	18-24	Backend Developer, Fullstack	Graduation	1-3	Monthly
DX15	Man	18-24	Backend Developer	High School	1-3	Monthly
DX16	Man	25-34	Backend Developer, Devops	Master's Degree	7-14	Monthly

Source: Prepared by the author.

5.3.1 *Which code metrics are considered most relevant by developers to determine the triviality of a refactoring operation? (RQ$_1$)*

To answer RQ$_1$, we considered the developers' opinions on code metrics. We also evaluated the key results by analyzing the average ratings for each metric, as well as the significant differences between the two groups of developers. Table 18 presents an overview of the evaluations of different code metrics conducted by both internal and external developers, with scores ranging from 1 to 5 on the Likert scale (Likert, 1932) of perceived importance.

The developers' experience levels range from 1 to 40 years, with the 46.7% having between 6 and 10 years of experience. 80% of developers have at least a graduate educational background. Familiarity with refactoring varies significantly: some developers are experts who use it routinely, while others have limited knowledge and rarely or never apply it. The frequency of code refactoring also shows variation, with 53.3% developers performing it often or always, while 13.2% claims to have never refactored. 33.3% execute rarely or sometimes.

The developers hold various roles, including back-end with 21.2%, full-stack developers with 15.1%, tech leads with 15.1%, front-end with 9.1%, project managers with 9.1%, DevOps engineers with 6%, and other roles only counted once. This diversity of roles suggests that the need for and application of refactoring can vary significantly across different teams and software projects. Moreover, their formal education levels range from high school to PhDs, which may further influence their exposure to and approach to software refactoring.

In summary, our findings indicate that external developers tend to place higher value on code metrics related to structural complexity, best practices, performance, and security. They generally provided more positive ratings across most metrics, which may reflect a heightened level of caution or a more critical perspective on code quality. In contrast, internal developers appeared more conservative in their evaluations, possibly due to their familiarity with the code assessed in the study. These differences suggest that perceptions of the need for refactoring can vary significantly based on a developer's experience and context.

The developers' responses offer valuable insights into their experiences and perspectives on code refactoring. We list the key points raised by the developers as follows:

- (D1) Emphasize the importance of Test-Driven Development (TDD) and Agile methodologies, combined with continuous refactoring, as best practices;
- (D2) Highlight the importance of comprehensive integration tests to maintain the quality and speed of refactoring while avoiding the introduction of new errors;
- (D3) The complexity of existing code as the primary risk factor in refactoring is often more significant than the type of fix applied;
- (D7) Associates refactoring primarily with improving code execution time;
- (D9) Acknowledge the frequent challenge of managing legacy code, which often requires rewriting or updating to meet modern standards;
- (D11) Suggest incorporating more specific time metrics, such as "hours dedicated to refactoring activities," to better evaluate the efficiency of the refactoring process;
- (D13). Use refactoring to improve code quality and ensure adherence to standards set by static analysis tools;

The key points raised by developers reveal their recognition of refactoring's importance for software quality. They highlight its benefits in improving code performance, readability, and maintainability. They also emphasize the importance of agile practices, and the need for a flexible, continuous approach to refactoring. Additionally, they note the interdependence between testing, code complexity, and risk management. In this context, refactoring is viewed not merely as a technical task, but as a critical practice for maintaining software quality. This insight improves our understanding of refactoring in practice and offers valuable perspectives for researchers and developers aiming to enhance software quality. It also provides a clearer picture of how developers perceive the importance of refactoring.

Finding 1: Developers reveal their perceptions, highlighting an intersection between refactoring, testing, code complexity, and best practices.

By looking at Table 18, we can observe that metrics with the highest scores (above 4.0) are primarily associated with Risk and Complexity aspects. This suggests that developers consider these factors to be the most critical when evaluating the necessity and impact of refactoring. Both internal and external developers considered metrics related to security and class coupling as the most important. In particular, *Issues of Security* and CBO (*Coupling Between Object Classes*) stood out as the most valued metrics, with average scores of 4.40 and 4.30, respectively. We also observed that *Issues of Security* was unanimously considered as essential, with external developers assigning it an even higher importance rating (4.80). Additionally, the CBO metric received high ratings from both groups, reflecting a common concern about coupling complexity and its influence on the need for refactoring.

Conversely, the metric NOSI (*Number of Static Invocations*) was considered as the least important, with the lowest overall average among all evaluated metrics. With an average score of 2.30, and particularly a very low rating from internal developers (1.60), NOSI was identified as the least relevant metric in the refactoring decision process. Additionally, other metrics such as *Quantity of Number* (2.50), *Number of Unique Words* (2.55), *Quantity of Math Operations* (2.65), and *File Type* (2.65) also received low ratings. These metrics, which focus on basic and repetitive aspects of code, are viewed as less significant for justifying refactoring.

From these observations, it is evident that developers placed a high priority on metrics such as *Issues of Security* and *CBO*, which suggests that developers prioritize factors that can significantly impact system integrity and architecture. Conversely, metrics related to static invocations and basic code elements are viewed as less impactful. This suggests that, in practice, refactoring decisions are driven more by long-term considerations, such as preventing vulnerabilities and reducing complexity, rather than by speed-related factors.

Regarding metrics related to Speed, although they are less emphasized compared to Risk and Complexity, they still received considerable attention. Among them, FOC7 (*Frequency of Change in 7 Days*) was the most valued speed metric, with an average score of 3.47. This indicates that the frequency of code changes over a short period is a significant factor in assessing refactoring efficiency. Other speed metrics, including LOCC (*Lines of Code Changed*) with an average of 3.20, FOC14 with 3.27, FOC21 with 3.20, and FOC28 with 3.07. These scores suggest that while speed is considered, it is not the primary criterion for justifying refactoring but rather a supplementary factor.

In summary, developers place significant importance on metrics that assess Risk and Complexity, with a particular focus on security issues, cohesion, coupling, and structural complexity. These metrics are regarded as crucial for maintaining software quality and mitigating issues that may arise during development and maintenance. Developers' perceptions highlight the need to prioritize practices that minimize risk and complexity, ensuring that code remains secure, efficient, and maintainable.

Finding 2: Metrics related to risk and complexity are considered the most important for developers when evaluating code quality.

Several metrics received an average score below 3, suggesting that developers consider them less relevant when deciding whether to refactor or not. The metrics that did not reach a score of 3, which represents moderate importance, include: NOSI (*Number of Static Invocations*) with an average of 2.30; *Quantity of Number* with 2.50; *Number of Unique Words* with 2.55; *Quantity of Math Operations* with 2.65; *File Type* with 2.65; *Quantity of Modifiers* with 2.65; *String Literals* with 2.70; *Number of Log Statements* with 2.75; *Inner Classes* with an average of 2.80; *Path of Class* with 2.90; *Quantity of Returns* with 2.90;*Quantity of Parenthesized Expressions* with 2.90; and *RFC* with an average of 2.85.

Metrics such as *Quantity of Number*, *Number of Unique Words*, and *Quantity of Math Operations*, all related to counting basic elements, received averages below 2.70, indicating that the presence of these elements alone is insufficient to justify refactoring. Similarly, *File Type* and *Quantity of Modifiers*, both with an average of 2.65, suggest that developers view file type and the number of modifiers as minor factors that do not significantly impact refactoring decisions.

Overall, these metrics represent characteristics that, in the developers' perception, have a smaller impact on code quality and are therefore less prioritized when assessing the need for refactoring. This highlights a greater focus on more critical and structural aspects of the code, such as security and coupling, which are more frequently associated with maintainability and system robustness.

Table 18 – Developer perspectives on the importance of code metrics

METRICS	Internal Dev	External Dev	AVERAGE
Issues of Security	4.00	4.80	4.40
CBO	4.00	4.60	4.30
Issues of Error Proneness	4.00	4.00	4.00
Quantity of Fields	4.00	4.00	4.00
LCOM	3.60	4.30	3.95
Max Nested Blocks	3.80	4.10	3.95
Issues of Design	3.20	4.40	3.80
Quantity of Loops	3.00	4.60	3.80
Issues of Performance	3.20	4.30	3.75
Usage of Each Field	3.40	4.10	3.75
Usage of Each Variable	3.40	4.10	3.75
Issues of Best Pratices	3.20	4.20	3.70
Quantity of Comparisons	3.20	4.20	3.70
Issues of Multithreading	3.20	4.10	3.65
FAN-IN	3.00	4.20	3.60
TCC	3.40	3.80	3.60
NOC	3.20	3.90	3.55
Number of Visible Method	3.40	3.70	3.55
FAN-OUT	3.00	4.00	3.50
DIT	2.80	4.20	3.50
WMC	3.40	3.60	3.50
LCC	3.00	4.00	3.50
FOC7	3.60	3.40	3.50
Quantity of Try/Catches	3.00	4.00	3.50
Number of Method	3.00	3.80	3.40
FOC14	3.40	3.20	3.30
Quantity of Variables	3.00	3.60	3.30
Quantity Method Invocations	2.40	4.10	3.25
FOC21	3.20	3.20	3.20
Quantity of Anonymous Classes	2.60	3.70	3.15
LOCC	2.80	3.40	3.10
FOC28	3.00	3.10	3.05
Lambda Expressions	2.40	3.60	3.00
Quantity of Returns	2.20	3.60	2.90
Quantity of Parenthesized Expressions	2.20	3.60	2.90
Path of Class	2.40	3.40	2.90
RFC	2.40	3.30	2.85
Inner Classes	1.80	3.80	2.80
Number of Log Statements	2.00	3.50	2.75
String Literals	2.00	3.40	2.70
File Type	2.20	3.10	2.65
Quantity of Modifiers	2.20	3.10	2.65
Quantity of Math Operations	1.80	3.50	2.65
Number of Unique Words	2.00	3.10	2.55
Quantity of Number	1.80	3.20	2.50
NOSI	1.60	3.00	2.30

Source: Prepared by the author.

Finding 3: 28.2% of the code metrics assessed using the Likert scale are considered of low importance by developers in refactoring decisions.

The analysis of software metric evaluations provided by internal and external developers highlights significant differences in their perceptions. External developers tend to rate metrics such as *Issues of Security* (4.80 vs 4.00), *CBO* (4.60 vs 4.00), and *LCOM* (4.30 vs 3.60) more positively compared to internal developers. These discrepancies may suggest that external developers place greater emphasis on, or adopt a more critical stance toward, aspects of code security, complexity, and cohesion.

In contrast, metrics such as *Issues of Best Practices* and *Quantity of Returns* received higher ratings from external developers, which may reflect a greater emphasis on coding practices and return management in their development approach. The overall average rating from internal developers is 2.90, compared to 3.78 from external developers, indicating that external developers generally rate these metrics more positively.

Additionally, we observed the largest discrepancies in the ratings of metrics such as *DIT* (4.20 vs. 2.80), *Quantity of Loops* (4.60 vs. 3.00), and *Quantity of Method Invocations* (4.10 vs. 2.40). These differences suggest that external developers perceive aspects related to class hierarchy depth, loop count, and method invocation complexity more critically than internal developers. These variations may stem from differences in development methodologies, regional practices, and development cultures, potentially explaining the more detailed and rigorous approach taken by external developers.

On the other hand, metrics such as *FOC7* and *FOC14*, which measure commit frequency over 7 and 14 days, respectively, received slightly higher ratings from internal developers compared to external ones. For *FOC7*, internal developers rated it an average of 3.60, while external developers rated it 3.40. Similarly, for *FOC14*, internal developers rated it 3.40, whereas external developers rated it 3.20. This suggests that internal developers may consider commit frequency over short periods as a more critical or relevant factor for code quality. These differences in ratings highlight how developers' experience and focus can influence their perception and prioritization of various aspects of the code.

Finding 4: Small divergences in metric perceptions between internal and external developers, highlighting the greater scrutiny from external developers.

Implications for RQ$_1$. Both internal and external developers show strong agreement on prioritizing metrics related to risk and complexity aspects, such as security and class coupling (CBO), as crucial factors for refactoring. The high emphasis placed on these metrics underscores the importance of system integrity and structural complexity in determining the need for refactoring. Additionally, developers emphasize the importance of agile practices, an adaptable and continuous approach to refactoring, and the interdependence between testing, code complexity, and risk management.

Metrics with lower scores, such as *NOSI* and other metrics related to basic code elements like *Quantity of Number* and *Number of Unique Words*, are considered less relevant by both groups of developers. This low prioritization indicates that more superficial aspects of the code are not considered decisive factors for the need to refactor, reinforcing the focus on structural and security issues. External developers tend to provide more critical evaluations of metrics, particularly for metrics such as *DIT*, *Quantity of Loops*, and *Quantity of Method Invocations*. This heightened scrutiny may be attributed to a broader perspective that is less influenced by the specific project context.

5.3.2 How do different ML techniques behave in predicting the code refactorings triviality index? (RQ_2)

To address RQ_2, we analyzed all datasets and assessed the performance of various ML algorithms in predicting the triviality index. Table 19 provides important insights into the average performance of different ML algorithms across different metrics and configurations used in this study. The values in the table are highlighted on a grayscale, where the intensity of the color increases as values approach the highest and lowest extremes. The average analysis of performance metrics offers an overview of the strengths and limitations of each model, highlighting those that are effective for predicting the triviality index of code refactorings in the context of our study.

In summary, our results indicate that tree-based models *(Random Forest*, *Gradient Boosting*, *XGBoost*) are the most effective for predicting the triviality index of code refactorings. These models achieved the lowest mean squared error (MSE) values, averaging around 0.0016 for Random Forest, Gradient Boosting, and XGBoost, compared to 0.0019 for Decision Tree, indicating high prediction accuracy. The root mean squared error (RMSE) was similarly low, with average values of 0.0384 for Gradient Boosting and 0.0386 for Random Forest and XGBoost. Moreover, the mean absolute errors (MAE) were lower for these models, approximately 0.0125 for Random Forest and 0.0132 for Gradient Boosting. Regarding the mean absolute percentage error (MAPE), these models recorded average values of 2.0474% for the Random Forest and

2.0841% for the Decision Tree, underscoring their superior performance with minimal percentage deviation.

The average R^2 coefficients were high, with Gradient Boosting reaching an average of 0.9273 and XGBoost approximately 0.9270, indicating that these models explain over 92% of the variability in the data. The average adjusted R^2 was similarly high, with XGBoost at 0.9285, Random Forest at 0.9246, and Gradient Boosting at 0.9245, reinforcing the suitability of these models for the context of the study. In contrast, the Elastic Net model demonstrated lower performance, with average MSE values around 0.0040, RMSE of 0.0628, MAE of 0.0354, MAPE of 6.5349%, R^2 of 0.8188, and adjusted R^2 of 0.7979. These metrics suggest that Elastic Net may not be the optimal choice for predicting the triviality index.

These results highlight the effectiveness of tree-based models, such as Gradient Boosting, particularly when used with a comprehensive set of metrics, to enhance the accuracy and robustness of predictions in code refactoring environments.

Ridge regularization is a technique used to reduce model complexity and prevent overfitting by penalizing regression coefficients with large magnitudes (McDonald, 2009). However, the analysis of the average results across datasets suggests that Ridge regularization did not have a significant impact on improving predictions for these datasets.

Considering the average metrics for Linear Regression and the Ridge model, we observed that both models exhibited an average MSE of approximately 0.0027, suggesting that Ridge regularization did not significantly improve prediction accuracy. The average RMSE differed by only 0.0001 between the two models, indicating no difference between the two models. This reinforces the conclusion that Ridge regularization did not provide a meaningful enhancement to prediction accuracy.

The average MAE was approximately 0.023 for both models, suggesting that Ridge regularization did not impact the model's ability to predict average values. The average MAPE was around 4.1 for both models, indicating that the relative proportion of error to the true values remained consistent. Both models had an average R^2 of about 0.879, suggesting they explained a similar amount of variance in the data. The average adjusted R^2 was approximately 0.866, which confirms that Ridge regularization did not improve the model's ability to fit the data when adjusted for the number of predictors. It is important to note that while Ridge regularization is an effective tool for mitigating multicollinearity and overfitting in high-dimensional datasets, its impact can vary based on the specific characteristics of the data and the fit of the model to the problem.

Table 19 – Table of machine learning model results

		APACHE		ECLIPSE		RANDOM		
Model	**Metric**	**DEV**	**ALL**	**DEV**	**ALL**	**DEV**	**ALL**	**AVG**
LinearRegression	mse	0.0033	0.0023	0.0032	0.0020	0.0032	0.0019	0.0027
	rmse	0.0577	0.0475	0.0566	0.0451	0.0564	0.0436	0.0512
	mae	0.0297	0.0187	0.0255	0.0182	0.0273	0.0209	0.0234
	mape	5.4001	3.3257	4.4645	3.4268	4.3312	3.7205	4.1115
	r2	0.8369	0.8861	0.8528	0.8999	0.8792	0.9202	0.8792
	r2a	0.8359	0.8810	0.8547	0.8918	0.8596	0.8745	0.8663
Ridge	mse	0.0033	0.0023	0.0032	0.0020	0.0032	0.0019	0.0027
	rmse	0.0577	0.0475	0.0566	0.0451	0.0565	0.0434	0.0511
	mae	0.0296	0.0187	0.0254	0.0181	0.0265	0.0203	0.0231
	mape	5.3791	3.3214	4.4495	3.4193	4.1952	3.5737	4.0564
	r2	0.8373	0.8861	0.8529	0.8999	0.8785	0.9207	0.8792
	r2a	0.8360	0.8810	0.8547	0.8917	0.8592	0.8741	0.8661
ElasticNet	mse	0.0047	0.0037	0.0044	0.0033	0.0047	0.0030	0.0040
	rmse	0.0685	0.0611	0.0666	0.0571	0.0688	0.0545	0.0628
	mae	0.0409	0.0377	0.0381	0.0365	0.0318	0.0271	0.0354
	mape	7.7201	6.9908	7.0111	6.9241	5.3722	5.1912	6.5349
	r2	0.7704	0.8111	0.7964	0.8397	0.8199	0.8753	0.8188
	r2a	0.7673	0.8041	0.8003	0.8323	0.7860	0.7974	0.7979
DecisionTree	mse	0.0025	0.0019	0.0029	0.0019	0.0012	0.0007	0.0019
	rmse	0.0502	0.0434	0.0537	0.0432	0.0350	0.0270	0.0421
	mae	0.0167	0.0140	0.0179	0.0129	0.0084	0.0061	0.0127
	mape	2.7332	2.4028	2.8854	2.1549	1.2251	1.1033	2.0841
	r2	0.8768	0.9050	0.8670	0.9082	0.9535	0.9694	0.9133
	r2a	0.8825	0.9057	0.8716	0.9098	0.9526	0.9481	0.9117
RandomForest	mse	0.0022	0.0015	0.0024	0.0017	0.0008	0.0007	0.0016
	rmse	0.0467	0.0393	0.0494	0.0411	0.0285	0.0268	0.0386
	mae	0.0170	0.0124	0.0183	0.0135	0.0075	0.0064	0.0125
	mape	2.7654	2.0721	3.0298	2.2836	1.0637	1.0695	2.0474
	r2	0.8934	0.9219	0.8873	0.9168	0.9691	0.9698	0.9264
	r2a	0.8983	0.9222	0.8896	0.9190	0.9610	0.9577	0.9246
GradientBoosting	mse	0.0021	0.0016	0.0024	0.0017	0.0008	0.0007	0.0016
	rmse	0.0462	0.0394	0.0490	0.0409	0.0275	0.0273	0.0384
	mae	0.0180	0.0130	0.0191	0.0144	0.0085	0.0064	0.0132
	mape	3.0005	2.2104	3.1916	2.4536	1.3346	0.9938	2.1974
	r2	0.8955	0.9216	0.8893	0.9177	0.9713	0.9686	0.9273
	r2a	0.8988	0.9221	0.8902	0.9183	0.9590	0.9584	0.9245
XGBoost	mse	0.0022	0.0015	0.0024	0.0017	0.0008	0.0008	0.0016
	rmse	0.0464	0.0390	0.0490	0.0411	0.0281	0.0279	0.0386
	mae	0.0188	0.0135	0.0199	0.0145	0.0087	0.0076	0.0138
	mape	3.1591	2.3081	3.1945	2.4680	1.2676	1.1461	2.2572
	r2	0.8946	0.9231	0.8901	0.9169	0.9700	0.9672	0.9270
	r2a	0.8984	0.9236	0.8950	0.9172	0.9599	0.9605	0.9258

Source: Prepared by the author.

Finding 5: Ridge regularization did not have a significant impact on improving predictions for the datasets used in our study: Apache, Eclipse, and Random.

The Linear Regression and Ridge models demonstrated moderate performance compared to the other algorithms. Both models achieved an MSE of approximately 0.0027, indicating a reasonable level of accuracy, but lower than tree-based models such as Random Forest and Gradient Boosting. The RMSE and MAE for these models averaged 0.0512 and 0.0234 for Linear Regression, and 0.0511 and 0.0231 for Ridge, indicating that these algorithms are less effective at capturing variability in the data.

The average R² for both models was 0.8792, with an adjusted R² of 0.8663 for Linear Regression and 0.8661 for Ridge, reflecting reasonable explanatory power. However, this is still below that of more advanced models, which achieved R² values of 0.9133 or higher, demonstrating superior performance. The MAPE metric, which indicates the mean absolute percentage error, was 4.1115% for Linear Regression and 4.0564% for Ridge, indicating reasonable accuracy, though not as robust as the tree-based models.

Finding 6: While Linear Regression and Ridge Regression models demonstrated moderate performance, they were outperformed by tree-based models advanced algorithms such as Decision Tree, Random Forest, Gradient Boosting, and XGBoost.

The Elastic Net model exhibited the worst average performance among all the algorithms analyzed. The average MSE was 0.0040, the highest among the models, indicating greater inaccuracy in the predictions. Additionally, the average RMSE and MAE were also higher, with values of 0.0628 and 0.0354, respectively, suggesting that the model is less effective at capturing data variability compared to the other algorithms. The average R² for Elastic Net was 0.8188, and the adjusted R² was 0.7979, both the lowest among the models studied, indicating that this algorithm explains less of the variability in the data. Additionally, the average MAPE was 6.5349%, the highest among the models, reflecting a higher percentage error in predictions. These results suggest that Elastic Net may not be the most suitable choice for predicting the triviality index of code refactorings in this specific context.

Finding 7: In our study, the Elastic Net model demonstrated the poorest performance.

Tree-based and ensemble models, such as Random Forest, Gradient Boosting, and XGBoost, emerged as the most effective across all aspects. They achieved the lowest average MSE of 0.0016, demonstrating excellent prediction accuracy even in the presence of outliers. The

RMSE and MAE metrics were consistently low, with average values of 0.0386 and 0.0125 for Random Forest, 0.0384 and 0.0132 for Gradient Boosting, and 0.0386 and 0.0138 for XGBoost. These results indicate that these models are highly proficient in capturing data variability and providing accurate and consistent predictions.

The average R² values were 0.9264 for Random Forest, 0.9273 for Gradient Boosting, and 0.9270 for XGBoost, demonstrating that these models account for over 92% of the data variability. The average adjusted R² values were 0.9246 for Random Forest, 0.9245 for Gradient Boosting, and 0.9258 for XGBoost, highlighting their robustness and appropriateness for the problem. The average MAPE was 2.0474% for Random Forest, 2.1974% for Gradient Boosting, and 2.2572% for XGBoost. These values are significantly lower compared to the regression models and Elastic Net, indicating minimal percentage errors and high prediction accuracy.

Finding 8: Tree-based and ensemble models –specifically Random Forest, Gradient Boosting, and XGBoost – demonstrated the best performance.

XGBoost, another tree-based model, demonstrated performance very close to that of the Random Forest and Gradient Boosting models. With an average MSE of 0.0016, it matched the accuracy of the other tree-based models. The average RMSE was 0.0386, and the average MAE was 0.0138, both reflecting effective capture of data variability. The average R² was 0.9270, and the average adjusted R² was 0.9258, demonstrating the highest explanatory power among the models, with the incorporation of new features enhancing this capability. The average MAPE was 2.2572%, slightly higher than that of the other tree-based models but still indicating high prediction accuracy. This places XGBoost among the top models, although it falls behind Random Forest and Gradient Boosting in certain metrics.

Finding 9: The XGBoost model presented the best average explanatory power among the models, with improved performance due to the addition of new features.

In the Random ecosystem, tree-based models, particularly Random Forest, Gradient Boosting, and XGBoost, exhibited efficient and consistent performance across error metrics (MSE, RMSE, MAE, MAPE) and determination coefficients (R² and adjusted R²). Random Forest achieved the lowest MAPE in both datasets (DEV and ALL), with 1.0637% and 1.0695%, respectively, indicating the smallest percentage variation in errors relative to the true values.

Gradient Boosting obtained the lowest RMSE in the DEV dataset (0.0275), indicating a slight edge in root mean squared error precision, though all three models showed very close performance. XGBoost exhibited an excellent balance between accuracy and explanatory power, with an adjusted R^2 of 0.9605 in the ALL dataset, the highest among the three models, suggesting that it is somewhat more effective at incorporating new features.

These results highlight that, within the Random ecosystem, tree-based and ensemble models like Random Forest, Gradient Boosting, and XGBoost are exceptionally effective. They not only minimize prediction errors but also exhibit strong explanatory power, demonstrating robustness in both development (DEV) and complete (ALL) datasets. This consistency indicates that these models are suitable for applications where accuracy is critical.

Finding 10: Tree-based and ensemble models demonstrated superior performance within the Random ecosystem.

Implication for RQ_2. The results indicate that tree-based and ensemble models, particularly Random Forest, Gradient Boosting, and XGBoost, are the most effective for predicting the triviality index of code refactorings in the context of our study. These models not only achieved the lowest error metrics but also exhibited strong explanatory power, effectively capturing data variability. In contrast, Elastic Net was less effective, suggesting that it may not be the optimal choice for the data types and issues addressed in this study.

These findings have important implications for selecting models in contexts where prediction accuracy and robustness are critical. Employing tree-based and ensemble models can substantially enhance prediction quality in code refactoring environments, especially when managing large datasets and capturing complex nuances in predictor variables. Additionally, the analysis highlights the necessity of choosing the right model for the specific problem, taking into account not just error metrics but also the model's explanatory power and robustness.

5.3.3 What is the impact of prioritizing features ranked by developers on the effectiveness of triviality index prediction models? (RQ_3)

To answer RQ_3, we compared the performance of ML models configured with features prioritized by developers (DEV) against those using all collected features (ALL). We assessed how these approaches influence the prediction of the triviality index for code refactorings.

Table 20 presents the results obtained for the applied models. The analysis of the data reveals that the difference between using all features (ALL) and only the prioritized features (DEV) provides insights into the effectiveness of feature prioritization. Overall, the results show that using all features (ALL) often resulted in better model performance, as evidenced by reduced error metrics such as MSE, RMSE, MAE, and MAPE, as well as an increase in the R^2 and adjusted R^2 coefficients.

This trend suggests that while the features prioritized by developers are valuable, they may not fully capture the complexity of the data as effectively as including all available variables. Incorporating a comprehensive set of features can offer a more detailed understanding of the data, contributing to more accurate and robust models for predicting the triviality index.

Therefore, the results indicate that the approach using all collected features tends to enhance the effectiveness of prediction models, offering superior performance in predicting the triviality index of code refactorings. This finding highlights the importance of a comprehensive analysis of features to optimize machine learning model performance and suggests that feature prioritization should be done carefully to ensure that critical information is not omitted.

The reduction in MSE was consistent across all three ecosystems when using all features instead of only those prioritized by developers. The most significant reduction was observed in the Random ecosystem for the Elastic Net model, with a decrease of -0.0017. This suggests that incorporating all features may enhance accuracy by including additional information that could have been overlooked during developer selection. Furthermore, RMSE values were also lower when all features were used compared to the DEV features. In the Random ecosystem for the Elastic Net model, the difference was -0.0143. The analysis indicates that using all available features can reduce the mean squared error of the models, potentially increasing prediction accuracy by capturing more relevant variables.

Finding 11: Incorporating all available features can reduce the mean squared error of the models, leading to improved prediction accuracy.

The MAE was lower when all features were used, with the most decrease in the Linear Regression model: -0.0110, -0.0073, and -0.0064 for the Apache, Eclipse, and Random ecosystems, respectively. This indicates that predictions are generally closer to the actual values when all features are included. Additionally, the MAPE showed a more substantial decrease, especially in the Linear Regression and Ridge models within the Apache ecosystem, with

Table 20 – Results of impact of selected metrics by developer

		APACHE			ECLIPSE			RANDOM		
Model	**Metric**	**DEV**	**ALL**	**DIFF**	**DEV**	**ALL**	**DIFF**	**DEV**	**ALL**	**DIFF**
LinearRegression	mse	0.0033	0.0023	-0.0010	0.0032	0.0020	-0.0012	0.0032	0.0019	-0.0013
	rmse	0.0577	0.0475	-0.0102	0.0566	0.0451	-0.0115	0.0564	0.0436	-0.0128
	mae	0.0297	0.0187	-0.0110	0.0255	0.0182	-0.0073	0.0273	0.0209	-0.0064
	mape	5.4001	3.3257	-2.0744	4.4645	3.4268	-1.0377	4.3312	3.7205	-0.6107
	r2	0.8369	0.8861	0.0492	0.8528	0.8999	0.0471	0.8792	0.9202	0.0410
	r2a	0.8359	0.8810	0.0451	0.8547	0.8918	0.0371	0.8596	0.8745	0.0149
Ridge	mse	0.0033	0.0023	-0.0010	0.0032	0.0020	-0.0012	0.0032	0.0019	-0.0013
	rmse	0.0577	0.0475	-0.0102	0.0566	0.0451	-0.0115	0.0565	0.0434	-0.0131
	mae	0.0296	0.0187	-0.0109	0.0254	0.0181	-0.0073	0.0265	0.0203	-0.0062
	mape	5.3791	3.3214	-2.0577	4.4495	3.4193	-1.0302	4.1952	3.5737	-0.6215
	r2	0.8373	0.8861	0.0488	0.8529	0.8999	0.0470	0.8785	0.9207	0.0422
	r2a	0.8360	0.8810	0.0450	0.8547	0.8917	0.0370	0.8592	0.8741	0.0149
ElasticNet	mse	0.0047	0.0037	-0.0010	0.0044	0.0033	-0.0011	0.0047	0.0030	-0.0017
	rmse	0.0685	0.0611	-0.0074	0.0666	0.0571	-0.0095	0.0688	0.0545	-0.0143
	mae	0.0409	0.0377	-0.0032	0.0381	0.0365	-0.0016	0.0318	0.0271	-0.0047
	mape	7.7201	6.9908	-0.7293	7.0111	6.9241	-0.0870	5.3722	5.1912	-0.1810
	r2	0.7704	0.8111	0.0407	0.7964	0.8397	0.0433	0.8199	0.8753	0.0554
	r2a	0.7673	0.8041	0.0368	0.8003	0.8323	0.0320	0.7860	0.7974	0.0114
DecisionTree	mse	0.0025	0.0019	-0.0006	0.0029	0.0019	-0.0010	0.0012	0.0007	-0.0005
	rmse	0.0502	0.0434	-0.0068	0.0537	0.0432	-0.0105	0.0350	0.0270	-0.0080
	mae	0.0167	0.0140	-0.0027	0.0179	0.0129	-0.0050	0.0084	0.0061	-0.0023
	mape	2.7332	2.4028	-0.3304	2.8854	2.1549	-0.7305	1.2251	1.1033	-0.1218
	r2	0.8768	0.9050	0.0282	0.8670	0.9082	0.0412	0.9535	0.9694	0.0159
	r2a	0.8825	0.9057	0.0232	0.8716	0.9098	0.0382	0.9526	0.9481	-0.0045
RandomForest	mse	0.0022	0.0015	-0.0007	0.0024	0.0017	-0.0007	0.0008	0.0007	-0.0001
	rmse	0.0467	0.0393	-0.0074	0.0494	0.0411	-0.0083	0.0285	0.0268	-0.0017
	mae	0.0170	0.0124	-0.0046	0.0183	0.0135	-0.0048	0.0075	0.0064	-0.0011
	mape	2.7654	2.0721	-0.6933	3.0298	2.2836	-0.7462	1.0637	1.0695	0.0058
	r2	0.8934	0.9219	0.0285	0.8873	0.9168	0.0295	0.9691	0.9698	0.0007
	r2a	0.8983	0.9222	0.0239	0.8896	0.9190	0.0294	0.9610	0.9577	-0.0033
GradientBoosting	mse	0.0021	0.0016	-0.0005	0.0024	0.0017	-0.0007	0.0008	0.0007	-0.0001
	rmse	0.0462	0.0394	-0.0068	0.0490	0.0409	-0.0081	0.0275	0.0273	-0.0002
	mae	0.0180	0.0130	-0.0050	0.0191	0.0144	-0.0047	0.0085	0.0064	-0.0021
	mape	3.0005	2.2104	-0.7901	3.1916	2.4536	-0.7380	1.3346	0.9938	-0.3408
	r2	0.8955	0.9216	0.0261	0.8893	0.9177	0.0284	0.9713	0.9686	-0.0027
	r2a	0.8988	0.9221	0.0233	0.8902	0.9183	0.0281	0.9590	0.9584	-0.0006
XGBoost	mse	0.0022	0.0015	-0.0007	0.0024	0.0017	-0.0007	0.0008	0.0008	0.0000
	rmse	0.0464	0.0390	-0.0074	0.0490	0.0411	-0.0079	0.0281	0.0279	-0.0002
	mae	0.0188	0.0135	-0.0053	0.0199	0.0145	-0.0054	0.0087	0.0076	-0.0011
	mape	3.1591	2.3081	-0.8510	3.1945	2.4680	-0.7265	1.2676	1.1461	-0.1215
	r2	0.8946	0.9231	0.0285	0.8901	0.9169	0.0268	0.9700	0.9672	-0.0028
	r2a	0.8984	0.9236	0.0252	0.8950	0.9172	0.0222	0.9599	0.9605	0.0006

Source: Prepared by the author.

reductions of -2.0744 and -2.0577, respectively. This indicates that percentage errors were reduced with the use of all features, suggesting that feature limitations can be particularly detrimental in contexts where percentage accuracy is critical. The improvement in MAE and MAPE further indicates that using all features can result in more accurate predictions in absolute terms, aligning them more closely with the actual values.

Finding 12: Incorporating all features can improve model efficiency by producing predictions that are closer to the actual values and by reducing percentage errors.

Linear models, such as Linear Regression, Ridge, and Elastic Net, are more sensitive to feature selection. When all features (ALL) are used, these models often exhibit higher R² and adjusted R² values, indicating that they are better at capturing data variability. This is because linear models rely heavily on including all relevant variables to explain the relationships between independent and dependent variables.

Additionally, the R² metric increased with the use of all features, with an average of 4.2%, 4.4%, and 2.5% in the Apache, Eclipse, and Random ecosystems, respectively. In the Random ecosystem, the Elastic Net model showed the largest increase, with a 6.8% in R². Moreover, there was a similar and positive percentage increase in the adjusted R² metric. Of the 21 trained models, 18 showed an improvement in adjusted R² with the use of all features, suggesting that the increase in the number of features was beneficial, even when accounting for the penalty of greater model complexity. This means that 85.7% of the models performed better with the inclusion of all features (ALL).

These results indicate that, in most cases, using all features led to a better explanation of data variability, both in terms of R² and adjusted R², demonstrating that including more features was beneficial for model performance.

Finding 13: Models with more features provided a better explanation of data variability, effectively mitigating the penalties associated with increased feature count and complexity.

We can observe in Table 18 that the effectiveness of using all features compared to prioritizing features selected by developers varies significantly depending on the model. Linear models (Linear Regression, Ridge, and Elastic Net) were the most sensitive to feature prioritization, demonstrating that predictive performance improves significantly with the use of all features. Linear Regression, in particular, was notably affected, showing larger absolute values of DIFF. Tree-based models (Decision Tree, Random Forest, Gradient Boosting, and XGBoost) were less impacted by feature prioritization, with smaller differences (DIFF) between ALL and DEV feature sets.

These models demonstrate greater robustness and stability, regardless of the feature set used. The variation in behavior across models suggests that when choosing an ML model for a given problem, the feature selection strategy should be aligned with the model type. For linear models, an approach that includes as many features as possible may be more beneficial. Conversely, for tree-based models, feature prioritization or reduction may be less crucial, as these models have internal mechanisms to handle variable selection.

Therefore, feature prioritization by developers has a significantly greater impact on linear models, where the absence of relevant variables can lead to a substantial decrease in performance. In contrast, tree-based models tend to be more performant and less sensitive to this prioritization, maintaining stable performance regardless of the feature set used.

Finding 14: Feature prioritization by developers has a greater impact on linear models, while tree-based models are less affected and maintain more consistent performance.

Implications for RQ$_3$. The analysis of the results reveals that using all features generally leads to better model prediction performance compared to the selection of developer-prioritized features. This is evidenced by a significant reduction in error metrics such as MSE, RMSE, MAE, and MAPE, as well as an increase in the R^2 and adjusted R^2 coefficients. These findings indicate that including all relevant variables enhances model accuracy and predictive capability, providing a more comprehensive and detailed view of the data.

Although feature prioritization may reflect developers' perceived importance of certain variables, a comprehensive data analysis can uncover additional variables that significantly contribute to model performance. Therefore, the most robust approach involves combining developers' expert knowledge with a thorough feature analysis, ensuring that essential information is not overlooked and that ML models are developed more effectively and with greater explanatory power.

5.3.4 To what extent is the proposed triviality index aligned with the developers' perception regarding the triviality of applying refactorings? (RQ$_4$)

To address RQ$_4$, we collected feedback from developers with experience in refactoring by asking questions that covered various areas: developers' profiles (such as age, gender, country, and education level), their experiences with software refactoring, and their evaluations of complexity, speed, and risk in various refactoring scenarios. We rely on the Likert scale to measure the perceived difficulty of performing operations based on these aspects, ranging from *very difficult* (1) to *very easy* (5).

The results indicate that 43.7% of developers who participated in the survey have over 7 years of experience in software development, indicating a high level of seniority. Many hold positions as backend or full-stack developers, equivalent to 56.2%. Additionally, a significant portion of the participants(93.7%) have experience in the software development process related to refactoring and have worked directly with it. This is supported by responses indicating that many developers regularly engage in refactoring as a common practice in their work. Regarding age distribution, more than 62% of the developers are between 25 and 34 years old, followed by a smaller group of younger developers aged 18 to 24. This indicates that the sample primarily consists of professionals in an intermediate stage of their careers.

Table 21 presents experts' evaluations on refactoring operations, considering complexity, speed, and risk aspects. Each developer is identified by the letter "D" followed by a number, and they are categorized into two groups based on their refactoring frequency: *D - Daily, W - Weekly*, *M - Monthly*, and *A - Annually*. Group G1 includes developers who perform refactoring daily or weekly, while Group G2 comprises those who perform refactoring monthly or annually. Group G3 represents the total number of developers. Each row in the table details the complexity, speed, and risk aspects, and the average of these three factors to calculate the final index. Refactoring operations are listed from 1 to 12, with evaluations on a Likert scale ranging from 1 (*Very Difficult*) to 5 (*Very Easy*).

In summary, our results indicate that the frequency of refactoring does not significantly impact the perception of complexity and speed among developer groups who performing refactor at different frequencies, with a minimal variation of 2%. However, the group that refactors less frequently (G2) perceives a 6% higher risk than those who refactors daily or weekly. Finally, comparisons between the groups and the predictive model showed high p-values, indicating no statistically significant differences between developers' perceptions and the model's

Table 21 – Expert assessment

Freq.	**W**	**W**	**D**	**D**	**W**	**W**	**W**	**W**	**D**		**M**	**M**	**M**	**M**	**A**	**M**	**M**		**Models**		
Ref.	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**Med**	**D10**	**D11**	**D12**	**D13**	**D14**	**D15**	**D16**	**MED**	**RF**	**GB**	**XGB**
r1	4	4	4	5	4	4	4	3	4	**4**	5	4	5	3	4	3	4	**4**			
r1	3	4	4	5	4	4	5	4	4	**4**	5	4	5	2	5	4	4	**4**			
r1	1	3	4	5	4	2	5	3	4	**4**	4	4	4	1	4	3	4	**4**	0.78	0.76	0.76
i1	3	4	4	5	4	3	5	3	4	**4**	5	4	5	2	4	3	4	**4**	**4**	**4**	**4**
r2	4	5	5	5	4	5	5	4	3	**5**	5	4	5	2	4	4	4	**4**			
r2	4	5	5	5	4	5	5	3	3	**5**	5	4	5	2	2	5	4	**4**			
r2	4	5	5	5	2	5	5	4	3	**5**	5	3	5	2	4	4	3	**4**	0.84	0.80	0.75
i2	4	5	5	5	3	5	5	4	3	**5**	5	4	5	2	3	4	4	**4**	**5**	**5**	**4**
r3	4	4	5	3	4	2	5	2	2	**4**	5	3	5	4	4	4	3	**4**			
r3	4	3	5	3	4	2	5	2	3	**3**	5	3	5	3	2	4	3	**3**			
r3	4	4	5	3	5	2	5	2	2	**4**	5	3	3	3	3	2	3	**3**	0.79	0.75	0.75
i3	4	4	5	3	4	2	5	2	2	**4**	5	3	4	3	3	3	3	**3**	**4**	**4**	**4**
r4	4	4	4	3	4	2	5	4	2	**4**	5	3	3	3	5	3	4	**3**			
r4	4	3	4	3	5	2	5	4	3	**4**	5	3	4	4	5	3	3	**4**			
r4	4	3	4	3	4	2	5	2	3	**3**	4	2	3	2	4	2	3	**3**	0.77	0.75	0.73
i4	4	3	4	3	4	2	5	3	3	**3**	5	3	3	3	5	3	3	**3**	**4**	**4**	**4**
r5	3	4	5	5	4	2	5	3	3	**4**	5	4	5	3	5	4	3	**4**			
r5	3	4	5	5	4	2	5	3	3	**4**	5	4	5	3	5	4	3	**4**			
r5	3	3	5	5	5	2	5	3	3	**3**	5	4	5	1	3	4	4	**4**	0.74	0.73	0.72
i5	3	4	5	5	4	2	5	3	3	**4**	5	4	5	2	4	4	3	**4**	**4**	**4**	**4**
r6	4	5	5	5	5	4	4	2	3	**4**	5	5	5	4	5	4	5	**5**			
r6	4	5	5	5	5	4	3	2	3	**4**	5	5	4	3	5	4	4	**4**			
r6	4	4	5	5	4	4	3	2	3	**4**	5	4	5	4	1	4	3	**4**	0.74	0.73	0.72
i6	4	5	5	5	5	4	3	2	3	**4**	5	5	5	4	4	4	4	**4**	**4**	**4**	**4**
r7	2	3	2	3	2	2	3	3	3	**3**	4	3	2	2	2	4	2	**2**			
r7	2	3	2	4	3	2	3	4	3	**3**	4	2	2	2	2	4	2	**2**			
r7	4	4	5	3	4	2	2	2	3	**3**	4	3	2	1	2	4	2	**2**	0.74	0.74	0.72
i7	3	3	3	3	3	2	3	3	3	**3**	4	3	2	2	2	4	2	**2**	**4**	**4**	**4**
r8	4	3	5	3	3	2	4	2	3	**3**	5	4	3	3	1	4	3	**3**			
r8	4	3	5	4	2	2	4	2	2	**3**	5	3	4	2	3	4	2	**3**			
r8	4	3	5	2	4	2	4	2	3	**3**	5	3	2	2	2	4	2	**2**	0.77	0.75	0.71
i8	4	3	5	3	3	2	4	2	3	**3**	5	3	3	2	2	4	2	**3**	**4**	**4**	**4**
r9	3	3	5	3	4	2	3	2	3	**3**	5	4	4	3	2	4	3	**4**			
r9	3	3	5	4	4	2	3	2	3	**3**	5	3	5	2	3	4	4	**4**			
r9	3	3	5	2	4	2	2	1	3	**3**	5	3	3	1	2	4	3	**3**	0.79	0.76	0.73
i9	3	3	5	3	4	2	3	2	3	**3**	5	3	4	2	2	4	3	**3**	**4**	**4**	**4**
r10	1	2	5	5	2	2	4	3	3	**3**	5	3	5	5	2	3	3	**3**			
r10	1	3	5	5	2	2	4	2	3	**3**	5	2	5	5	2	3	2	**3**			
r10	1	2	5	5	4	2	4	3	3	**3**	5	2	5	1	3	3	3	**3**	0.79	0.76	0.73
i10	1	2	5	5	3	2	4	3	3	**3**	5	2	5	4	2	3	3	**3**	**4**	**4**	**4**
r11	4	2	5	2	4	3	3	2	3	**3**	5	3	3	4	1	3	3	**3**			
r11	4	3	5	2	5	3	3	2	3	**3**	5	2	5	4	1	4	3	**4**	0.75	0.74	0.72
r11	4	3	5	2	2	3	2	2	3	**3**	5	2	2	1	1	3	2	**2**			
i11	4	3	5	2	4	3	3	2	3	**3**	5	2	3	3	1	3	3	**3**	**4**	**4**	**4**
r12	4	2	5	2	4	2	3	2	3	**3**	5	3	4	3	1	3	3	**3**			
r12	4	3	5	2	5	2	3	2	4	**3**	5	2	5	4	1	4	3	**4**			
r12	4	3	5	2	2	2	2	2	3	**2**	5	2	3	3	1	3	2	**3**	0.75	0.74	0.72
i12	4	3	5	2	4	2	3	2	3	**3**	5	2	4	3	1	3	3	**3**	**4**	**4**	**4**
	3.4	**3.4**	**4.7**	**3.7**	**3.8**	**2.6**	**3.9**	**2.6**	**3.0**	**3.5**	**4.9**	**3.2**	**4.0**	**2.7**	**2.8**	**3.6**	**3.1**	**3.4**			

Source: Prepared by the author

predictions.

Despite the initial expectation that frequent practice would significantly reduce the perception of complexity, our results indicate that the difference between groups G1 (daily and weekly) and G2 (monthly and annually) is minimal. The variation in perceived difficulty related to complexity and risk is approximately 2% between these groups. This limited variation can be attributed to the fact that refactoring, irrespective of its frequency, requires a consistent level of technical expertise to implement the necessary steps effectively. As a result, the perceived differences in complexity and risk between the groups are minimal.

Finding 15: The frequency of refactoring has a minimal impact, with only a 2% difference in the perception of complexity and speed among the expert groups.

Regarding the perception of risk in refactoring operations, we observed that there is a noticeable difference between the expert groups. The developers in Group G2, who perform refactoring monthly or annually, tend to assess the risk of each operation as higher compared to developers in Group G1, who refactor daily or weekly. This increase in perceived risk may be related to a lack of familiarity and confidence in performing these tasks, as infrequent practice may not provide the same level of mastery over the operations, nor the ability to anticipate and manage potential issues. In contrast, developers in Group G1, who refactor more frequently, exhibit greater confidence and less concern about risks, which may be linked to their accumulated experience in this activity. This finding highlights the importance of continuous refactoring practice as a means of reducing perceived risk and enhancing security in operations.

Finding 16: Developers in Group G2 rate the risk of refactoring 6% higher than those in Group G1.

Table 22 presents the results of applying the Mann-Whitney U test to the triviality index values derived from the experts' evaluations and the mapped results from ensemble models with similar outcomes and greater effectiveness. This non-parametric test is employed to compare differences between two independent groups and is particularly useful when the data do not follow a normal distribution.

Comparisons between Groups G1 and G2, as well as between G1, G2, and the predictive model, yield similar results. The U values indicate variations between the groups, but the p-values are consistently high, often exceeding 0.05, suggesting the absence of statistically significant differences. Neither Group G1 nor Group G2 showed significant deviations when compared to the predictive model. Group G3, which combines G1 and G2, also exhibited similar results in relation to the predictive model, with variations in U values but without achieving statistical significance.

Table 22 – Test of Mann-Whitney U with groups and models

	G1 x G2		**G1 x Models**		**G2 x Models**		**G3 x Models**	
Refactoring	**U**	**p**	**U**	**p**	**U**	**p**	**U**	**p**
REFACTORING_1	30.5	0.955	12	0.835	10.5	1	22	0.903
REFACTORING_2	23	0.367	11	0.677	5.5	0.274	16.5	0.399
REFACTORING_3	31	1	10.5	0.618	4.5	0.166	15	0.320
REFACTORING_4	31	1	7.5	0.268	6	0.322	13.5	0.228
REFACTORING_5	30	0.912	12	846	10.5	1	22.5	0.906
REFACTORING_6	26	0.569	12	845	6	0.253	18	0.503
REFACTORING_7	25	0.476	0	0.004	3	0.078	3	0.013
REFACTORING_8	26.5	0.618	6	0.170	4.5	0.189	10.5	0.130
REFACTORING_9	28.5	0.78	4.5	0.096	6	0.327	10.5	0.128
REFACTORING_10	27.5	0.702	7.5	0.295	7.5	0.553	15	0.329
REFACTORING_11	25.5	0.532	6	0.170	3	0.095	9	0.088
REFACTORING_12	30.5	0.956	6	0.173	4.5	0.190	10.5	0.132
Note.	$H_a \mu_{G1} \neq \mu_{G2}$		$H_a \mu_{G1} \neq \mu_{Model}$		$H_a \mu_{G2} \neq \mu_{Model}$		$H_a \mu_{G3} \neq \mu_{Model}$	

Source: Prepared by the author

Finding 17: Despite variations in the U values, comparisons between Groups G1, G2, and G3 indicate that the observed differences between developer groups and models are not statistically significant, reinforcing the similarity between the distributions.

Implications RQ$_4$: These findings suggest that continuous refactoring practice may not be critically important for perceptions of complexity and speed, but it does have a subtle impact on the perception of risk. Additionally, there is alignment between the perceptions of the expert developer groups and the predictive models presented in this study. This indicates that the triviality index is effectively aligned with developers' perceptions, positioning it as a reliable proposal for implementation in automated refactoring solutions for developers.

5.4 Threats to validity

This section discusses threats to the validity of the study according to the classification of Wohlin *et al.* (2012).

Internal Validity. In our study, we utilized various tools for commit analysis: RefactoringMiner (Tsantalis *et al.*, 2020), known for its high accuracy in detecting software refactorings; Pydriller (Tsantalis *et al.*, 2018), for extracting source code from files; PMD[9], to identify code smells and security issues; and CKTool (Aniche, 2015), for collecting code metrics from files involved in refactorings. Although these tools offer high precision, failures can still occur during the mining process. To minimize this risk, we repeated some steps of the process whenever necessary.

Similarly, some metrics may have been used as independent variables in the ML models but may not be sufficiently related to the triviality index, which could inflate the results and accuracy of the models in some cases. To address this issue, we conducted a survey with developers to identify the most relevant metrics and ensure that the model uses only metrics pertinent to our objective. Additionally, open-source projects may vary in their refactoring practices, potentially introducing biases into the results. To mitigate this threat, we employed the SMOGN data balancing technique, which helps reduce bias in the models. Furthermore, cross-validation and hyperparameter tuning were applied to enhance the robustness of the models.

External Validity. Although we analyzed a large number of projects (1,259), only 77,630 refactorings met the criteria established by our filters, and the tools used were limited to JAVA projects. This may restrict the generalizability of the model, especially when considering factors such as programming language, maintainability, programming paradigm, or software quality. Additionally, depending on the domain of the systems, the results may vary significantly, potentially affecting the model's applicability to systems outside the context of the analyzed data. To mitigate this limitation, we selected projects from different domains and involved various developers to enhance the robustness and applicability of the model

Construct Validity. Defining the *Triviality Index* as a composite metric may present challenges in practical interpretation, especially if the weights assigned to different aspects (complexity, speed, and risk) do not accurately reflect the reality of the analyzed projects. To mitigate this threat, the validation process with developers aims to ensure that the metrics used are relevant and suitable for refactoring practice. Additionally, the flexibility of the index formula allows for adjustments to the weights assigned to each aspect of the triviality index, helping to tailor the model to different development contexts and enhancing its applicability.

Another threat to construct validity was the imbalance in the dataset. The features

[9] https://pmd.github.io/

extracted for the dataset were disproportionate, which could introduce bias into the results, with aspects having a higher number of features potentially exerting a stronger influence on predictions. To mitigate this threat, we applied the SMOGN oversampling technique, specifically developed for regression problems (Branco *et al.*, 2017). However, even with this approach, algorithms may vary in their sensitivity to this technique, and the challenge of ensuring proportional representation of all features may persist.

Conclusion Validity. Since the models were applied to a specific set of projects and refactorings, there is a risk that the conclusions may not be robust or that the results could be influenced by specific variables or potential overfitting. To address this issue, we utilized multiple projects and a range of ML models, conducting comparative analyses to ensure that the findings were not limited to a single approach. Additionally, to enhance the reliability of our conclusions and reduce this risk, we implemented cross-validation and hyperparameter tuning.

5.5 Concluding remarks

In this study, we analyzed and proposed an index based on code metrics to evaluate the triviality of refactoring. We investigated whether the prioritization of features by developers influenced the effectiveness of prediction models for the triviality index in code refactorings and validated how well these models aligned with expert developers in refactoring. Our results indicated that, although the features prioritized by developers are valuable, including all available features often leads to better performance of the predictive models. Additionally, we observed a strong alignment between the perceptions of expert developer groups and the predictive models presented in this study. The practical implications of our study are significant for software engineering. Using predictive models that are both highly effective and aligned with developers' perceptions makes refactoring decisions more reliable and accurate.

Our main findings were: (i) The most important aspects, based on our analysis of developers' opinions and the distribution of metrics, were risk and complexity, followed by speed. This suggests that simpler and faster refactorings with lower risk are more likely to be implemented; (ii) Our machine learning models configured with all available features performed significantly better than those using only the features prioritized by developers; (iii) Unlike linear models, ensemble models such as Random Forest and Gradient Boosting demonstrated higher performance and efficiency, regardless of whether they used all features or just those prioritized by developers. These models were less impacted by feature selection, maintaining stable and effective performance even with a reduced set of variables; and (iv) The proposed triviality index metric proved to be effective and aligns well with developers' perceptions, making it a reliable solution for implementation in automated refactoring tools.

For future research, we recommend exploring the assignment of differentiated weights to each aspect of the triviality index, such as complexity, speed, and risk, in order to refine prediction accuracy. Additionally, it would be valuable to investigate in detail which metrics have the greatest impact on predicting the triviality index, identifying those that truly influence the results. We also suggest expanding the study to include a wider variety of metrics, aiming for a more robust balance in predictive modeling. Finally, studies analyzing the impact of these approaches across different types of software projects and programming languages could provide valuable insights into the applicability and generalization of the results in diverse contexts.

6 CONCLUSION AND FUTURE WORK

In this Master's dissertation, we identify the relationship between trivial and non-trivial refactorings and propose a metric that evaluates the triviality of refactoring implementation. The triviality index is an important metric for developers, as it can influence the decision of whether or not to implement one refactoring. In particular, we conduct empirical studies to explore the triviality of refactoring implementation. Indeed, there is a growing need for reliable automated tools that can help developers evaluate refactoring implementation.

Even considering that refactoring brings benefits to software, such as: (i) increased overall software quality; (ii) decreased maintenance costs; and, (iii) increased developer productivity (Moser *et al.*, 2007; Fowler, 2018). Refactorings may sometimes negatively affect software maintainability (Bavota *et al.*, 2012; Kim *et al.*, 2014; Penta *et al.*, 2020; Almogahed *et al.*, 2023; Nikolaidis *et al.*, 2024). Furthermore, even developers using automated tools may hesitate to perform necessary refactorings, overestimating the risk associated with them (Silva *et al.*, 2016; Abid *et al.*, 2022; Tan *et al.*, 2024). Additionally, the literature review identified the importance of developing empirical studies that promote solutions to improve and automate the refactoring technique (Sharma *et al.*, 2015; Singh; Kaur, 2018; Kaur; Singh, 2019; Baqais; Alshayeb, 2020; Liu *et al.*, 2024).

To address these limitations, this Master's dissertation presented two empirical studies. The first investigated how trivial refactorings can affect the prediction of non-trivial refactorings, considering code attributes and metrics. The experiment was performed on 1,291 open-source projects and 55 metrics of code, using the following algorithms: Decision Tree, Random Forest, Logistic Regression, Naive Bayes, and Neural Network as a supervised learning technique with a classification problem. In addition, two data balancing techniques were applied, and the refactorings were grouped according to their triviality, proposing contexts based on combinations of refactoring types. The datasets were also separated to identify possible generalizations of the models.

The second study proposed a metric based on ML through the regression problem to evaluate the triviality of implementing a refactoring based on the aspects of complexity, speed and risk. The experiment was performed on 1,259 open-source projects and 58 metrics of code, using the following algorithms: Linear Regression, Elastic Net, Ridge, Decision Tree, Random Forest, Gradient Boosting, and XGBoost as a supervised learning technique with a regression problem. The study investigated how the prioritization of the features considered most important by developers affects the effectiveness of ML models in predicting the triviality index. In addition, we verified the alignment of perceptions between groups of developers with experience in refactoring and predictive models.

This chapter presents a summary of the main contributions of this master's thesis, as well as the publications derived during the period of its development. Additionally, suggestions for future work based on the contributions of this research are offered, concluding with the final considerations.

6.1 Main contributions

The main objective of this Master's dissertation was to identify the relationship between trivial and non-trivial refactorings, in addition to proposing a metric that evaluates the triviality of refactoring implementation. To this end, a study was initially conducted to investigate the impact of trivial refactorings on the prediction of non-trivial refactorings. This study is presented in Chapter 4. Next, a study was conducted that proposes an index that evaluates refactoring triviality to identify the degree of difficulty of its implementation concerning the aspects of complexity, speed and risk from the point of view of software developers. This study is presented in Chapter 5. The main contributions of this Master's dissertation are presented below.

Contribution 1: *Tree-based models and Neural Networks better detect refactoring opportunities.* One of the main contributions of this Master's dissertation is the detailed analysis of the performance of different machine learning algorithms in detecting refactoring opportunities, with a particular focus on tree-based models and neural networks. The study revealed that algorithms such as Random Forest, Decision Tree, and Neural Network performed better when trained with code metrics to identify refactoring opportunities. However, we observed that only Random Forest and Decision Tree models could achieve good generalization across different refactoring data contexts. This finding is relevant because it highlights the importance of selecting algorithms that perform well on a specific dataset and generalize their performance to other data domains. Finally, this contribution provides a solid basis for choosing machine learning algorithms in future research and practical applications in detecting refactoring opportunities, considering the context of this study.

Contribution 2: *Impact of separating trivial and non-trivial refactorings on model efficiency.* Separating trivial and non-trivial refactorings into different classes results in more efficient models, even when applied to different datasets. This classification approach allows ML models to acquire greater efficiency, thus improving prediction accuracy. Furthermore, the study demonstrated that using balancing techniques that increase or decrease samples may not be the best strategy to improve models trained by datasets composed of code metrics. This contribution is particularly relevant to software development practice, as it suggests that proper classification and separation of refactorings can significantly improve the efficiency of predictive models.

Contribution 3: *Proposal of a Triviality Index based on code metrics with developers' perception.* One of the main contributions of this Master's dissertation is the proposal of a metric called the Triviality Index. We designed this index to evaluate the triviality of implementing refactorings, which can assess the degree of difficulty of implementation based on complexity, speed, and risk. To achieve the objective of this proposal, ML models based on the Regression problem were used with features composed of static code metrics. Additionally, we consulted internal and external developers for the projects to prioritize the most relevant features for refactoring activity. However, the research demonstrated that, when including all available features, predictive models achieve superior performance compared to the prioritization of features by developers. This finding is significant because it suggests that considering a wide range of code metrics, a comprehensive approach can provide a more accurate and robust assessment of the triviality of refactorings. In addition, we conducted a survey to verify the index's agreement with the perceptions of expert developers regarding refactoring. The survey showed significant alignment between expert perceptions and the results of the predictive models. This indicates that the index's effectiveness is consistent with that of developers, making it useful for decision-making about refactoring in software development.

Contribution 4: *Implementations of additional metrics.* An important contribution of this Master's dissertation is the implementation of additional metrics, such as Score of Similarity (SS) and Frequency of Commit (FOC), which represent important information for analyzing refactoring and the effectiveness of predictive models. The Similarity Score was used to measure the similarity between code versions. We provide a Java program to calculate the metric in the code using Abstract Static Tree (AST) as the basis for the analysis. The similarity between code files is an important metric to evaluate the degree of changes made over time and serves for comparative analysis in terms of refactoring. On the other hand, the FOC metric was implemented with Python scripts. It was used to evaluate the frequency with which certain parts of the code are modified. It was calculated in different time intervals, representing the frequency with which refactoring commits were made within these periods. The inclusion of these additional metrics allowed us to obtain significant information for the analysis of refactoring triviality, contributing to the creation of more accurate and robust predictive models. This contribution enriches refactoring analysis by providing complementary information on code changes and practical implementation to be reused and improved in future research.

Contribution 5: *Developers' perceptions regarding the refactoring triviality.* Another significant contribution of this Master's dissertation was the developers' perceptions resulting from the surveys on the relevance of code metrics and the degree of difficulty of implementing refactorings. The first survey, which involved developers internal and external to the projects, collected qualitative and quantitative data on the most relevant metrics for refactoring. Our results indicate that developers external to the projects used in the study tend to value the code metrics that involve complexity, best practices, performance, and security. On the other hand, internal developers seem to be more conservative in their assessments. These findings provide an empirical basis for the selection of metrics in predictive models, aligning them with the needs of developers. The second survey evaluated the difficulty of implementing refactorings based on the aspects of complexity, speed and risk. For data analysis, developers were divided into two groups according to the frequency of applying refactoring. The results suggest that the ongoing practice of refactoring may not be as critical to the perception of the degree of refactoring implementation in terms of complexity and speed. Still, it does have a modest impact on the perception of risk. This assessment is essential to understand developers' barriers and develop mitigation strategies. Thus, developers' responses provide valuable insights into their experiences and perceptions regarding code refactoring.

6.2 Publications

Until the completion of this Master's dissertation, two articles on the topic were published, references to Chapter 4. The most recent article, detailed in Chapter 5, is in the submission phase. Table 23 contains information about these publications.

6.3 Future work

During the development of this Master's dissertation, several insights, challenges and opportunities emerged. Below, some directions for future work are proposed.

Table 23 – Research publication

Publication	Description
Pinheiro, D., Bezerra, C., Uchôa, A. (2022). How do Trivial Refactorings Affect Classification Prediction Models?. In Proceedings of the 16th Brazilian Symposium on Software Components, Architectures, and Reuse (pp. 81-90). https://doi.org/10.1145/3559712.3559720	Direct result of this Master's dissertation. This paper investigates how less complex (trivial) refactorings affect the prediction of more complex (non-trivial) refactorings. To do this, we classify refactorings based on their triviality, extract metrics from the code, contextualize the data and train machine learning algorithms to investigate the effect caused. Best paper awards at SBCARS'22
Pinheiro, D., Bezerra, C., Uchôa, A. (2024). On the Effectiveness of Trivial Refactorings in Predicting Non-trivial Refactorings. Journal of Software Engineering Research and Development, 12(1), 5:1 – 5:16. https://doi.org/10.5753/jserd.2024.3324	Direct result of this Master's dissertation and is an extension of the previous publication. This study aims to identify refactoring activity in non-trivial operations through trivial operations accurately. For this, we use classifier models of supervised learning, considering the influence of trivial refactorings and evaluating performance in other data domains. It is presented in Chapter 4

Source: Prepared by the author

Future Work 1: *Development of an automated tool to calculate the triviality index*: This Master's dissertation proposes an index that evaluates the triviality of implementing refactorings. A possible extension is to develop an automated tool that calculates this metric during software development. The tool can be integrated into IDEs, version control systems, or a standalone module to help developers evaluate the triviality of refactorings in real development environments.

Future Work 2: *Conducting studies in different contexts.* We address the use in several open source software projects extracted from GitHub with few domains. The behavior in other software project domains may generate other results. Thus, we recommend expanding the study to other domains, companies, and industrial software to validate the effectiveness of the proposed triviality index.

Future Work 3: *Exploring differentiated weights in the triviality index.* The metric was proposed to adopt different weights in aspects to calculate the triviality index. However, this study used the weight 1 for all aspects, which may not reflect in other contexts. We recommend investigating the assignment of differentiated weights to each aspect, such as complexity, speed, and risk, to refine the prediction accuracy.

Future Work 4: *Analysis of the metrics used to configure the prediction models.* Our study prioritized the importance of the metrics through the developers' perception. However, performing a detailed analysis of the metrics using a statistical approach and investigating their correlations can generate new results and improve the effectiveness of the predictive models. In addition, it can generate insights to identify the metrics that most influence the results.

Future Work 5: *Extending the set of metrics.* Another approach to investigate the effectiveness of the proposed index is to adopt a greater variety of metrics for each aspect, seeking a more robust balance in predictive modeling. In addition, the inclusion of specific and detailed metric types can provide new results and insights.

Future Work 6: *Extending the study proposal to other ecosystems.* The studies presented in this Master's dissertation used only systems on Java. New studies can be carried out to investigate how the triviality prediction approach behaves in different software projects in other programming languages, evaluating the consistency of the results.

Future Work 7: *Investigation of the application of new ML algorithms.* This Master's dissertation used common algorithms to train predictive models. Thus, future work that investigates the application of new ML algorithms, such as deep neural networks and reinforcement learning algorithms, can improve the prediction of the triviality of refactorings, as well as suggest a new implementation approach. In addition, this study can compare the performance of these new algorithms with the traditional methods used in this Master's dissertation, evaluating their effectiveness in different contexts and types of software projects.

Future Work 8: *Investigation of how trivial refactorings influence the prediction of non-trivial refactorings.* The first study of this Master's dissertation investigated how trivial refactorings influence the prediction of non-trivial refactorings. However, the definition of triviality in the first study is based on the set of operations and changes that occur in each refactoring operation. A new research can be conducted using the triviality metric proposed in the second study to identify whether refactoring is trivial or not. In this way, it would be possible to compare the results obtained based on two different definitions of triviality and evaluate their influences.

Future Work 9: *Investigation of triviality aspects in other contexts*: The concept of refactoring triviality was only explored in depth in this Master's dissertation. However, it is based on the aspects of complexity, speed, and risk. This way, different software companies can conduct a case study to observe how aspects related to refactoring triviality are identified, prioritized, and implemented in the real development environment. This study can provide valuable insights into the applicability and effectiveness of the proposed index in different development contexts.

Future Work 10: *Application of the Triviality Index in production environments* The maintenance of systems in production requires greater caution to mitigate risks. A relevant suggestion for future work is the application of the triviality index in systems in the production environment. In this context, the evaluation of the triviality of refactorings, with a focus on risk prioritization, can provide valuable data for decision-making to avoid introducing new failures and bugs into the system.

Future Work 11: *Integrating Large Language Models for Enhanced Refactoring Prediction* This Master's dissertation used several machine learning algorithms to train the models, but did not use Large Language Models (LLMs). As future work, the implementation of LLMs can improve the accuracy and interpretability of refactoring triviality indices. In addition, LLMs can extract insights into code quality and enrich features beyond traditional metrics. This approach would develop more robust predictive models aligned with developer intentions, making predictive models more efficient.

BIBLIOGRAPHY

ABID, C.; GAALOUL, K.; KESSENTINI, M.; ALIZADEH, V. What refactoring topics do developers discuss? A large scale empirical study using stack overflow. **IEEE Access**, v. 10, p. 56362–56374, 2022.

AGGARWAL, K.; SINGH, Y.; KAUR, A.; MALHOTRA, R. Empirical study of object-oriented metrics. **J. Object Technol.**, v. 5, n. 8, p. 149–173, 2006.

AGNIHOTRI, M.; CHUG, A. A systematic literature survey of software metrics, code smells and refactoring techniques. **Journal of Information Processing Systems**, v. 16, n. 4, p. 915–934, 2020.

AKHTAR, S. M.; NAZIR, M.; ALI, A.; KHAN, A. S.; ATIF, M.; NASEER, M. A systematic literature review on software-refactoring techniques, challenges, and practices. **VFAST Transactions on Software Engineering**, v. 10, n. 4, p. 93–103, 2022.

ALKHALID, A.; ALSHAYEB, M.; MAHMOUD, S. Software refactoring at the function level using new adaptive k-nearest neighbor algorithm. **Advances in Engineering Software**, v. 41, n. 10-11, p. 1160–1178, 2010.

ALKHALID, A.; ALSHAYEB, M.; MAHMOUD, S. A. Software refactoring at the package level using clustering techniques. **IET software**, IET, v. 5, n. 3, p. 274–286, 2011.

ALMOGAHED, A.; MAHDIN, H.; OMAR, M.; ZAKARIA, N. H.; MOSTAFA, S. A.; ALQAHTANI, S. A.; PATHAK, P.; SHAHARUDIN, S. M.; HIDAYAT, R. A refactoring classification framework for efficient software maintenance. **IEEE Access**, v. 11, p. 78904–78917, 2023.

ALOMAR, E.; LIU, J.; ADDO, K.; MKAOUER, M. W.; NEWMAN, C.; OUNI, A.; YU, Z. On the documentation of refactoring types. **Automated Software Engineering**, v. 29, 05 2022.

ALOMAR, E. A.; PERUMA, A.; MKAOUER, M. W.; NEWMAN, C.; OUNI, A.; KESSENTINI, M. How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. **Expert Systems with Applications**, Elsevier, v. 167, p. 114176, 2021.

ANICHE, M. **Java code metrics calculator (CK)**. [*S. l.*], 2015. Available in https://github.com/mauricioaniche/ck/.

ANICHE, M.; MAZIERO, E.; DURELLI, R.; DURELLI, V. H. The effectiveness of supervised machine learning algorithms in predicting software refactoring. **IEEE Transactions on Software Engineering**, IEEE, v. 48, n. 4, p. 1432–1450, 2020.

AVGERIOU, P.; KRUCHTEN, P.; OZKAYA, I.; SEAMAN, C. Managing technical debt in software engineering (dagstuhl seminar 16162). Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2016.

AZEEM, M. I.; PALOMBA, F.; SHI, L.; WANG, Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. **Information and Software Technology**, v. 108, p. 115–138, 2019.

BAQAIS, A.; ALSHAYEB, M. Automatic software refactoring: a systematic literature review. **Software Quality Journal**, Springer, v. 28, n. 2, p. 459–502, 2020.

BAVOTA, G.; CARLUCCIO, B. D.; LUCIA, A. D.; PENTA, M. D.; OLIVETO, R.; STROLLO, O. When does a refactoring induce bugs? An empirical study. In: **2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation**. [*S. l.*: *s. n.*], 2012. p. 104–113.

BAVOTA, G.; LUCIA, A. D.; PENTA, M. D.; OLIVETO, R.; PALOMBA, F. An experimental investigation on the innate relationship between quality and refactoring. **Journal of Systems and Software**, Elsevier, v. 107, p. 1–14, 2015.

BAVOTA, G.; OLIVETO, R.; LUCIA, A. D.; ANTONIOL, G.; GUÉHÉNEUC, Y.-G. Playing with refactoring: Identifying extract class opportunities through game theory. In: IEEE. **2010 IEEE International Conference on Software Maintenance**. [*S. l.*], 2010. p. 1–5.

BERRY, M. W.; MOHAMED, A.; YAP, B. W. **Supervised and unsupervised learning for data science**. Berlin, Germany: Springer, 2019.

BERTRAND, G. Simple points, topological numbers and geodesic neighborhoods in cubic grids. **Pattern recognition letters**, v. 15, n. 10, p. 1003–1011, 1994.

BIBIANO, A. C.; COUTINHO, D.; UCHÔA, A.; ASSUNÇAO, W. K.; GARCIA, A.; MELLO, R. de; COLANZI, T. E.; TENÓRIO, D.; VASCONCELOS, A.; FONSECA, B. *et al.* Enhancing recommendations of composite refactorings based on the practice. In: IEEE. **24th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)**. [*S. l.*], 2024. p. 1–12.

BIBIANO, A. C.; UCHÔA, A.; ASSUNÇÃO, W. K.; TENÓRIO, D.; COLANZI, T. E.; VERGILIO, S. R.; GARCIA, A. Composite refactoring: Representations, characteristics and effects on software projects. **Information and Software Technology**, Elsevier, v. 156, p. 107134, 2023.

BISHOP, C. M.; NASRABADI, N. M. **Pattern recognition and machine learning**. New York: Springer, 2006. v. 4.

BOIS, B. D.; DEMEYER, S.; VERELST, J. Refactoring-improving coupling and cohesion of existing code. In: IEEE. **11th working conference on reverse engineering**. [*S. l.*], 2004. p. 144–151.

BRANCO, P.; TORGO, L.; RIBEIRO, R. P. SMOGN: a pre-processing approach for imbalanced regression. In: **First international workshop on learning with imbalanced domains: Theory and applications**. [*S. l.*]: PMLR, 2017. p. 36–50.

BREIMAN, L. Random forests. **Machine learning**, v. 45, p. 5–32, 2001.

BREIMAN, L. **Classification and regression trees**. [*S. l.*]: Routledge, 2017.

BRYKSIN, T.; NOVOZHILOV, E.; SHPILMAN, A. Automatic recommendation of move method refactorings using clustering ensembles. In: **Proceedings of the 2nd International Workshop on Refactoring**. New York: Association for Computing Machinery, 2018. p. 42–45.

CARVALHO, D. V.; PEREIRA, E. M.; CARDOSO, J. S. Machine learning interpretability: A survey on methods and metrics. **Electronics**, MDPI, v. 8, n. 8, p. 832, 2019.

CASSELL, K.; ANDREAE, P.; GROVES, L. A dual clustering approach to the extract class refactoring. In: **SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering**. [*S. l.*: *s. n.*], 2011. p. 77–82.

CHAI, T.; DRAXLER, R. R. Root mean square error (RMSE) or mean absolute error (MAE). **Geoscientific model development discussions**, v. 7, n. 1, p. 1525–1534, 2014.

CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. Smote: synthetic minority over-sampling technique. **Journal of artificial intelligence research**, v. 16, p. 321–357, 2002.

CHEN, T.; GUESTRIN, C. Xgboost: A scalable tree boosting system. In: **Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining**. New York: Association for Computing Machinery, 2016. p. 785–794.

CHICCO, D.; JURMAN, G. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. **BMC genomics**, Springer, v. 21, n. 1, p. 1–13, 2020.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on software engineering**, IEEE, v. 20, n. 6, p. 476–493, 1994.

CORD, M.; CUNNINGHAM, P. **Machine learning techniques for multimedia: case studies on organization and retrieval**. Berlin, Germany: Springer, 2008.

CUTLER, A.; CUTLER, D. R.; STEVENS, J. R. Random forests. In: ZHANG, C.; MA, Y. (Ed.). **Ensemble Machine Learning: Methods and Applications**. New York, NY: Springer New York, 2012. p. 157–175.

DALLAL, J. A. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. **Information and Software Technology**, Elsevier, v. 54, n. 10, p. 1125–1141, 2012.

DAVIS, J.; GOADRICH, M. The relationship between precision-recall and roc curves. In: **Proceedings of the 23rd International Conference on Machine Learning**. New York, NY, US: Association for Computing Machinery, 2006. p. 233–240.

DEHAGHANI, S. M. H.; HAJRAHIMI, N. Which factors affect software projects maintenance cost more? **Acta Informatica Medica**, The Academy of Medical Sciences of Bosnia and Herzegovina, v. 21, n. 1, p. 63, 2013.

EPOSHI, A.; OIZUMI, W.; GARCIA, A.; SOUSA, L.; OLIVEIRA, R.; OLIVEIRA, A. Removal of design problems through refactorings: Are we looking at the right symptoms? In: **2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)**. Montreal, QC, Canada: IEEE, 2019. p. 148–153.

FERNANDES, E.; CHáVEZ, A.; GARCIA, A.; FERREIRA, I.; CEDRIM, D.; SOUSA, L.; OIZUMI, W. Refactoring effect on internal quality attributes: What haven't they told you yet? **Information and Software Technology**, Elsevier, v. 126, p. 106347, 2020.

FERREIRA, T.; IVERS, J.; YACKLEY, J. J.; KESSENTINI, M.; OZKAYA, I.; GAALOUL, K. Dependent or Not: Detecting and Understanding Collections of Refactorings. **IEEE Transactions on Software Engineering**, v. 49, n. 6, p. 3344–3358, 2023.

FOWLER, M. **Refactoring: improving the design of existing code**. 2nd. ed. Boston, MA, US: Addison-Wesley Professional, 2018.

FRIEDMAN, J. H. Greedy function approximation: A gradient boosting machine. **Annals of statistics**, JSTOR, v. 29, p. 1189–1232, 2001.

HANLEY, J. A.; MCNEIL, B. J. The meaning and use of the area under a receiver operating characteristic (roc) curve. **Radiology**, v. 143, n. 1, p. 29–36, 1982.

HASANIN, T.; KHOSHGOFTAAR, T. The effects of random undersampling with simulated class imbalance for big data. In: IEEE. **2018 IEEE international conference on information reuse and integration (IRI)**. Salt Lake City, UT, US, 2018. p. 70–79.

HUTTON, D. M. Clean code: a handbook of agile software craftsmanship. **Kybernetes**, v. 38, n. 6, p. 1035–1035, 2009.

INTERNATIONAL ELECTROTECHNICAL COMMISSION. **ISO/IEC 25010:2011 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models**. Geneva, Switzerland, 2011.

JACCARD, P. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. **Bull Soc Vaudoise Sci Nat**, v. 37, p. 547–579, 1901.

JAMES, G.; WITTEN, D.; HASTIE, T.; TIBSHIRANI, R.; TAYLOR, J. **An introduction to statistical learning: With applications in python**. 3rd. ed. New York, NY, US: Springer Nature, 2023.

JIN, W.; LI, Z. J.; WEI, L. S.; ZHEN, H. The improvements of bp neural network learning algorithm. In: IEEE. **WCC 2000-ICSP 2000. 2000 5th international conference on signal processing proceedings. 16th world computer congress 2000**. Beijing, China, 2000. v. 3, p. 1647–1649.

JORDAN, M. I.; MITCHELL, T. M. Machine learning: Trends, perspectives, and prospects. **Science**, American Association for the Advancement of Science, v. 349, n. 6245, p. 255–260, 2015.

KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement learning: A survey. **Journal of artificial intelligence research**, v. 4, p. 237–285, 1996.

KAUR, S.; SINGH, P. How does object-oriented code refactoring influence software quality? research landscape and challenges. **Journal of Systems and Software**, v. 157, p. 110394, 2019.

KHANAM, Z. Analyzing refactoring trends and practices in the software industry. **International Journal of Advanced Research in Computer Science**, v. 10, n. 5, 2018.

KHLEEL, N. A. A.; NEHéZ, K. Detection of code smells using machine learning techniques combined with data-balancing methods. **International Journal of Advances in Intelligent Informatics**, v. 9, n. 3, p. 402–417, 2023.

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. An empirical study of refactoring challenges and benefits at microsoft. **IEEE Transactions on Software Engineering**, IEEE, v. 40, n. 7, p. 633–649, 2014.

KIM, S.; KIM, H. A new metric of absolute percentage error for intermittent demand forecasts. **International Journal of Forecasting**, v. 32, n. 3, p. 669–679, 2016.

KLUYVER, T.; RAGAN-KELLEY, B.; PEREZ, F.; GRANGER, B.; BUSSONNIER, M.; FREDERIC, J.; KELLEY, K.; HAMRICK, J.; GROUT, J.; CORLAY, S.; IVANOV, P.; AVILA, D.; ABDALLA, S.; WILLING, C.; TEAM, J. D. Jupyter Notebooks - A publishing format for reproducible computational workflows. In: LOIZIDES, F.; SCMIDT, B. (Ed.). **Positioning and Power in Academic Publishing: Players, Agents and Agendas**. [*S. l.*]: IOS Press, 2016. p. 87–90.

KUHN, M.; JOHNSON, K. **Applied Predictive Modeling**. New York, NY, US: Springer, 2013.

KUMAR, L.; LAL, S.; GOYAL, A.; MURTHY, N. L. B. Change-proneness of object-oriented software using combination of feature selection techniques and ensemble learning techniques. In: **Proceedings of the 12th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference)**. New York, NY, US: Association for Computing Machinery, 2019.

LACERDA, G.; PETRILLO, F.; PIMENTA, M.; GUéHéNEUC, Y. G. Code smells and refactoring: A tertiary systematic review of challenges and observations. **Journal of Systems and Software**, v. 167, p. 110610, 2020.

LEHMAN, M. M. Programs, life cycles, and laws of software evolution. **Proceedings of the IEEE**, IEEE, v. 68, n. 9, p. 1060–1076, 1980.

LI, W.; HENRY, S. Object-oriented metrics that predict maintainability. **Journal of systems and software**, Elsevier, v. 23, n. 2, p. 111–122, 1993.

LIKERT, R. A technique for the measurement of attitudes. **Archives of Psychology**, v. 22 140, p. 55–55, 1932.

LIU, H.; GAO, Y.; NIU, Z. An initial study on refactoring tactics. In: IEEE. **2012 IEEE 36th Annual Computer Software and Applications Conference**. Izmir, Turkey, 2012. p. 213–218.

LIU, J.; JIN, W.; ZHOU, J.; FENG, Q.; FAN, M.; WANG, H.; LIU, T. 3erefactor: Effective, efficient and executable refactoring recommendation for software architectural consistency. **IEEE Transactions on Software Engineering**, p. 1–23, 2024.

LORENZ, M.; KIDD, J. **Object-oriented software metrics: a practical guide**. River, NJ, US: Prentice-Hall, Inc., 1994.

MAHESH, B. Machine learning algorithms-a review. **International Journal of Science and Research (IJSR)**, v. 9, p. 381–386, 2020.

MALHOTRA, R.; CHUG, A. An empirical study to assess the effects of refactoring on software maintainability. In: IEEE. **2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)**. Jaipur, India, 2016. p. 110–117.

MALHOTRA, R.; JAIN, J. Analysis of refactoring effect on software quality of object-oriented systems. In: **International Conference on Innovative Computing and Communications**. Singapore: Springer, 2019. p. 197–212.

MALHOTRA^1, R.; CHUG, A. Software maintainability prediction using machine learning algorithms. **Software engineering: an international Journal (SeiJ)**, v. 2, n. 2, 2012.

MCCABE, T. J. A complexity measure. **IEEE Transactions on software Engineering**, IEEE, n. 4, p. 308–320, 1976.

McDonald, G. C. Ridge regression. **WIREs Computational Statistics**, v. 1, n. 1, p. 93–100, 2009.

MELLO, R. de; OLIVEIRA, R.; UCHÔA, A.; OIZUMI, W.; GARCIA, A.; FONSECA, B.; MELLO, F. de. Recommendations for developers identifying code smells. **IEEE Software**, IEEE, v. 40, n. 2, p. 90–98, 2022.

MENDEZ, D.; GRAZIOTIN, D.; WAGNER, S.; SEIBOLD, H. Open science in software engineering. **Contemporary empirical methods in software engineering**, Springer, p. 477–501, 2020.

MENS, T.; TOURWÉ, T. A survey of software refactoring. **IEEE Transactions on software engineering**, IEEE, v. 30, n. 2, p. 126–139, 2004.

MITCHELL, R.; MICHALSKI, J.; CARBONELL, T. An artificial intelligence approach. **Machine learning. Berlin, Germany**, Springer, 2013.

MITCHELL, T. M.; MITCHELL, T. M. **Machine learning**. New York, NY, US: McGraw-hill, 1997. v. 1.

MOHAMMED, R.; RAWASHDEH, J.; ABDULLAH, M. Machine learning with oversampling and undersampling techniques: overview study and experimental results. In: **2020 11th international conference on information and communication systems (ICICS)**. Irbid, Jordan: IEEE, 2020. p. 243–248.

MONTGOMERY, D. C.; PECK, E. A.; VINING, G. G. **Introduction to linear regression analysis**. Hoboken, NJ, US: John Wiley & Sons, 2021.

MOREO, A.; ESULI, A.; SEBASTIANI, F. Distributional random oversampling for imbalanced text classification. In: **Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval**. New York, NY, US: Association for Computing Machinery, 2016. p. 805–808.

MOSER, R.; ABRAHAMSSON, P.; PEDRYCZ, W.; SILLITTI, A.; SUCCI, G. A case study on the impact of refactoring on quality and productivity in an agile team. In: **IFIP Central and East European Conference on Software Engineering Techniques**. Berlin, Germany: Springer, 2007. p. 252–266.

MURPHY-HILL, E.; BLACK, A. P. Refactoring tools: Fitness for purpose. **IEEE software**, IEEE, v. 25, n. 5, p. 38–44, 2008.

MURPHY-HILL, E.; PARNIN, C.; BLACK, A. P. How we refactor, and how we know it. **IEEE Transactions on Software Engineering**, v. 38, n. 1, p. 5–18, 2012.

MUSCHELLIIII, J. Roc and auc with a binary predictor: a potentially misleading metric. **Journal of classification**, Springer, v. 37, n. 3, p. 696–708, 2020.

NAIK, P.; NELABALLI, S.; PUSULURI, V. S.; KIM, D.-K. Deep learning-based code refactoring: A review of current knowledge. **Journal of Computer Information Systems**, Taylor & Francis, v. 64, n. 2, p. 314–328, 2023.

NIKOLAIDIS, N.; MITTAS, N.; AMPATZOGLOU, A.; FEITOSA, D.; CHATZIGEORGIOU, A. A metrics-based approach for selecting among various refactoring candidates. **Empirical Software Engineering**, Springer, v. 29, n. 1, p. 25, 2024.

NUCCI, D. D.; PALOMBA, F.; TAMBURRI, D. A.; SEREBRENIK, A.; LUCIA, A. D. Detecting code smells using machine learning techniques: Are we there yet? In: **2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. Campobasso, Italy: IEEE, 2018. p. 612–621.

NYAMAWE, A. S. Mining commit messages to enhance software refactorings recommendation: A machine learning approach. **Machine Learning with Applications**, v. 9, p. 100316, 2022.

OPDYKE, W. F. **Refactoring Object-Oriented Frameworks**. Thesis (Ph.D.) – University of Illinois at Urbana-Champaign, Urbana, IL, US, 1992.

OUNI, A.; KESSENTINI, M.; BECHIKH, S.; SAHRAOUI, H. Prioritizing code-smells correction tasks using chemical reaction optimization. **Software Quality Journal**, Springer, v. 23, n. 2, p. 323–361, 2015.

PADHY, N.; PANIGRAHI, R.; BABOO, S. A systematic literature review of an object oriented metric: Reusability. In: **2015 International Conference on Computational Intelligence and Networks**. Odisha, India: IEEE, 2015. p. 190–191.

PAIXÃO, M.; UCHÔA, A.; BIBIANO, A. C.; OLIVEIRA, D.; GARCIA, A.; KRINKE, J.; ARVONIO, E. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In: **Proceedings of the 17th International Conference on Mining Software Repositories**. New York, NY, US: Association for Computing Machinery, 2020. p. 125–136.

PALOMBA, F.; ZAIDMAN, A.; OLIVETO, R.; LUCIA, A. D. An exploratory study on the relationship between changes and refactoring. In: **2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)**. Buenos Aires, Argentina: IEEE, 2017. p. 176–185.

PANIGRAHI, R.; KUANAR, S. K.; KUMAR, L. Application of naïve bayes classifiers for refactoring prediction at the method level. In: **2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)**. Gunupur, India: IEEE, 2020. p. 1–6.

PANIGRAHI, R.; KUANAR, S. K.; MISRA, S.; KUMAR, L. Class-level refactoring prediction by ensemble learning with various feature selection techniques. **Applied Sciences**, v. 12, n. 23, p. 12217, 2022.

PENTA, M. D.; BAVOTA, G.; ZAMPETTI, F. On the relationship between refactoring actions and bugs: a differentiated replication. In: **Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, US: Association for Computing Machinery, 2020. p. 556–567.

PÉREZ, B.; CASTELLANOS, C.; CORREAL, D.; RIOS, N.; FREIRE, S.; SPÍNOLA, R.; SEAMAN, C.; IZURIETA, C. Technical debt payment and prevention through the lenses of software architects. **Information and Software Technology**, Elsevier, v. 140, p. 106692, 2021.

PERUMA, A.; MKAOUER, M. W.; DECKER, M. J.; NEWMAN, C. D. Contextualizing rename decisions using refactorings, commit messages, and data types. **Journal of Systems and Software**, v. 169, p. 110704, 2020.

PINHEIRO, D.; BEZERRA, C.; UCHôA, A. On the effectiveness of trivial refactorings in predicting non-trivial refactorings. **Journal of Software Engineering Research and Development**, v. 12, n. 1, p. 5–1, 2024.

PINHEIRO, D.; BEZERRA, C. I. M.; UCHOA, A. How do trivial refactorings affect classification prediction models? In: **Proceedings of the 16th Brazilian Symposium on Software Components, Architectures, and Reuse**. New York, NY, US: Association for Computing Machinery, 2022. p. 81–90.

QUINLAN, J. R. **C4.5: Programs for Machine Learning**. 1st. ed. San Francisco, CA, US: Morgan Kaufmann Publishers Inc., 1993.

RANSTAM, J.; COOK, J. A. LASSO regression. **Journal of British Surgery**, Publisher: Oxford University Press, v. 105, n. 10, p. 1348–1348, 2018.

RAWLINGS, J. O.; PANTULA, S. G.; DICKEY, D. A. **Applied regression analysis: a research tool**. New York, NY, US: Springer, 1998.

RISH, I. An empirical study of the naive bayes classifier. In: **IJCAI 2001 workshop on empirical methods in artificial intelligence**. Seattle, Washington, US: [*S. n.*], 2001. v. 3, n. 22, p. 41–46.

RURA, S. **Refactoring aspect-oriented software**. Thesis (Bachelor of Arts) – Williams College, Williamstown, Massachusetts, US, 2003.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. **IBM Journal of Research and Development**, v. 3, p. 210–229, 1959.

SELLITTO, G.; IANNONE, E.; CODABUX, Z.; LENARDUZZI, V.; LUCIA, A.; PALOMBA, F.; FERRUCCI, F. Toward understanding the impact of refactoring on program comprehension. In: **2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)**. Honolulu, HI, USA: IEEE, 2021.

SHARMA, T.; SURYANARAYANA, G.; SAMARTHYAM, G. Challenges to and solutions for refactoring adoption: An industrial perspective. **IEEE Software**, IEEE, v. 32, n. 6, p. 44–51, 2015.

SHULL, F.; SINGER, J.; SJØBERG, D. I. **Guide to advanced empirical software engineering**. [*S. l.*]: Springer, 2007.

SILVA, D.; SILVA, J.; SANTOS, G. J. D. S.; TERRA, R.; VALENTE, M. T. O. Refdiff 2.0: A multi-language refactoring detection tool. **IEEE Transactions on Software Engineering**, IEEE, v. 47, n. 12, p. 2786–2802, 2021.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? Confessions of github contributors. In: **Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering**. New York, NY, USA: Association for Computing Machinery, 2016. p. 858–870.

SINGH, S.; KAUR, S. A systematic literature review: Refactoring for disclosing code smells in object oriented software. **Ain Shams Engineering Journal**, Elsevier, v. 9, n. 4, p. 2129–2151, 2018.

SMIARI, P.; BIBI, S.; AMPATZOGLOU, A.; ARVANITOU, E.-M. Refactoring embedded software: A study in healthcare domain. **Information and Software Technology**, Elsevier, v. 143, p. 106760, 2022.

SOBRINHO, E. V. de P.; LUCIA, A. D.; MAIA, M. de A. A systematic literature review on bad smells–5 w's: which, when, what, who, where. **IEEE Transactions on Software Engineering**, IEEE, v. 47, n. 1, p. 17–66, 2018.

SONG, X. Y.; DAO, N.; BRANCO, P. Distsmogn: Distributed smogn for imbalanced regression problems. In: **Fourth International Workshop on Learning with Imbalanced Domains: Theory and Applications**. Grenoble, France: PMLR, 2022. p. 38–52.

SPADINI, D.; ANICHE, M.; BACCHELLI, A. PyDriller: Python framework for mining software repositories. In: **Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, New York, US: ACM Press, 2018. p. 908–911.

SRIVASTAVA, A. K.; SRIVASTAVA, V. K.; ULLAH, A. The coefficient of determination and its adjusted version in linear regression models. **Econometric reviews**, Taylor & Francis, v. 14, n. 2, p. 229–240, 1995.

STEHMAN, S. V. Selecting and interpreting measures of thematic classification accuracy. **Remote sensing of Environment**, Elsevier, v. 62, n. 1, p. 77–89, 1997.

SU, X.; YAN, X.; TSAI, C.-L. Linear regression. **Wiley Interdisciplinary Reviews: Computational Statistics**, v. 4, n. 3, p. 275–294, 2012.

TABASSUM, N.; NAMOUN, A.; ALYAS, T.; TUFAIL, A.; TAQI, M.; KIM, K.-H. Classification of bugs in cloud computing applications using machine learning techniques. **Applied Sciences**, v. 13, n. 5, p. 2880, 2023.

TAN, A. J. J.; CHONG, C. Y.; ALETI, A. Rearrange: Effort estimation approach for software clustering-based remodularisation. **Information and Software Technology**, v. 176, p. 107567, 2024.

TORGO, L.; BRANCO, P.; RIBEIRO, R. P.; PFAHRINGER, B. Resampling strategies for regression. **Expert Systems**, v. 32, n. 3, p. 465–476, 2015.

TSANTALIS, N.; CHAIKALIS, T.; CHATZIGEORGIOU, A. Ten years of jdeodorant: Lessons learned from the hunt for smells. In: **2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)**. Campobasso, Italy: IEEE, 2018. p. 4–14.

TSANTALIS, N.; KETKAR, A.; DIG, D. Refactoringminer 2.0. **IEEE Transactions on Software Engineering**, v. 48, n. 3, p. 930–950, 2020.

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**. New York, NY, US: ACM, 2018. p. 483–494.

UCHÔA, A.; BARBOSA, C.; OIZUMI, W.; BLENÍLIO, P.; LIMA, R.; GARCIA, A.; BEZERRA, C. How does modern code review impact software design degradation? An in-depth empirical study. In: **2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. Adelaide, SA, Australia: IEEE, 2020. p. 511–522.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. Berlin, Germany: Springer, 2012.

YAMASHITA, A.; MOONEN, L. Do code smells reflect important maintainability aspects? In: **2012 28th IEEE international conference on software maintenance (ICSM)**. Trento, Italy: IEEE, 2012. p. 306–315.

ZARNEKOW, R.; BRENNER, W. Distribution of cost over the application lifecycle - A multi-case study. **ECIS 2005 Proceedings**, p. 26, 2005.

ZHOU, Z.-H. **Machine learning**. Berlin, Germany: Springer Nature, 2021.

ZOU, H.; HASTIE, T. Regularization and variable selection via the elastic net. **Journal of the Royal Statistical Society Series B: Statistical Methodology**, Oxford University Press, v. 67, n. 2, p. 301–320, 2005.

APPENDIX A – SURVEY FOR DEVELOPERS

Survey for collect detailed information from developers about their practices and perceptions regarding code metrics and the refactoring process.

Research: Refactoring and Software Quality

Dear Developer,

You are being invited to participate and contribute to research on Refactoring and Software Quality, carried out as part of an academic study for a dissertation.

Research Objective
Our goal is to collect detailed information from developers about their practices and perceptions regarding code metrics and the refactoring process. The research seeks to better understand which metrics are most relevant for each aspect in the refactoring process. The data collected will be used to identify trends, challenges and opportunities for improvement in development practices, promoting the production of more efficient, sustainable and high-quality programs. In this context, we aim at creating a index, based on these practices and perceptions, for predicting, indicating and assessing code refactorings.

Research Participation
To participate in this research, you must be over 18 years old and have already worked in the area of software development. Your participation will consist of answering a questionnaire about aspects of the source code that can contribute to refactoring prediction. The questionnaire will be completed online and will take approximately 7 minutes to complete.

Secrecy and Confidentiality
We guarantee the secrecy and confidentiality of all information you provide during your participation in the research. The data will be treated anonymously and used exclusively for academic purposes, contributing to the advancement of knowledge in the area of software quality and refactoring.

Researchers
The researchers responsible are: Darwin Pinheiro (darwinfederal@alu.ufc.br), Carla Bezerra (carlailane@ufc.br), Anderson Uchôa (andersonuchoa@ufc.br) and Alessandro Garcia (afgarcia@inf.puc-rio.br)

We thank you in advance for your collaboration and participation in this important research.

* Indica uma pergunta obrigatória

1. E-mail *

____

2. The **Free and Informed Consent Form (TCLE)** aims to ensure your rights as a participant and is available at this link. Please read carefully and calmly, trying to fully understand the research proposal. There will be no penalty or loss if you do not wish to participate or withdraw your authorization at any time.* *

   After receiving clarifications about the nature of the research, its objectives and methods, I declare that I am over 18 years old and agree to participate through the participation form, in accordance with the provisions of the General Data Protection Law (Law No. 13,709/2018) .

   In case of a negative response, the form will be closed, thus ensuring respect for the autonomy and privacy of participants, as established by current legislation.

   Marcar apenas uma oval.

   ◯ Yes *Pular para a pergunta 3*

   ◯ No

Pular para a pergunta 3

CHARACTERIZATION PROFILE

In this section, we will collect basic information and personal data to better understand your profile.

3. How many years of experience do you have in software development? Please provide your best estimate. *

   ____

4. Do you want to share some additional information about your experience?

   ____
   ____
   ____
   ____
   ____

5. Which best roles describes your current activities in software development projects? *

Marque todas que se aplicam.

- [ ] Frontend Developer
- [ ] Backend Developer
- [ ] Fullstack
- [ ] Devops
- [ ] UX/UI Designer
- [ ] Product Owner
- [ ] Tech Leader
- [ ] Quality Assurance (QA)
- [ ] Scrum Master
- [ ] Project Manager
- [ ] Stakeholder
- [ ] Outro: ____________________

6. What level of formal education have you completed? *

Marcar apenas uma oval.

- ( ) High School
- ( ) Technical Course
- ( ) Graduation
- ( ) Specialization
- ( ) Master's Degree
- ( ) Doctorate Degree
- ( ) Outro: ____________________

CHARACTERIZATION OF THE EXPERIENCE WITH SOFTWARE REFACTORING

In this section, we will collect information about the characteristics relevant to software quality.

7. How familiar are you with refactoring software? *

Marcar apenas uma oval.

- Never heard of it.
- I heard about it, but never used it.
- I know it, but I hardly use it.
- I have knowledge and sometimes I use it.
- I have in-depth knowledge and use it frequently.
- I'm an expert on the subject and I always use it.
- Outro: ____________________

8. In the past semester, how often have you performed any code refactoring? *

Marcar apenas uma oval.

- Never
- Rarely
- Sometimes
- Often
- Always
- Outro: ____________________

9. Do you want to share some additional information about your experience with code refactoring?

10. Regarding the **COMPLEXITY ASPECT** of the code being refactored, which refers to the technical difficulty and level of knowledge required to modify the code (More complex refactorings typically require a deeper understanding of the system and may involve substantial changes to several parts of the code). *

What level of importance do you believe each code metric below has for identifying refactoring and/or improving software quality (with refactoring), on a scale from 1 to 5, where 1 is not at all important and 5 is very important?

If you prefer, access this link containing definitions and examples to help you remember.

Marque todas que se aplicam.

	1	2	3	4	5
FAN-IN (Number of input dependencies)	☐	☐	☐	☐	☐
FAN-OUT (Number of output dependencies)	☐	☐	☐	☐	☐
CBO (Coupling Between Objects)	☐	☐	☐	☐	☐
DIT (Depth Inheritance Tree)	☐	☐	☐	☐	☐
NOC (Number of Children)	☐	☐	☐	☐	☐
NOSI (Number of Static Invocations)	☐	☐	☐	☐	☐
RFC (Response for a Class)	☐	☐	☐	☐	☐
WMC (Weighted Methods per Class or	☐	☐	☐	☐	☐

Class or McCabe's Complexity)					
LCOM (Normalized Lack of Cohesion of Methods)	☐	☐	☐	☐	☐
TCC (Tight Class Cohesion)	☐	☐	☐	☐	☐
LCC (Loose Class Cohesion)	☐	☐	☐	☐	☐

11. **COMPLEXITY ASPECT**
    Do you know of any other items in this aspect that could be relevant to improving the quality of the software and/or have any relationship with the aspect?

    ____

12. Regarding the **SPEED ASPECT** of the code being refactored, which refers to the frequency of implementing operations and changing the code size in lines (Fast refactorings are characterized by the number of operations performed in the interval of 2 to 4 weeks). *

What level of importance do you believe each code metric below has for identify refactoring implementation speed and/or improving software quality (with refactoring), on a scale from 1 to 5, where 1 is not at all important and 5 is very important?

If you prefer, access this link containing definitions and examples to help you remember.

Marque todas que se aplicam.

	1	2	3	4	5
LOCC (Lines of Code Changed)	☐	☐	☐	☐	☐
FOC7 (Frequency of Commit in 7days)	☐	☐	☐	☐	☐
FOC14 (Frequency of Commit in 14days)	☐	☐	☐	☐	☐
FOC21 (Frequency of Commit in 21days)	☐	☐	☐	☐	☐
FOC28 (Frequency of Commit in 28days)	☐	☐	☐	☐	☐

13. **<u>SPEED ASPECT</u>**
Do you know of any other items in this aspect that could be relevant to improving the quality of the software and/or have any relationship with the aspect?

____

14. Regarding the **<u>RISK ASPECT</u>** of the code being refactored, which refers to the likelihood that a refactoring will cause new problems or failures in the code (High-risk refactorings can result in more bugs or performance issues after the modification). *

What level of importance do you believe each occurrences of issues below has for identifying implementation risk and/or improving software quality (with refactoring), on a scale from 1 to 5, where 1 is not at all important and 5 is very important?

If you prefer, access this link containing definitions and examples to help you remember.

Marque todas que se aplicam.

	1	2	3	4	5
Issues of Best Pratices	☐	☐	☐	☐	☐
Issues of Design	☐	☐	☐	☐	☐
Issues of Error Proneness	☐	☐	☐	☐	☐
Issues of Multithreading	☐	☐	☐	☐	☐
Issues of Performance	☐	☐	☐	☐	☐
Issues of Security	☐	☐	☐	☐	☐

15. **RISK ASPECT**
Do you know of any other items in this aspect that could be relevant to improving the quality of the software and/or have any relationship with the aspect?

16. Regarding the **Other Structural Metrics** of the code being refactored, which refers to quantitative measures of the internal structure of the code. *

What level of importance do you believe each code metric below has for identifying refactoring and/or improving software quality, on a scale from 1 to 5, where 1 is not at all important and 5 is very important?

If you prefer, access this link containing definitions and examples to help you remember.

Marque todas que se aplicam.

	1	2	3	4	5
Path of Class	☐	☐	☐	☐	☐
File Type	☐	☐	☐	☐	☐
Number of Method	☐	☐	☐	☐	☐
Number of Visible Method	☐	☐	☐	☐	☐
Quantity of Returns	☐	☐	☐	☐	☐
Quantity of Loops	☐	☐	☐	☐	☐
Quantity of Comparisons	☐	☐	☐	☐	☐
Quantity of Try/Catches	☐	☐	☐	☐	☐
Quantity of Parenthesized Expressions	☐	☐	☐	☐	☐
String Literals	☐	☐	☐	☐	☐
Quantity of Number	☐	☐	☐	☐	☐
Quantity of Math Operations	☐	☐	☐	☐	☐

Quantity Method Invocations	☐	☐	☐	☐	☐
Quantity of Fields	☐	☐	☐	☐	☐
Usage of Each Field	☐	☐	☐	☐	☐
Quantity of Variables	☐	☐	☐	☐	☐
Usage of Each Variable	☐	☐	☐	☐	☐
Max Nested Blocks	☐	☐	☐	☐	☐
Quantity of Anonymous Classes	☐	☐	☐	☐	☐
Inner Classes	☐	☐	☐	☐	☐
Lambda Expressions	☐	☐	☐	☐	☐
Number of Unique Words	☐	☐	☐	☐	☐
Number of Log Statements	☐	☐	☐	☐	☐
Has Javadoc	☐	☐	☐	☐	☐
Quantity of Modifiers	☐	☐	☐	☐	☐

17. **Other Structural Metrics**
Do you know of any other items that could be relevant to improving the quality of the software and/or have any relationship with the structural metrics?

DOCUMENT OF DEFINITIONS

Coupling and Dependencies:

FAN-IN:

- Definition: It counts the number of input dependencies of a class, i.e. how many other classes use that class.
- Example: If three classes (X, Y, Z) call methods from class A, the FAN-IN of A is 3.

FAN-OUT:

- Definition: Counts the number of output dependencies of a class, that is, how many other classes are used by this class.
- Example:If class A uses methods from three classes (B, C, D), A's FAN-OUT is 3.

CBO (Coupling Between Objects):

- Definition: Measures the coupling between objects, that is, how many classes are used by a given class, measuring both input and output.
- Example: If class A uses methods from five other classes (B, C, D, E, F) and class B uses a method from class A, A's CBO is 6.

Inheritance:

DIT (Depth Inheritance Tree):

- Definition: Counts the number of "parents" of a class, that is, the depth of the class in the inheritance tree.
- Example: If A inherits from B, which inherits from C, which inherits from D, the DIT of A is 3.

NOC (Number of Children):

- Definition: Counts the number of immediate subclasses a class has.
- Example: If class A has three direct subclasses (B, C, D), the NOC of A is 3.

Invocations

NOSI (Number of Static Invocations):

- Definition: Counts the number of invocations to static methods within a class.
- Example: If class A calls static methods B's staticMethod1 and C's staticMethod2 four times in total, A's NOSI is 4.

RFC (Response for a Class):

- Definition: Counts the number of unique methods invoked within a class.
- Example: If class A calls methods method1, method2 of B and method3 of C, the RFC of A is 3.

Complexity and Cohesion

WMC (Weight Method Class):

- Definition: Counts the number of branching statements in a class, such as if, for, while.
- Example: If class A has five methods with a total of 10 branching statements (3 if, 2 for, 2 while, 3 switch), A's WMC is 10.

LCOM:

- Definition: It is a metric that measures cohesion within a class, that is, how well the methods of a class work together. Cohesive classes have methods that operate on the same instance attributes(attributes), while classes with low cohesion have methods that operate on different attributes.
- Example: If class A has three methods where none share variables, the LCOM is high, indicating low cohesion, which is a bad indicator.

TCC (Tight Class Cohesion):

- Definition: It is a cohesion metric that measures the proportion of visible (or public) method pairs of a class that are directly connected through instance variables. In other words, TCC evaluates the degree to which the methods of a class work together, using the same instance variables. A value from 0 to 1, where 1 means high cohesion.
- Example: If 80% of a class's visible methods are directly connected, the TCC is 0.8.

LCC (Loose Class Cohesion):

- Definition: Similar to TCC, but includes indirect connections between visible methods.
- Example: If a method A makes a call to other methods B and C that are directly connected to instance variables, this method A will also be counted.

Lines of Code Changed

LOCC (Line of Code Changed):

- Definition: Counts the number of lines of code changed, excluding empty lines and comments.
- Example: If class A has 15 lines of code after refactoring, excluding empty lines and comments, its LOCC is 15.

.

Frequency of Refactoring

FOC7 (Frequency of Commit in 7 days)

- Definition: Definition refers to the frequency with which commits are made to a refactoring-focused version control repository, within a 7-day interval.
- Example: this metric can be analyzed at different granularities, such as sprints of 1 to 4 weeks.

FOC14 (Frequency of Commit in 14 days)

- Definition: Definition refers to the frequency with which commits are made to a refactoring-focused version control repository, within a 14-day interval.
- Example: this metric can be analyzed at different granularities, such as sprints of 1 to 4 weeks.

FOC21 (Frequency of Commit in 21 days)

- Definition: Definition refers to the frequency with which commits are made to a refactoring-focused version control repository, within a 21-day interval.
- Example: this metric can be analyzed at different granularities, such as sprints of 1 to 4 weeks.

FOC28 (Frequency of Commit in 28 days)

- Definition: Definition refers to the frequency with which commits are made to a refactoring-focused version control repository, within a 28-day interval.
- Example: this metric can be analyzed at different granularities, such as sprints of 1 to 4 weeks.

Other Structural Metrics

Path of class

- Definition: The "Path of Class" generally refers to the file path on the file system where the class definition is located.ExampleDefinition: The "Path of Class" generally refers to the file path on the file system where the class definition is located.
- Example: /src/main/java/com/example/ExampleClass.java

File Type

- Definition: It is the type of file, class or interface, from which the metrics were extracted.
- Example: class, interface, innerclass, enum e anonymous.

Number of Method:

- Definition: Counts the total number of methods in a class.
- Example: If class A has 10 methods, its Number of Method is 10.

Number of Visible Method:

- Definition: Counts the number of non-private methods in a class.
- Example: If class A has 10 methods, of which 7 are public or protected, the Number of Visible Method is 7.

Quantity of Returns:

- Definition: Counts the number of return statements in a class.
- Example: If class A has 5 return instructions, the Quantity of Returns is 5.

Quantity of Loops:

- Definition: Counts the number of loops (for, while, do while, enhanced for) in a class.
- Example: If class A has 3 for and 2 while loops, the Quantity of Loops is 5.

Quantity of Comparisons:

- Definition: Counts the number of comparisons (== and !=) in a class.
- Example: If class A has 4 comparisons == and 2 !=, the Quantity of Comparisons is 6.

Quantity of Try/Catches:

- Definition: Counts the number of try/catch blocks in a class.
- Example: If class A has 3 try/catch blocks, the Quantity of Try/Catches is 3.

Quantity of Parenthesized Expressions:

- Definition: Counts the number of parenthesized expressions in a class.
- Example: If class A has 10 parenthesized expressions, the Quantity of Parenthesized Expressions is 10.

String Literals:

- Definition: Counts the number of string literals in a class. Example: If class A has 8 string literals, the String Literals is 8.

Quantity of Number:

- Definition: Counts the number of numeric literals (int, long, double, float) in a class.
- Example: If class A has 12 numeric literals, the Quantity of Number is 12.

Quantity of Math Operations:

- Definition: Counts the number of mathematical operations (multiplication, division, remainder, addition, subtraction, shift left, shift right) in a class.
- Example: If class A has 7 mathematical operations, the Quantity of Math Operations is 7.

Method Invocations:

- Definition: Counts all direct method invocations in a class, with variations for local and indirect invocations.
- Example: If class A calls 15 methods directly, the Method Invocations is 15. Quantity of Fields:

Quantity of Variables:

- Definition: Counts the number of variables declared in a class.
- Example: If class A has 15 variables, the Quantity of Variables is 15.

Usage of Each Variable:

- Definition: Measures the frequency of use of each variable within each method.
- Example: If a variable y is used 3 times in different methods, its usage is 3.

Max Nested Blocks:

- Definition: Counts the largest number of nested blocks in a class.
- Example: If class A has a maximum of 4 nested blocks, the Max Nested Blocks is 4.

Quantity of Anonymous Classes, Inner Classes, and Lambda Expressions:

- Definition: Counts the number of anonymous classes, inner classes, and lambda expressions in a class.
- Example: If class A has 2 anonymous classes, 1 inner class and 3 lambda expressions, the total returned is 6.

Number of Unique Words:

- Definition: Counts the number of unique words in the source code, separating names by camel case and underscore.
- Example: If class A has 50 unique words, the Number of Unique Words is 50.

Number of Log Statements:

- Definition: Counts the number of log statements (log.info, log.debug, etc.) in the source code.
- Example: If class A has 8 log statements, the Number of Log Statements is 8

Has Javadoc:

- Definition: Indicates with a boolean whether a method has javadoc.
- Example: If a method m has javadoc, Has Javadoc will be true.

Quantity of Modifiers:

- Definition: Counts modifiers (public, abstract, private, protected, native) of classes/methods.

- Example: If class A has 3 public, 2 private and 1 protected methods, the Quantity of Modifiers is 6.

Code Smell

Issues of Best Practices
- Definition: Set of problems related to best programming practices.
- Example: It covers a variety of issues, such as detecting unused variables, using static methods, and best practices for synchronization.

Issues of Design

- Description: Set of detected problems that aim to improve code design, promoting cohesion and modularity.
- Example: CouplingBetweenObjects, god class, and other code smells.

Potential points of failure

Issues of Error Prone

- Description: Set of detected problems based on code patterns that are prone to errors.
- Example: Detects empty catch blocks. Detects bodyless if statements.

Issues of Multithreading

- Description: Set of detected problems based on parallel programming and multithreading
- Example: Detects unsynchronized uses of SimpleDateFormat

Issues of Performance

- Description: Set of problems detected with a focus on improving code performance.
- Example: Detects objects that are instantiated repeatedly in loops.

Issues of Security

- Description: Set of problems detected with a focus on identifying and correcting security vulnerabilities in the code.
- Example: Detects hard coded passwords in the code. Detects insufficient string validation.

Free and Informed Consent Form (TCLE)

This document, called Free and Informed Consent Form (TCLE), aims to ensure your rights as a participant and will be made available electronically.

Please read carefully and calmly, trying to fully understand the research proposal. There will be no penalty or loss if you do not wish to participate or withdraw your authorization at any time.

To participate in this research you must be over 18 years old and work in the area of software development. Your participation will consist of answering a questionnaire about code metrics that are important in refactoring. The questionnaire will be answered online and will take approximately 7 minutes to complete. We guarantee the secrecy and confidentiality of all information you provide during your participation in the research.

INFORMATION ABOUT THIS RESEARCH:

Objectives: The objective of this research is to complement a study carried out to compose the master's thesis, collecting detailed information from developers about their practices and perceptions in relation to code metrics and the refactoring process.

Importance of the study: The project seeks to better understand how developers measure code quality, identify areas that need improvement and implement refactorings. The data collected will be used to identify trends, challenges and opportunities for improvement in development practices, with the aim of promoting the production of more efficient, sustainable and high-quality code.

Procedures and methodologies: This study is aimed at any developer profile that has some knowledge about code metrics, refactoring or even cares about software quality and/or studies that aim to improve this area. Your participation in the study will consist of answering an online questionnaire. The confidentiality and anonymity of participants will be strictly maintained, encouraging frank and honest responses to provide valuable insights into the topic in question.

Data processing: Throughout the process, respect for privacy and ethics standards will be ensured, as recommended by the General Data Protection Law (Law nº 13,709/2018), ensuring that participants' information is treated confidentially. Initially, the data collected through the de-identified online questionnaire will be anonymized, removing any information that could directly identify the participants, in accordance with the guidelines of the aforementioned legislation. The data will then be organized and coded to facilitate analysis. The results will be presented in an aggregated and non-individually identifiable form, thus preserving the privacy of participants in accordance with applicable legal provisions.
How to contact the researchers: Whenever you wish, you can contact us to obtain information about this research project, your participation or other matters related to the

research. The researchers responsible are: Darwin Pinheiro (darwinfederal@alu.ufc.br), Carla Bezerra (carlailane@ufc.br) and Anderson Uchôa (andersonuchoa@ufc.br).

GUARANTEES TO PARTICIPANTS:

Right to refuse to participate: At any time, you can refuse to participate and withdraw from the research, without embarrassment, penalties or any loss. The information and materials obtained in this research will not be used for purposes other than within the context of this research.

Confidentiality and privacy: We guarantee the complete confidentiality and anonymity of your responses. All sensitive information, such as company or other people's names, will be completely anonymized.
Responsibility of the Researcher: We ensure to provide this document to the research participant and use the data obtained exclusively for the purposes described in this document or in accordance with the consent given by the participant.

APPENDIX B – SURVEY FOR EXPERTS

Survey to collect detailed information from experts about their perceptions regarding the triviality of refactorings.

Survey: Refactoring's Triviality

Dear Developer,

You are being invited to participate and contribute to a survey on Refactoring and Software Quality, conducted as part of an academic study for a dissertation.

Survey Objective
Our goal is to collect information about your perceptions of some aspects when performing a refactoring operation. The survey seeks to better understand which aspects are most relevant to the refactoring process. The data collected will be used to identify trends, challenges and opportunities for improvement in development practices, promoting the production of more efficient, sustainable and high-quality programs.

Survey Participation
To participate in this survey, you must be over 18 years old and have already worked in the area of software development. Your participation will consist of answering a questionnaire about the difficulty of performing a certain refactoring operation. The questionnaire will be filled out online and will take approximately 7 minutes to complete.

Secrecy and Confidentiality
We guarantee the secrecy and confidentiality of all information you provide during your participation in the survey. The data will be treated anonymously and used exclusively for academic purposes, contributing to the advancement of knowledge in the area of software quality and refactoring.

Researchers
The researchers responsible are: Darwin Pinheiro (darwinfederal@alu.ufc.br), Carla Bezerra (carlailane@ufc.br), Anderson Uchôa (andersonuchoa@ufc.br) and Alessandro Garcia (afgarcia@inf.puc-rio.br)

We would like to thank you in advance for your collaboration and participation in this important research.

* Indica uma pergunta obrigatória

1. The **Free and Informed Consent Form (TCLE)** aims to ensure your rights as a participant and is available at this link. Please read carefully and calmly, trying to fully understand the research proposal. There will be no penalty or loss if you do not wish to participate or withdraw your authorization at any time.* *

   After receiving clarifications about the nature of the research, its objectives and methods, I declare that I am over 18 years old and agree to participate through the participation form, in accordance with the provisions of the General Data Protection Law (Law No. 13,709/2018) .

   In case of a negative response, the form will be closed, thus ensuring respect for the autonomy and privacy of participants, as established by current legislation.

   Marcar apenas uma oval.

   - Yes *Pular para a pergunta 2*
   - No

Pular para a pergunta 2

CHARACTERIZATION PROFILE

In this section, we will collect basic information and personal data to better understand your profile.

2. What is your e-mail? *

   ____

3. What is your gender? *

   Marcar apenas uma oval.

   - Man
   - Woman
   - Non-binary, gender queer or gender non-conforming
   - I prefer not to respond

4. What is your age? *

Marcar apenas uma oval.

- ◯ Under 18 years old
- ◯ 18-24 years old
- ◯ 25-34 years old
- ◯ 35-44 years old
- ◯ 45-54 years old
- ◯ 55-64 years old
- ◯ 65 years or older
- ◯ Opção 8

5. What is your country? *

____

6. What level of formal education have you completed? *

Marcar apenas uma oval.

- ◯ High School
- ◯ Technical Course
- ◯ Graduation
- ◯ Specialization
- ◯ Master's Degree
- ◯ Doctorate Degree
- ◯ Outro: _______________________________

7. How many years of experience do you have in software development? Please provide your best estimate. *

Marcar apenas uma oval.

- ◯ Less than 1 years
- ◯ 1-3 years
- ◯ 4-6 years
- ◯ 7-14 years
- ◯ More than 15 years

8. Which best roles describes your current activities in software development projects? *

Marque todas que se aplicam.

- ☐ Frontend Developer
- ☐ Backend Developer
- ☐ Fullstack
- ☐ Devops
- ☐ UX/UI Designer
- ☐ Product Owner
- ☐ Tech Leader
- ☐ Quality Assurance (QA)
- ☐ Scrum Master
- ☐ Project Manager
- ☐ Stakeholder
- ☐ Outro: ______________________

9. Do you want to share some additional information about your experience?

____

____

____

____

____

CHARACTERIZATION OF THE EXPERIENCE WITH SOFTWARE REFACTORING

In this section, we will collect information about the characteristics relevant to software quality.

10. Do you have (or have you had) any experience in the software development process related to refactoring? *

Marcar apenas uma oval.

- ◯ I have worked (or work) with refactoring.
- ◯ My group or development team works with refactoring, but I am indirectly involved.
- ◯ I have never had direct or indirect experience with refactoring.

11. How familiar are you with refactoring software? *

Marcar apenas uma oval.

- ◯ Never heard of it.
- ◯ I heard about it, but never used it.
- ◯ I know it, but I hardly use it.
- ◯ I have knowledge and sometimes I use it.
- ◯ I have in-depth knowledge and use it frequently.
- ◯ I'm an expert on the subject and I always use it.
- ◯ Outro: ____________________

12. How often do you perform any code refactoring? *

Marcar apenas uma oval.

- ⬭ Daily
- ⬭ Weekly
- ⬭ Monthly
- ⬭ Annually
- ⬭ Quartlety
- ⬭ Not applicable

13. Do you want to share some additional information about your experience with code refactoring?

Evaluation of the refactoring operations triviality

Consider the following definitions to answer the next questions:

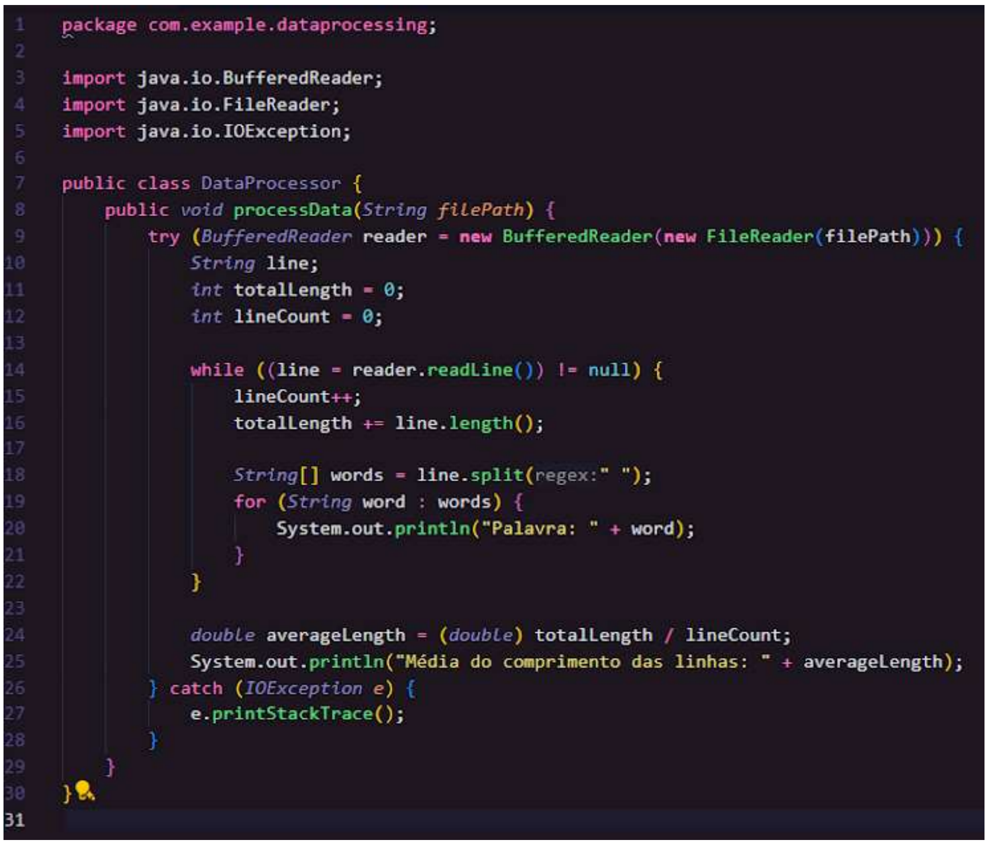
Regarding the **COMPLEXITY ASPECT** of the code being refactored, which refers to the technical difficulty and level of knowledge required to modify the code (More complex refactorings typically require a deeper understanding of the system and may involve substantial changes to several parts of the code).

Regarding the **SPEED ASPECT** of the code being refactored, which refers to the frequency of implementing operations and changing the code size in lines (Fast refactorings are characterized by the number of operations performed in the interval of 2 to 4 weeks).

Regarding the **RISK ASPECT** of the code being refactored, which refers to the likelihood that a refactoring will cause new problems or failures in the code (High-risk refactorings can result in more bugs or performance issues after the modification).

14. **1/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK aspects, how would you rate the level of difficulty for each aspect when applying the **EXTRACT METHOD** refactoring operation to the following source code? *

If you prefer, you can see the code and explanation at this link.



```java
package com.example.dataprocessing;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class DataProcessor {
    public void processData(String filePath) {
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            int totalLength = 0;
            int lineCount = 0;

            while ((line = reader.readLine()) != null) {
                lineCount++;
                totalLength += line.length();

                String[] words = line.split(regex:" ");
                for (String word : words) {
                    System.out.println("Palavra: " + word);
                }
            }

            double averageLength = (double) totalLength / lineCount;
            System.out.println("Média do comprimento das linhas: " + averageLength);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

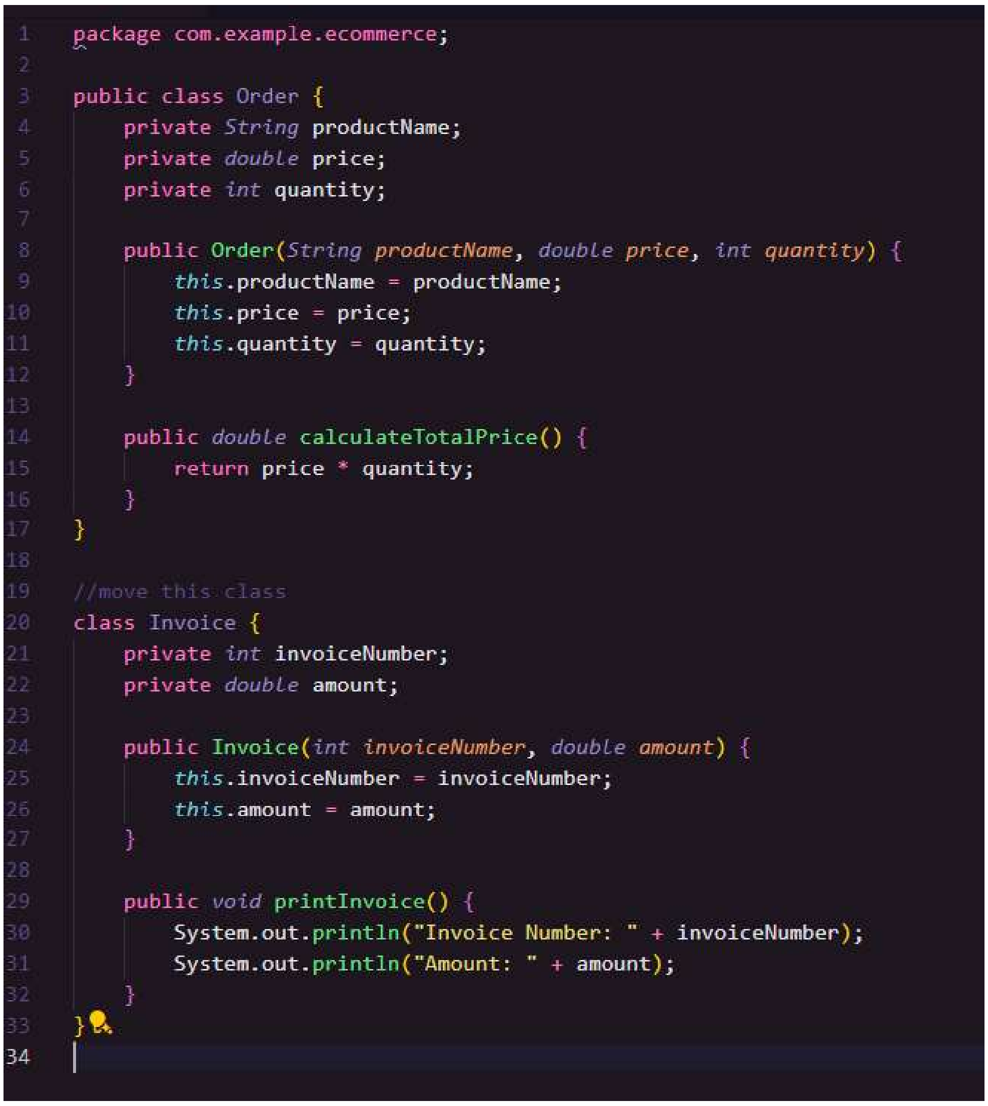
Marcar apenas uma oval por linha.

	Very Difficult	Difficult	Neutral	Easy	Very easy
COMPLEXITY ASPECT	⬭	⬭	⬭	⬭	⬭
SPEED ASPECT	⬭	⬭	⬭	⬭	⬭
RISK ASPECT	⬭	⬭	⬭	⬭	⬭

15. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

16. **2/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK * aspects, how would you rate the level of difficulty for each aspect when applying the **MOVE CLASS** refactoring operation to the following source code?

If you prefer, you can see the code and explanation at this link.



```java
package com.example.ecommerce;

public class Order {
    private String productName;
    private double price;
    private int quantity;

    public Order(String productName, double price, int quantity) {
        this.productName = productName;
        this.price = price;
        this.quantity = quantity;
    }

    public double calculateTotalPrice() {
        return price * quantity;
    }
}

//move this class
class Invoice {
    private int invoiceNumber;
    private double amount;

    public Invoice(int invoiceNumber, double amount) {
        this.invoiceNumber = invoiceNumber;
        this.amount = amount;
    }

    public void printInvoice() {
        System.out.println("Invoice Number: " + invoiceNumber);
        System.out.println("Amount: " + amount);
    }
}

```

Marcar apenas uma oval por linha.

	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY ASPECT	⬭	⬭	⬭	⬭	⬭
SPEED ASPECT	⬭	⬭	⬭	⬭	⬭

RISK ASPECT

RISK ASPECT	○	○	○	○	○

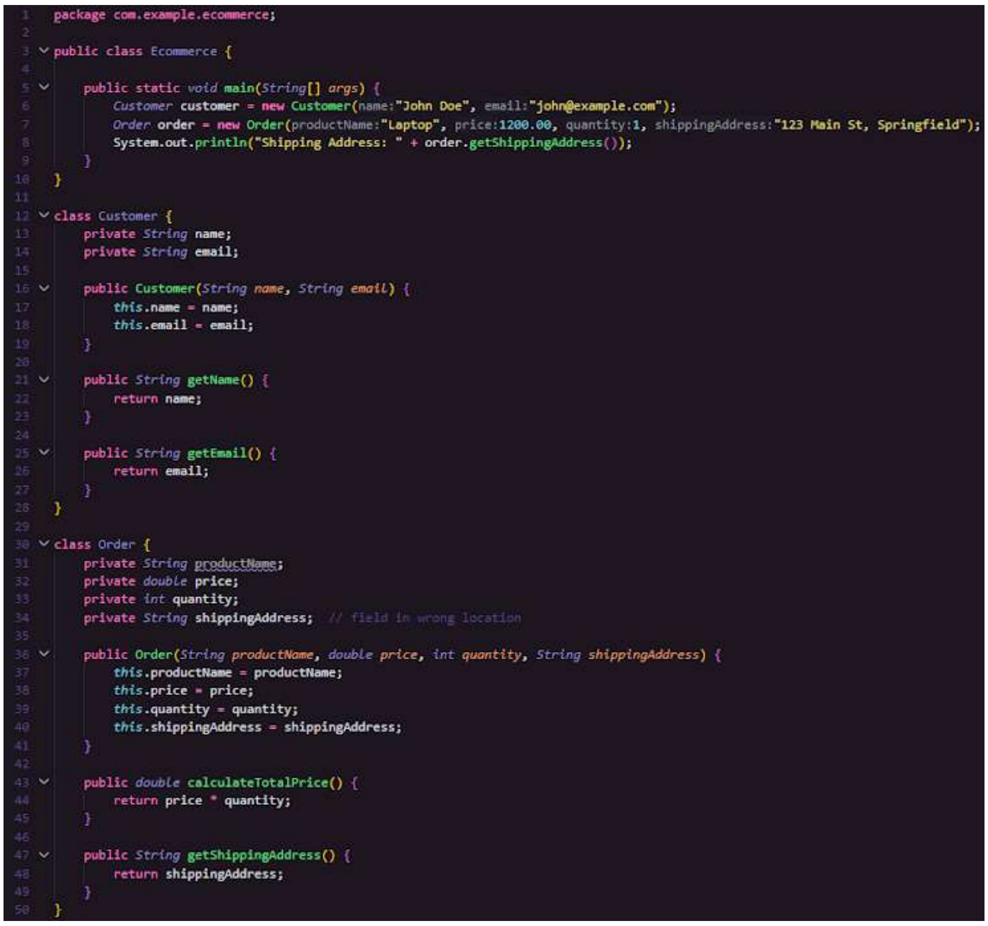
17. **Other Informations**

Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

____

18. **3/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK aspects, how would you rate the level of difficulty for each aspect when applying the **MOVE ATTRIBUTE** refactoring operation to the following source code? *

If you prefer, you can see the code and explanation at this link.



```java
package com.example.ecommerce;

public class Ecommerce {

    public static void main(String[] args) {
        Customer customer = new Customer(name:"John Doe", email:"john@example.com");
        Order order = new Order(productName:"Laptop", price:1200.00, quantity:1, shippingAddress:"123 Main St, Springfield");
        System.out.println("Shipping Address: " + order.getShippingAddress());
    }
}

class Customer {
    private String name;
    private String email;

    public Customer(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }
}

class Order {
    private String productName;
    private double price;
    private int quantity;
    private String shippingAddress;  // field in wrong location

    public Order(String productName, double price, int quantity, String shippingAddress) {
        this.productName = productName;
        this.price = price;
        this.quantity = quantity;
        this.shippingAddress = shippingAddress;
    }

    public double calculateTotalPrice() {
        return price * quantity;
    }

    public String getShippingAddress() {
        return shippingAddress;
    }
}
```

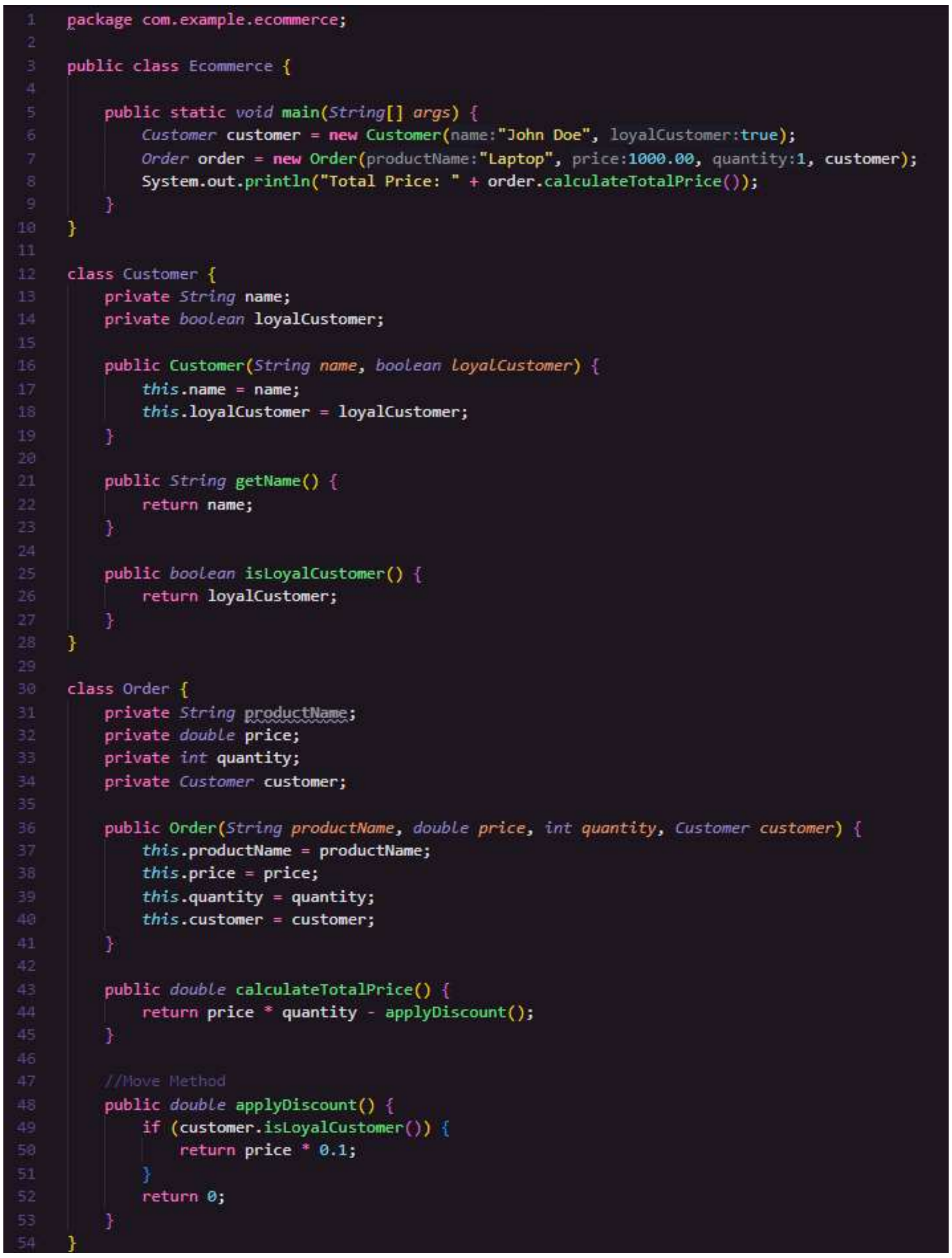
Marcar apenas uma oval por linha.

	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY ASPECT	⬭	⬭	⬭	⬭	⬭
SPEED ASPECT	⬭	⬭	⬭	⬭	⬭
RISK ASPECT	⬭	⬭	⬭	⬭	⬭

19. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

20. **4/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK aspects, how would you rate the level of difficulty for each aspect when applying the **MOVE METHOD** refactoring operation to the following source code? *

If you prefer, you can see the code and explanation at this link.



```java
package com.example.ecommerce;

public class Ecommerce {

    public static void main(String[] args) {
        Customer customer = new Customer(name:"John Doe", loyalCustomer:true);
        Order order = new Order(productName:"Laptop", price:1000.00, quantity:1, customer);
        System.out.println("Total Price: " + order.calculateTotalPrice());
    }
}

class Customer {
    private String name;
    private boolean loyalCustomer;

    public Customer(String name, boolean loyalCustomer) {
        this.name = name;
        this.loyalCustomer = loyalCustomer;
    }

    public String getName() {
        return name;
    }

    public boolean isLoyalCustomer() {
        return loyalCustomer;
    }
}

class Order {
    private String productName;
    private double price;
    private int quantity;
    private Customer customer;

    public Order(String productName, double price, int quantity, Customer customer) {
        this.productName = productName;
        this.price = price;
        this.quantity = quantity;
        this.customer = customer;
    }

    public double calculateTotalPrice() {
        return price * quantity - applyDiscount();
    }

    //Move Method
    public double applyDiscount() {
        if (customer.isLoyalCustomer()) {
            return price * 0.1;
        }
        return 0;
    }
}
```

Marcar apenas uma oval por linha.

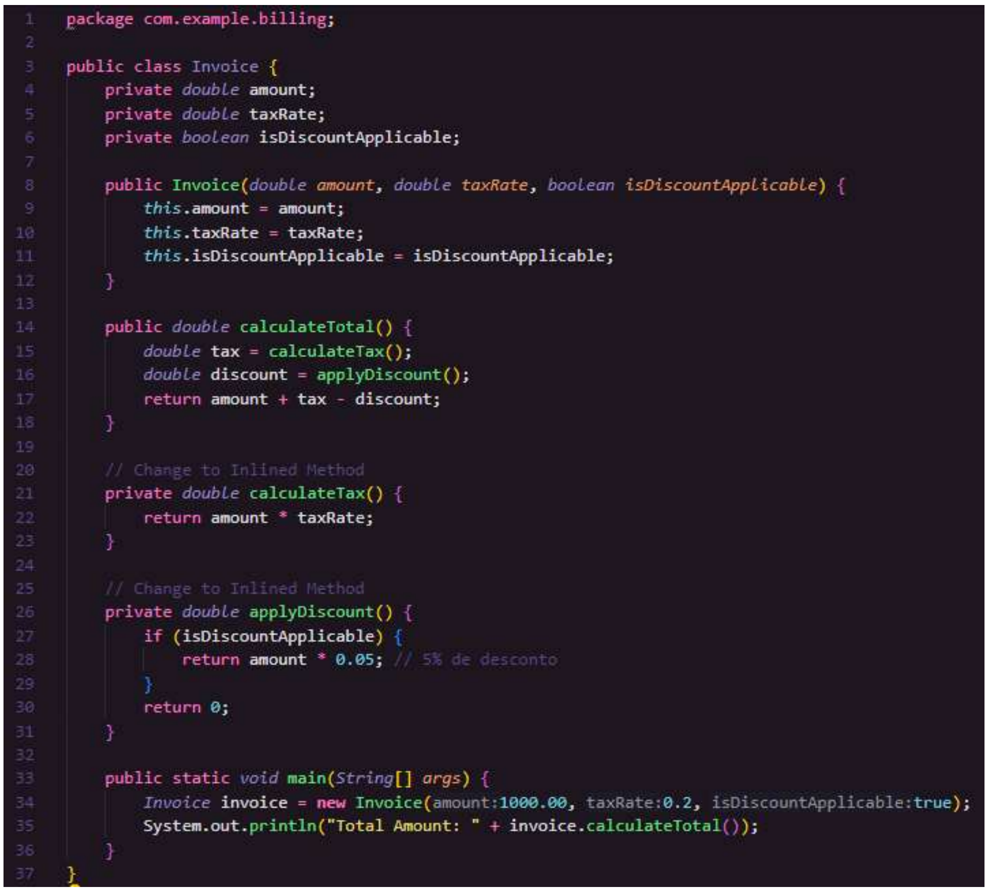
	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY					

COMPLEXITY ASPECT	◯	◯	◯	◯	◯
SPEED ASPECT	◯	◯	◯	◯	◯
RISK ASPECT	◯	◯	◯	◯	◯

21. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

22. **5/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK aspects, how would you rate the level of difficulty for each aspect when applying the **INLINE METHOD** refactoring operation to the following source code? *

If you prefer, you can see the code and explanation at this link.



```java
package com.example.billing;

public class Invoice {
    private double amount;
    private double taxRate;
    private boolean isDiscountApplicable;

    public Invoice(double amount, double taxRate, boolean isDiscountApplicable) {
        this.amount = amount;
        this.taxRate = taxRate;
        this.isDiscountApplicable = isDiscountApplicable;
    }

    public double calculateTotal() {
        double tax = calculateTax();
        double discount = applyDiscount();
        return amount + tax - discount;
    }

    // Change to Inlined Method
    private double calculateTax() {
        return amount * taxRate;
    }

    // Change to Inlined Method
    private double applyDiscount() {
        if (isDiscountApplicable) {
            return amount * 0.05; // 5% de desconto
        }
        return 0;
    }

    public static void main(String[] args) {
        Invoice invoice = new Invoice(amount:1000.00, taxRate:0.2, isDiscountApplicable:true);
        System.out.println("Total Amount: " + invoice.calculateTotal());
    }
}
```

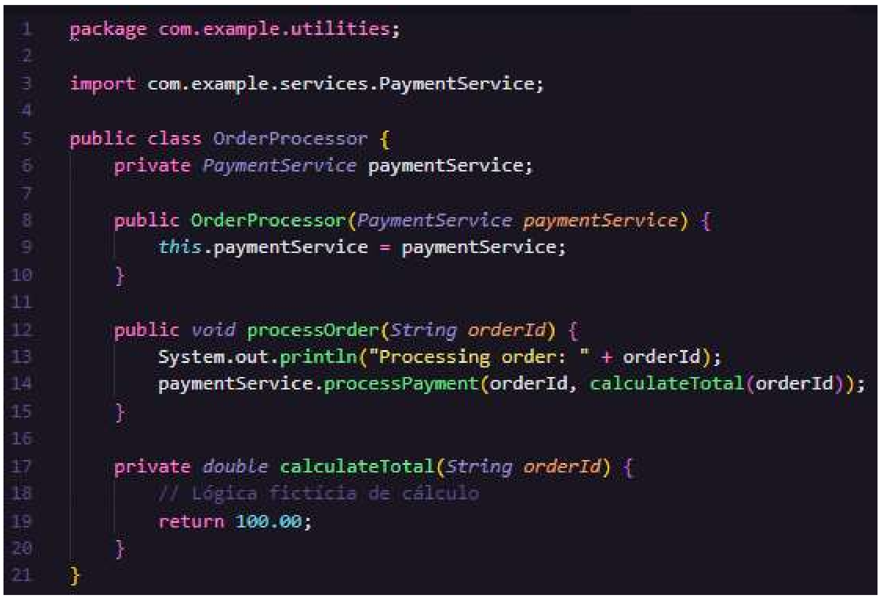
Marcar apenas uma oval por linha.

	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY ASPECT	○	○	○	○	○
SPEED ASPECT	○	○	○	○	○
RISK ASPECT	○	○	○	○	○

23. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

____

24. **6/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and * RISK aspects, how would you rate the level of difficulty for each aspect when applying the **RENAME PACKAGE** refactoring operation to the following source code?
If you prefer, you can see the code and explanation at this link.



```java
package com.example.utilities;

import com.example.services.PaymentService;

public class OrderProcessor {
    private PaymentService paymentService;

    public OrderProcessor(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void processOrder(String orderId) {
        System.out.println("Processing order: " + orderId);
        paymentService.processPayment(orderId, calculateTotal(orderId));
    }

    private double calculateTotal(String orderId) {
        // Lógica fictícia de cálculo
        return 100.00;
    }
}
```

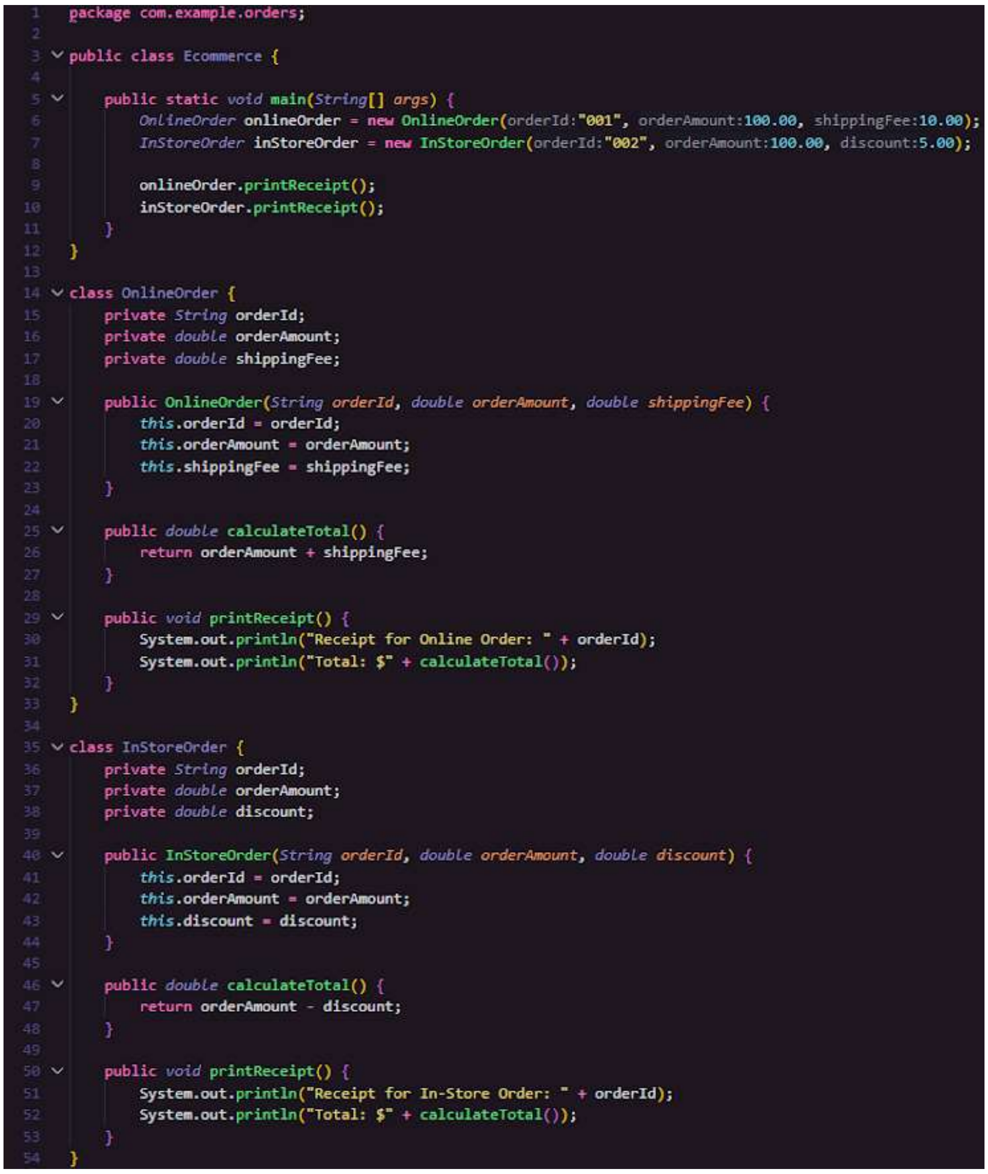
Marcar apenas uma oval por linha.

	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY ASPECT	◯	◯	◯	◯	◯
SPEED ASPECT	◯	◯	◯	◯	◯
RISK ASPECT	◯	◯	◯	◯	◯

25. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

26. **7/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK aspects, how would you rate the level of difficulty for each aspect when applying the **EXTRACT SUPERCLASS** refactoring operation to the following source code? *

If you prefer, you can see the code and explanation at this link.



```java
package com.example.orders;

public class Ecommerce {

    public static void main(String[] args) {
        OnlineOrder onlineOrder = new OnlineOrder(orderId:"001", orderAmount:100.00, shippingFee:10.00);
        InStoreOrder inStoreOrder = new InStoreOrder(orderId:"002", orderAmount:100.00, discount:5.00);

        onlineOrder.printReceipt();
        inStoreOrder.printReceipt();
    }
}

class OnlineOrder {
    private String orderId;
    private double orderAmount;
    private double shippingFee;

    public OnlineOrder(String orderId, double orderAmount, double shippingFee) {
        this.orderId = orderId;
        this.orderAmount = orderAmount;
        this.shippingFee = shippingFee;
    }

    public double calculateTotal() {
        return orderAmount + shippingFee;
    }

    public void printReceipt() {
        System.out.println("Receipt for Online Order: " + orderId);
        System.out.println("Total: $" + calculateTotal());
    }
}

class InStoreOrder {
    private String orderId;
    private double orderAmount;
    private double discount;

    public InStoreOrder(String orderId, double orderAmount, double discount) {
        this.orderId = orderId;
        this.orderAmount = orderAmount;
        this.discount = discount;
    }

    public double calculateTotal() {
        return orderAmount - discount;
    }

    public void printReceipt() {
        System.out.println("Receipt for In-Store Order: " + orderId);
        System.out.println("Total: $" + calculateTotal());
    }
}
```

Marcar apenas uma oval por linha.

	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY ASPECT	◯	◯	◯	◯	◯
SPEED					

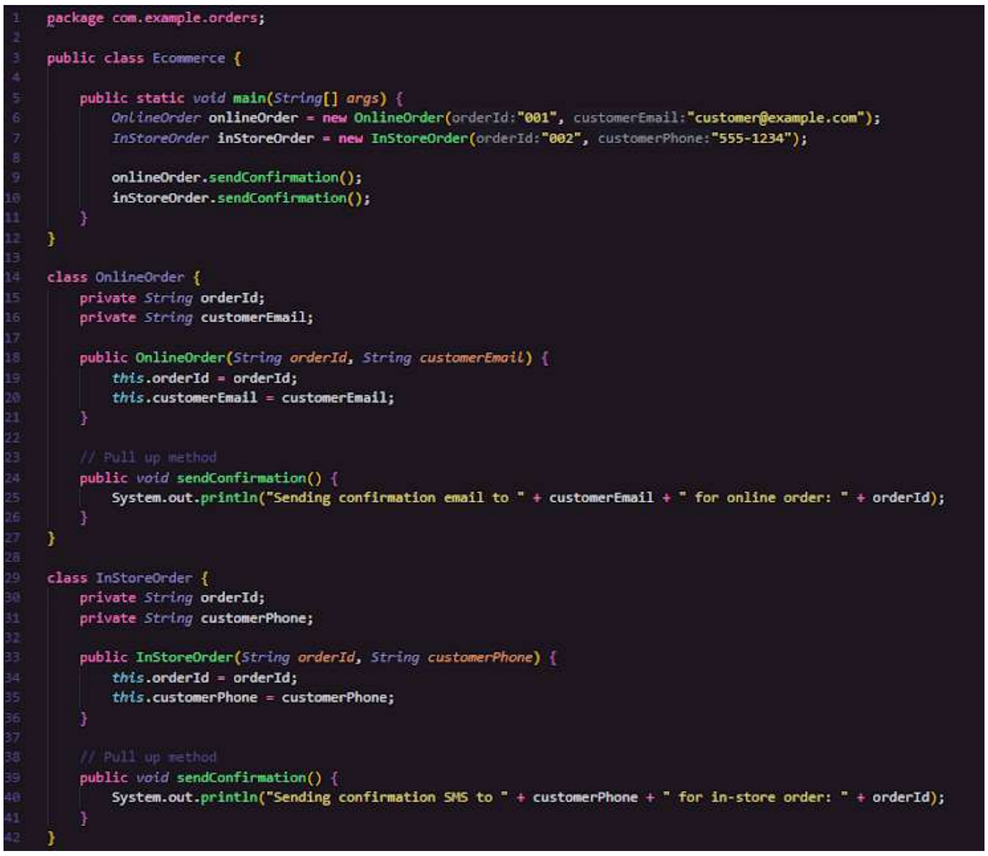
ASPECT					
SPEED ASPECT RISK ASPECT	○	○	○	○	○
RISK ASPECT	○	○	○	○	○

27. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

____

28. **8/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK aspects, how would you rate the level of difficulty for each aspect when applying the **PULL UP METHOD** refactoring operation to the following source code? *

If you prefer, you can see the code and explanation at this link.



```java
package com.example.orders;

public class Ecommerce {

    public static void main(String[] args) {
        OnlineOrder onlineOrder = new OnlineOrder(orderId:"001", customerEmail:"customer@example.com");
        InStoreOrder inStoreOrder = new InStoreOrder(orderId:"002", customerPhone:"555-1234");

        onlineOrder.sendConfirmation();
        inStoreOrder.sendConfirmation();
    }
}

class OnlineOrder {
    private String orderId;
    private String customerEmail;

    public OnlineOrder(String orderId, String customerEmail) {
        this.orderId = orderId;
        this.customerEmail = customerEmail;
    }

    // Pull up method
    public void sendConfirmation() {
        System.out.println("Sending confirmation email to " + customerEmail + " for online order: " + orderId);
    }
}

class InStoreOrder {
    private String orderId;
    private String customerPhone;

    public InStoreOrder(String orderId, String customerPhone) {
        this.orderId = orderId;
        this.customerPhone = customerPhone;
    }

    // Pull up method
    public void sendConfirmation() {
        System.out.println("Sending confirmation SMS to " + customerPhone + " for in-store order: " + orderId);
    }
}
```

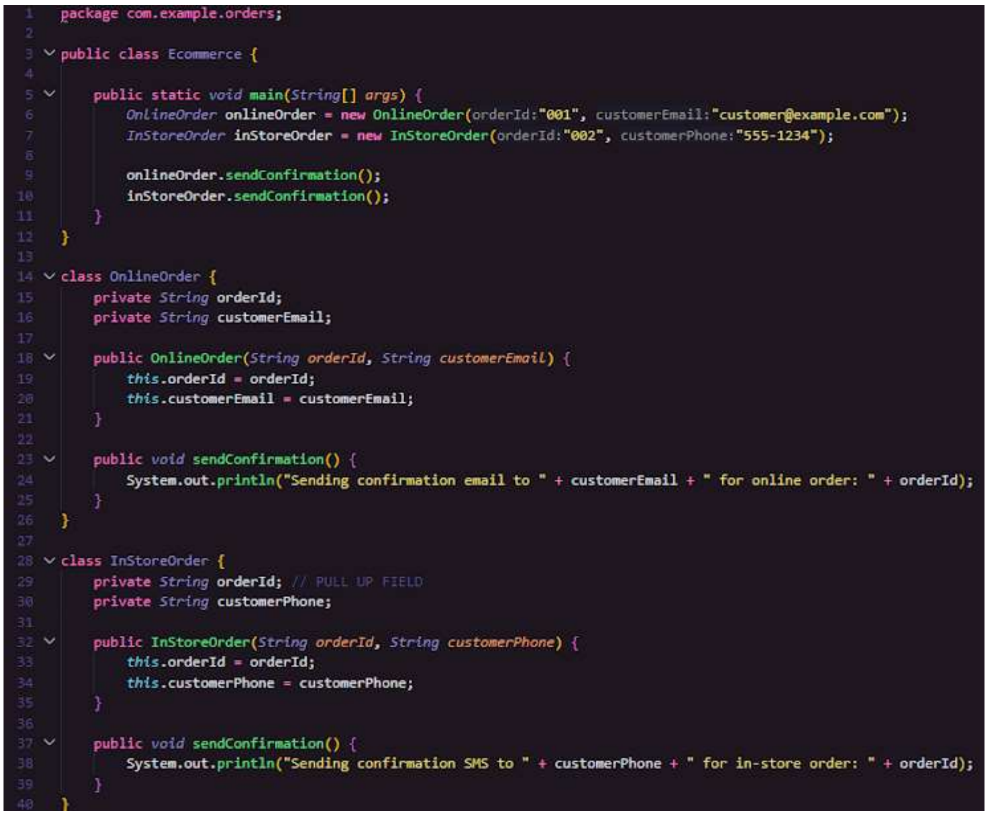
Marcar apenas uma oval por linha.

	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY ASPECT	◯	◯	◯	◯	◯
SPEED ASPECT	◯	◯	◯	◯	◯
RISK ASPECT	◯	◯	◯	◯	◯

29. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

30. **9/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK * aspects, how would you rate the level of difficulty for each aspect when applying the **PULL UP ATTRIBUTE** refactoring operation to the following source code?

If you prefer, you can see the code and explanation at this link.



```java
package com.example.orders;

public class Ecommerce {

    public static void main(String[] args) {
        OnlineOrder onlineOrder = new OnlineOrder(orderId:"001", customerEmail:"customer@example.com");
        InStoreOrder inStoreOrder = new InStoreOrder(orderId:"002", customerPhone:"555-1234");

        onlineOrder.sendConfirmation();
        inStoreOrder.sendConfirmation();
    }
}

class OnlineOrder {
    private String orderId;
    private String customerEmail;

    public OnlineOrder(String orderId, String customerEmail) {
        this.orderId = orderId;
        this.customerEmail = customerEmail;
    }

    public void sendConfirmation() {
        System.out.println("Sending confirmation email to " + customerEmail + " for online order: " + orderId);
    }
}

class InStoreOrder {
    private String orderId; // PULL UP FIELD
    private String customerPhone;

    public InStoreOrder(String orderId, String customerPhone) {
        this.orderId = orderId;
        this.customerPhone = customerPhone;
    }

    public void sendConfirmation() {
        System.out.println("Sending confirmation SMS to " + customerPhone + " for in-store order: " + orderId);
    }
}
```

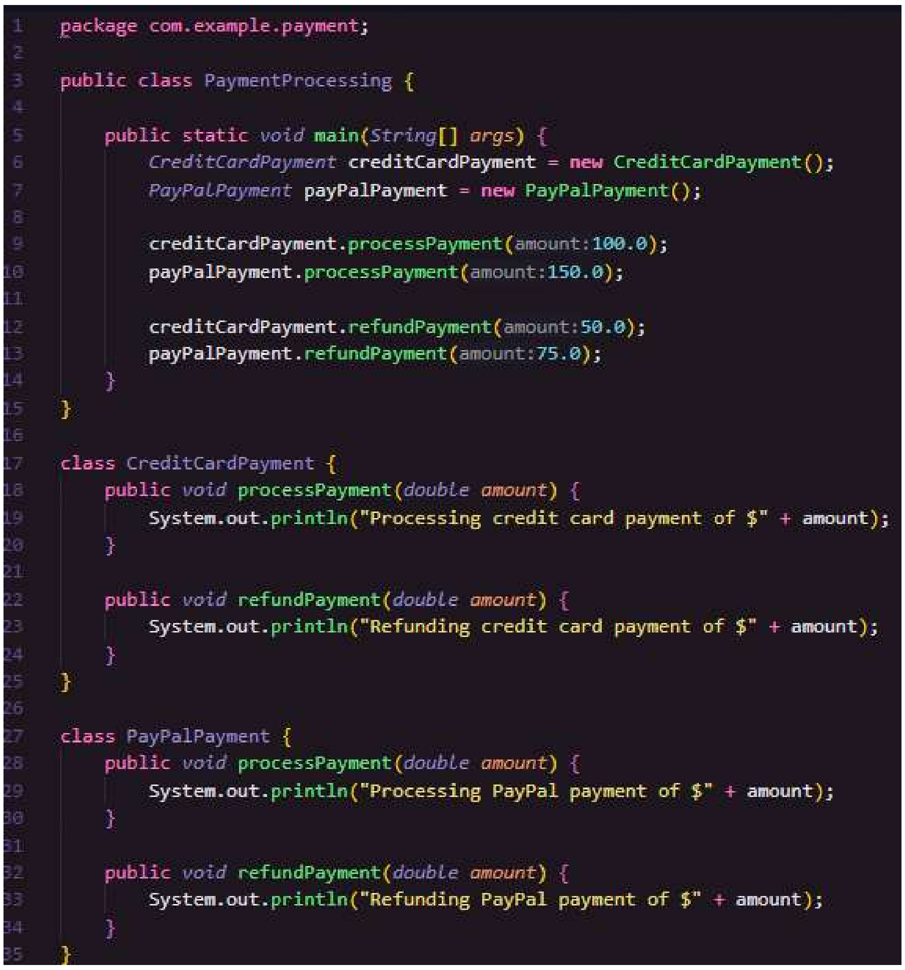
Marcar apenas uma oval por linha.

	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY ASPECT					
SPEED ASPECT					
RISK ASPECT					

31. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

32. **10/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK aspects, how would you rate the level of difficulty for each aspect when applying the **EXTRACT INTERFACE** refactoring operation to the following source code? *

If you prefer, you can see the code and explanation at this link.



```java
package com.example.payment;

public class PaymentProcessing {

    public static void main(String[] args) {
        CreditCardPayment creditCardPayment = new CreditCardPayment();
        PayPalPayment payPalPayment = new PayPalPayment();

        creditCardPayment.processPayment(amount:100.0);
        payPalPayment.processPayment(amount:150.0);

        creditCardPayment.refundPayment(amount:50.0);
        payPalPayment.refundPayment(amount:75.0);
    }
}

class CreditCardPayment {
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment of $" + amount);
    }

    public void refundPayment(double amount) {
        System.out.println("Refunding credit card payment of $" + amount);
    }
}

class PayPalPayment {
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }

    public void refundPayment(double amount) {
        System.out.println("Refunding PayPal payment of $" + amount);
    }
}
```

Marcar apenas uma oval por linha.

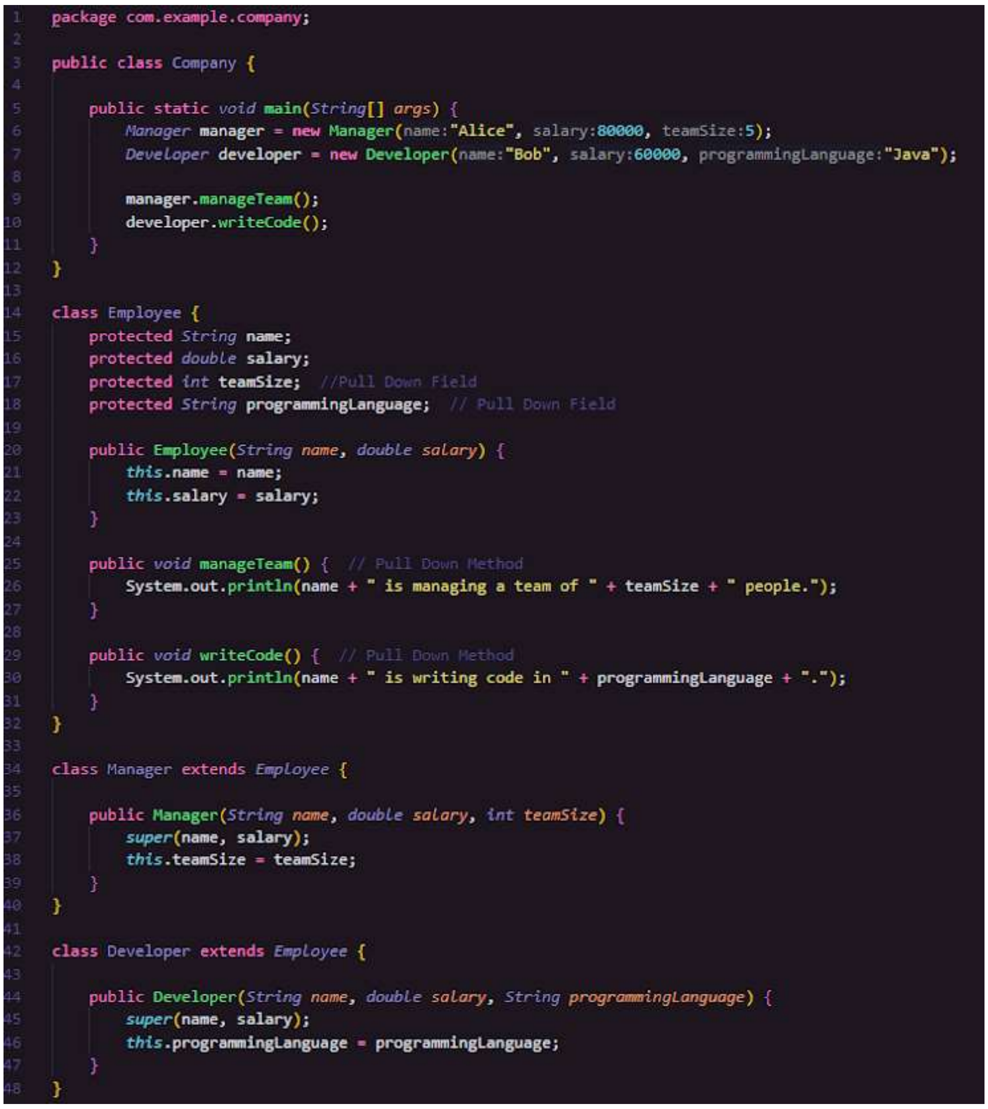
	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY ASPECT	⬭	⬭	⬭	⬭	⬭
SPEED ASPECT	⬭	⬭	⬭	⬭	⬭
RISK ASPECT	⬭	⬭	⬭	⬭	⬭

33. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

____

34. **11/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK aspects, how would you rate the level of difficulty for each aspect when applying the **PUSH DOWN ATTRIBUTE** refactoring operation to the following source code? *

If you prefer, you can see the code and explanation at this link.



```java
package com.example.company;

public class Company {

    public static void main(String[] args) {
        Manager manager = new Manager(name:"Alice", salary:80000, teamSize:5);
        Developer developer = new Developer(name:"Bob", salary:60000, programmingLanguage:"Java");

        manager.manageTeam();
        developer.writeCode();
    }
}

class Employee {
    protected String name;
    protected double salary;
    protected int teamSize;  //Pull Down Field
    protected String programmingLanguage;  // Pull Down Field

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    public void manageTeam() {  // Pull Down Method
        System.out.println(name + " is managing a team of " + teamSize + " people.");
    }

    public void writeCode() {  // Pull Down Method
        System.out.println(name + " is writing code in " + programmingLanguage + ".");
    }
}

class Manager extends Employee {

    public Manager(String name, double salary, int teamSize) {
        super(name, salary);
        this.teamSize = teamSize;
    }
}

class Developer extends Employee {

    public Developer(String name, double salary, String programmingLanguage) {
        super(name, salary);
        this.programmingLanguage = programmingLanguage;
    }
}
```

Marcar apenas uma oval por linha.

	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY ASPECT	◯	◯	◯	◯	◯
SPEED ASPECT	◯	◯	◯	◯	◯

RISK ASPECT

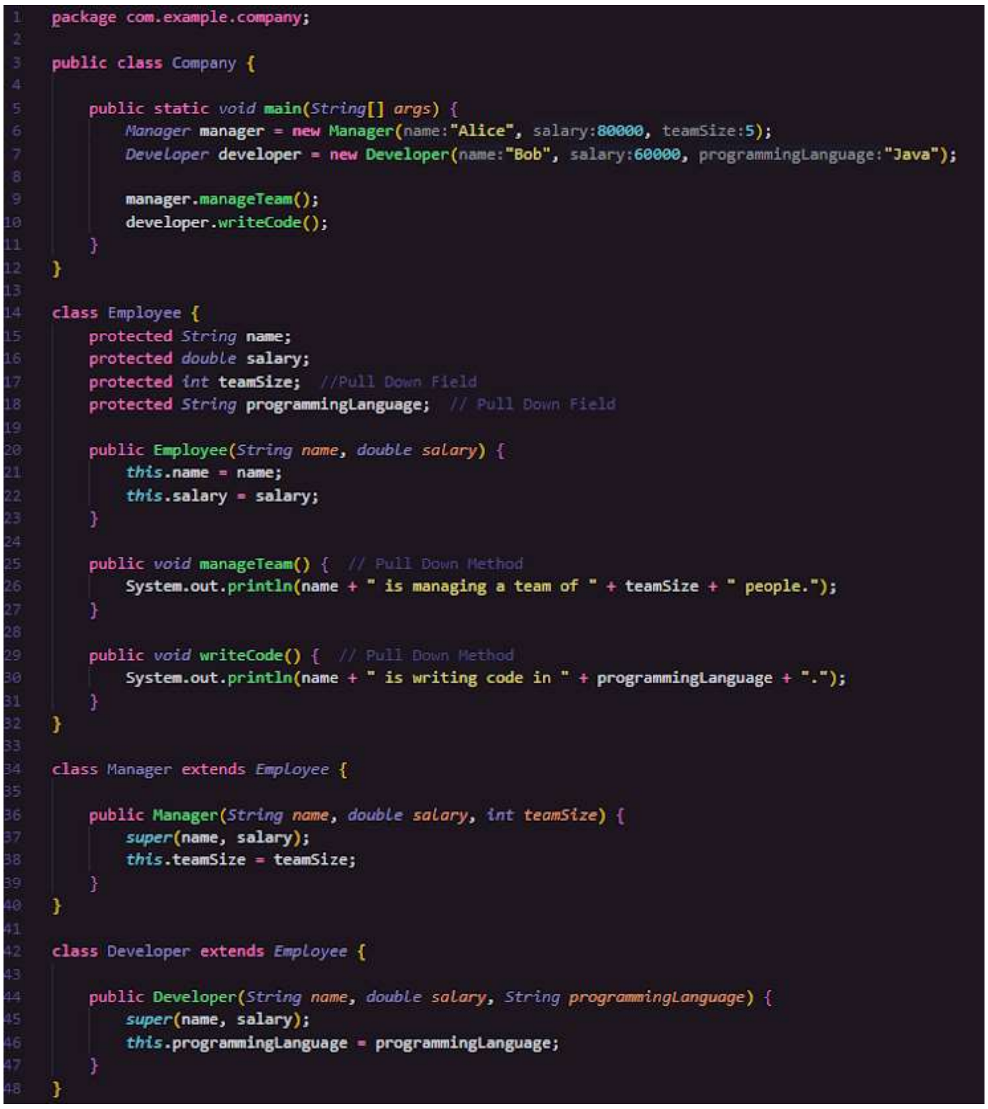
RISK ASPECT	○	○	○	○	○

35. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

____

36. **12/12 -** Based on your experience and the definitions of COMPLEXITY, SPEED, and RISK aspects, how would you rate the level of difficulty for each aspect when applying the **PUSH DOWN METHOD** refactoring operation to the following source code? *

If you prefer, you can see the code and explanation at this link.



```java
package com.example.company;

public class Company {

    public static void main(String[] args) {
        Manager manager = new Manager(name:"Alice", salary:80000, teamSize:5);
        Developer developer = new Developer(name:"Bob", salary:60000, programmingLanguage:"Java");

        manager.manageTeam();
        developer.writeCode();
    }
}

class Employee {
    protected String name;
    protected double salary;
    protected int teamSize;  //Pull Down Field
    protected String programmingLanguage;  // Pull Down Field

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    public void manageTeam() {  // Pull Down Method
        System.out.println(name + " is managing a team of " + teamSize + " people.");
    }

    public void writeCode() {  // Pull Down Method
        System.out.println(name + " is writing code in " + programmingLanguage + ".");
    }
}

class Manager extends Employee {

    public Manager(String name, double salary, int teamSize) {
        super(name, salary);
        this.teamSize = teamSize;
    }
}

class Developer extends Employee {

    public Developer(String name, double salary, String programmingLanguage) {
        super(name, salary);
        this.programmingLanguage = programmingLanguage;
    }
}
```

Marcar apenas uma oval por linha.

	Very Difficult	Difficult	Neutral	Easy	Very Easy
COMPLEXITY ASPECT	◯	◯	◯	◯	◯
SPEED ASPECT	◯	◯	◯	◯	◯

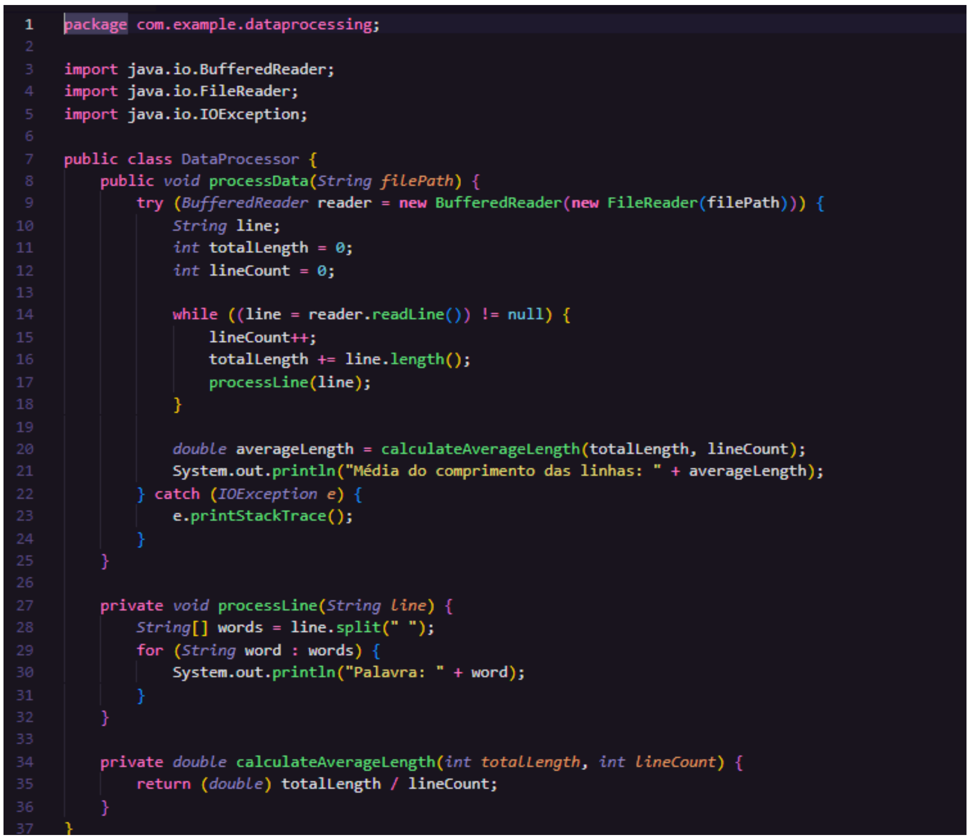
RISK ASPECT	○	○	○	○	○

RISK ASPECT

37. **Other Informations**
Would you like to report any additional information in terms of COMPLEXITY and/or SPEED and/or RISK about the above refactoring operation?

Survey: Refactoring's Triviality
Applied Refactorings

1/12 Extract Method

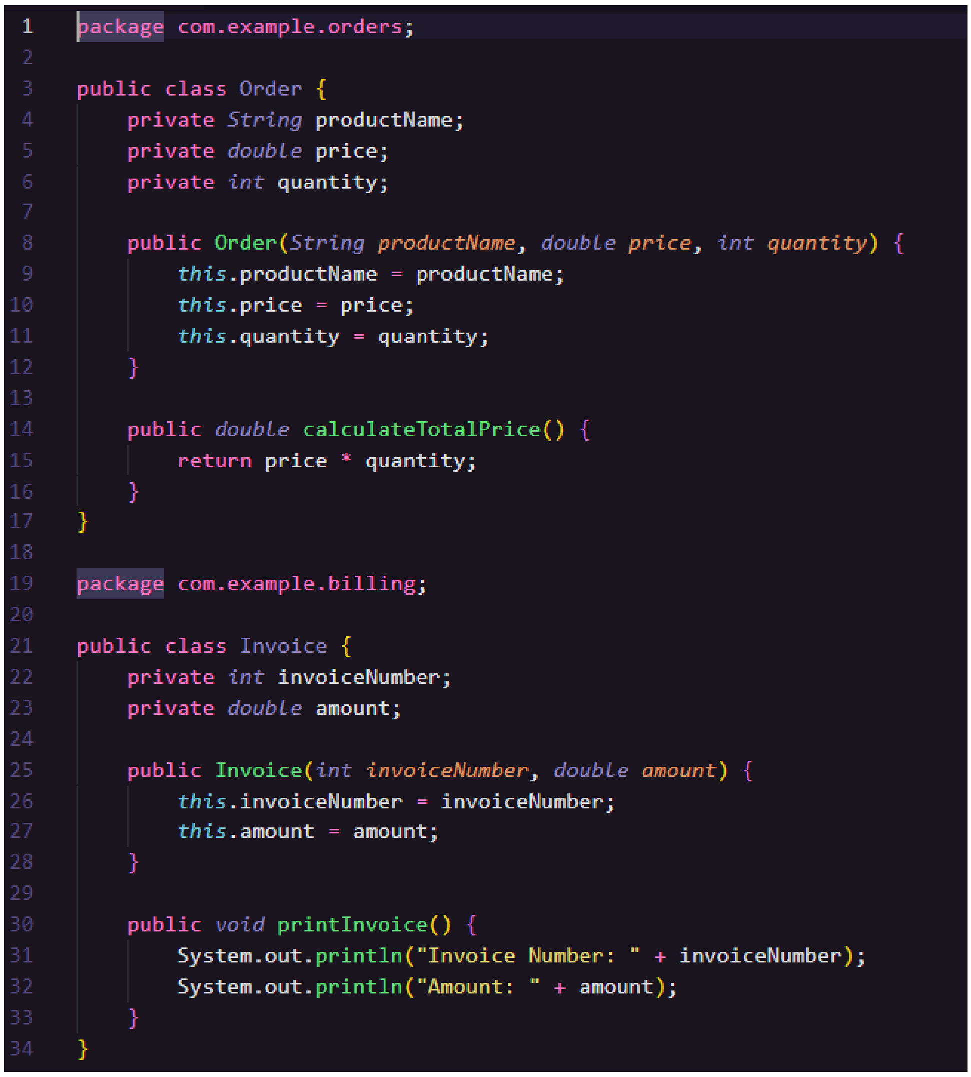


```java
package com.example.dataprocessing;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class DataProcessor {
    public void processData(String filePath) {
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            int totalLength = 0;
            int lineCount = 0;

            while ((line = reader.readLine()) != null) {
                lineCount++;
                totalLength += line.length();
                processLine(line);
            }

            double averageLength = calculateAverageLength(totalLength, lineCount);
            System.out.println("Média do comprimento das linhas: " + averageLength);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void processLine(String line) {
        String[] words = line.split(" ");
        for (String word : words) {
            System.out.println("Palavra: " + word);
        }
    }

    private double calculateAverageLength(int totalLength, int lineCount) {
        return (double) totalLength / lineCount;
    }
}
```

Explanation:
Before refactoring, the processData method does everything: opens the file, reads the lines, processes each line, and calculates the average length of the lines.

After refactoring, we extract two helper methods:
-processLine(String line): To process each line (splitting it into words and printing each word).
-calculateAverageLength(int totalLength, int lineCount): To calculate the average length of the lines.

2/12 Move Class

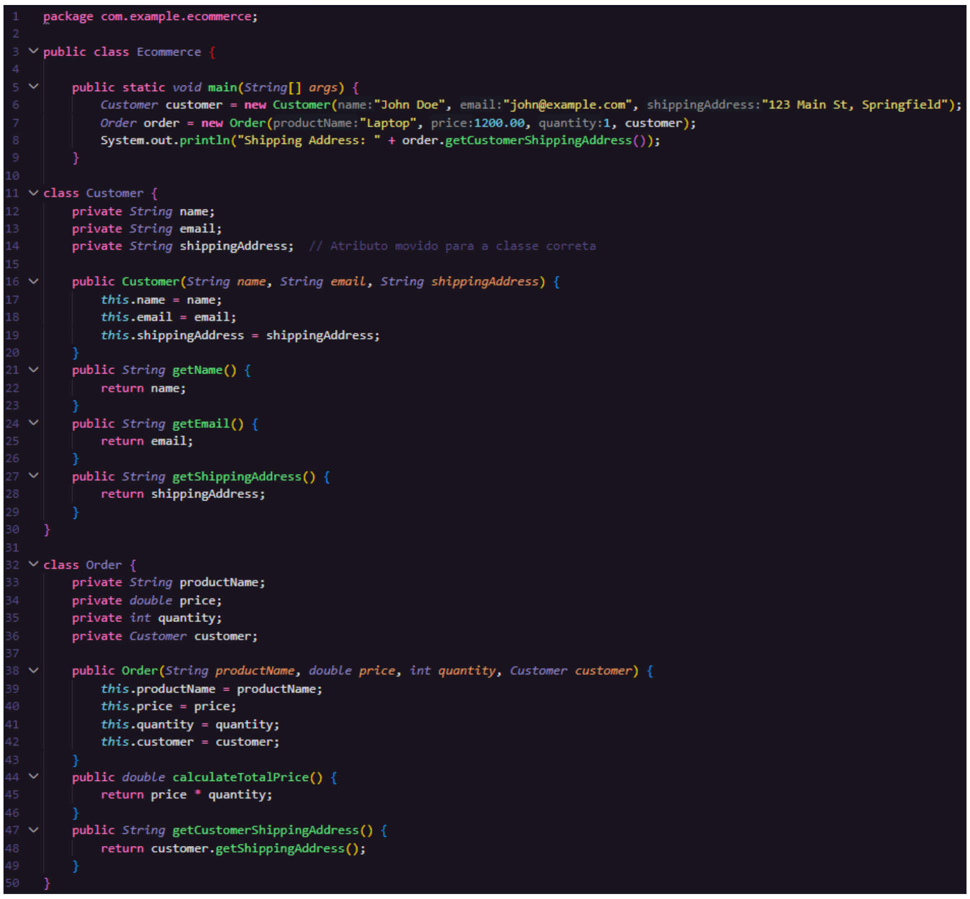


```java
package com.example.orders;

public class Order {
    private String productName;
    private double price;
    private int quantity;

    public Order(String productName, double price, int quantity) {
        this.productName = productName;
        this.price = price;
        this.quantity = quantity;
    }

    public double calculateTotalPrice() {
        return price * quantity;
    }
}

package com.example.billing;

public class Invoice {
    private int invoiceNumber;
    private double amount;

    public Invoice(int invoiceNumber, double amount) {
        this.invoiceNumber = invoiceNumber;
        this.amount = amount;
    }

    public void printInvoice() {
        System.out.println("Invoice Number: " + invoiceNumber);
        System.out.println("Amount: " + amount);
    }
}
```

Explanation:
Before Refactoring: The Invoice class was located in the orders package, which was not a relevant location for the functionality it represented.

After Refactoring: The Invoice class was moved to a new package called billing, which is better suited to its purpose of handling functionality related to invoices and finance.

3/12 Move Attribute

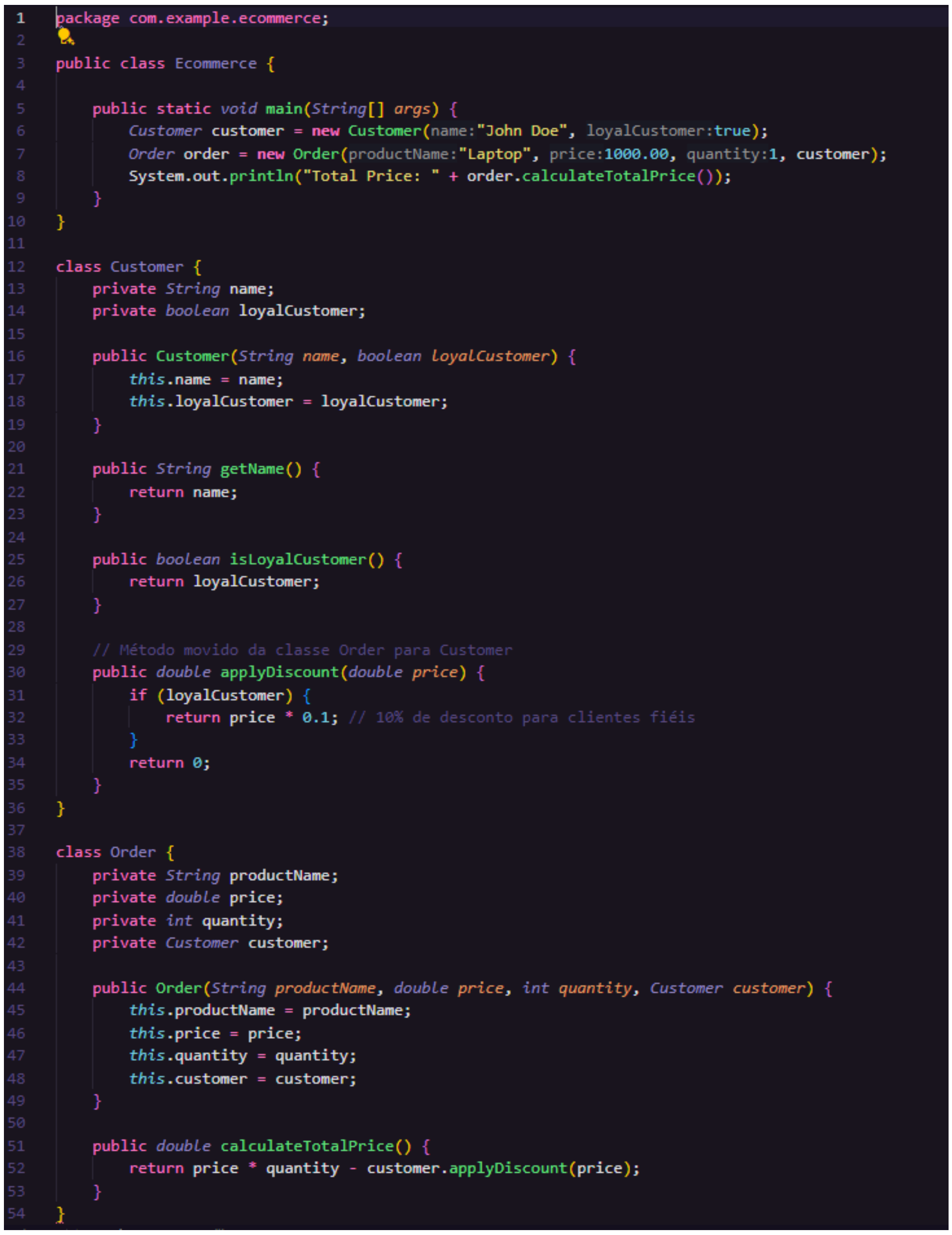


```java
package com.example.ecommerce;

public class Ecommerce {

    public static void main(String[] args) {
        Customer customer = new Customer(name:"John Doe", email:"john@example.com", shippingAddress:"123 Main St, Springfield");
        Order order = new Order(productName:"Laptop", price:1200.00, quantity:1, customer);
        System.out.println("Shipping Address: " + order.getCustomerShippingAddress());
    }

class Customer {
    private String name;
    private String email;
    private String shippingAddress;  // Atributo movido para a classe correta

    public Customer(String name, String email, String shippingAddress) {
        this.name = name;
        this.email = email;
        this.shippingAddress = shippingAddress;
    }
    public String getName() {
        return name;
    }
    public String getEmail() {
        return email;
    }
    public String getShippingAddress() {
        return shippingAddress;
    }
}

class Order {
    private String productName;
    private double price;
    private int quantity;
    private Customer customer;

    public Order(String productName, double price, int quantity, Customer customer) {
        this.productName = productName;
        this.price = price;
        this.quantity = quantity;
        this.customer = customer;
    }
    public double calculateTotalPrice() {
        return price * quantity;
    }
    public String getCustomerShippingAddress() {
        return customer.getShippingAddress();
    }
}
```

Explanation:
Before Refactoring: The shippingAddress attribute was inside the Order class, which was not suitable because the shipping address is a characteristic of the customer.

After Refactoring: The shippingAddress attribute was moved to the Customer class, where it makes more sense. The Order class now holds a reference to the Customer object and can access the shipping address through that reference.

4/12 Move Method

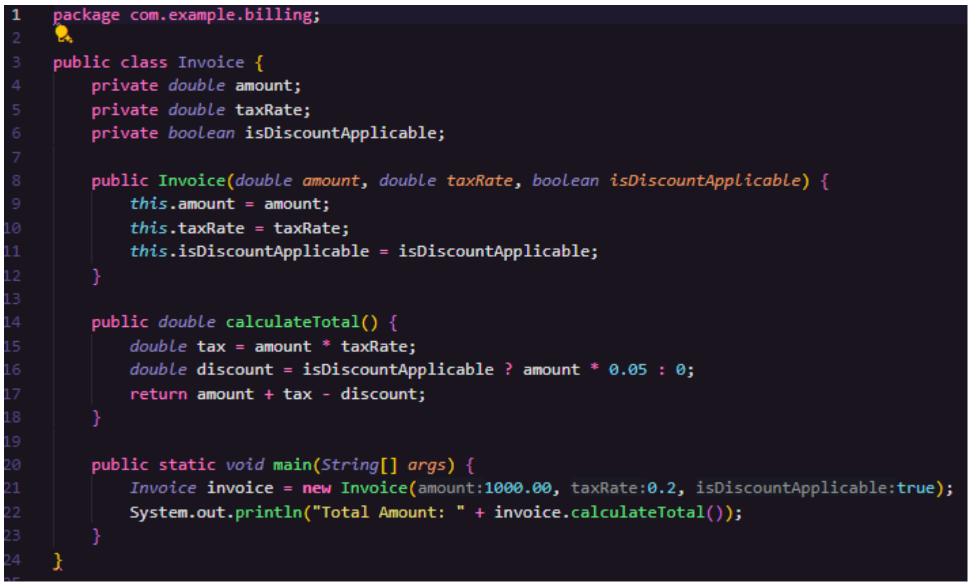


```java
package com.example.ecommerce;

public class Ecommerce {

    public static void main(String[] args) {
        Customer customer = new Customer(name:"John Doe", loyalCustomer:true);
        Order order = new Order(productName:"Laptop", price:1000.00, quantity:1, customer);
        System.out.println("Total Price: " + order.calculateTotalPrice());
    }
}

class Customer {
    private String name;
    private boolean loyalCustomer;

    public Customer(String name, boolean loyalCustomer) {
        this.name = name;
        this.loyalCustomer = loyalCustomer;
    }

    public String getName() {
        return name;
    }

    public boolean isLoyalCustomer() {
        return loyalCustomer;
    }

    // Método movido da classe Order para Customer
    public double applyDiscount(double price) {
        if (loyalCustomer) {
            return price * 0.1; // 10% de desconto para clientes fiéis
        }
        return 0;
    }
}

class Order {
    private String productName;
    private double price;
    private int quantity;
    private Customer customer;

    public Order(String productName, double price, int quantity, Customer customer) {
        this.productName = productName;
        this.price = price;
        this.quantity = quantity;
        this.customer = customer;
    }

    public double calculateTotalPrice() {
        return price * quantity - customer.applyDiscount(price);
    }
}
```

Explanation:

Before Refactoring: The applyDiscount method was in the Order class, but it depended on the attributes of the Customer class, which meant that it was not in the most appropriate class. After Refactoring: The applyDiscount method was moved to the Customer class, which is the most appropriate class, as the discount depends on the customer's characteristics. Now, the Order class calls the applyDiscount method of the Customer class.

5/12 Inline Method

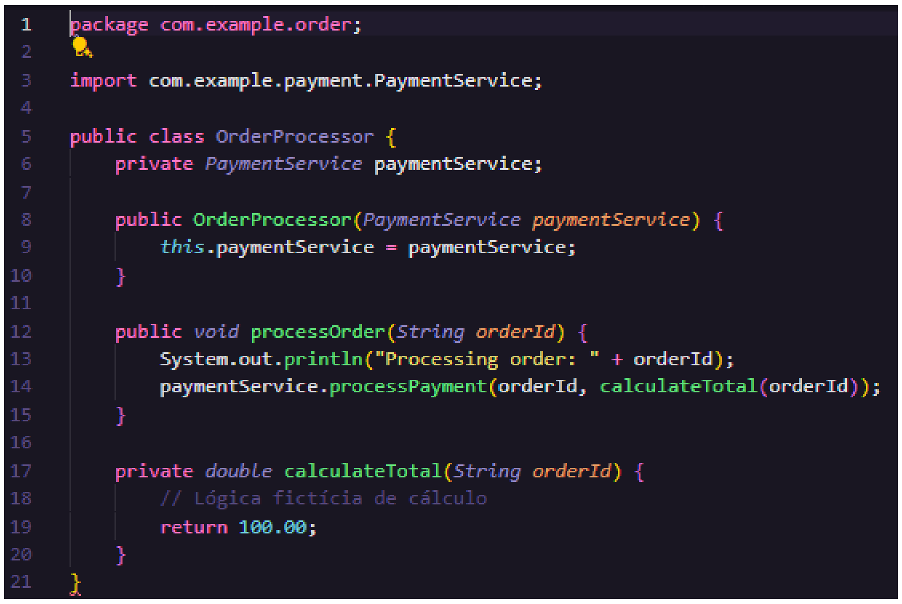


```java
package com.example.billing;

public class Invoice {
    private double amount;
    private double taxRate;
    private boolean isDiscountApplicable;

    public Invoice(double amount, double taxRate, boolean isDiscountApplicable) {
        this.amount = amount;
        this.taxRate = taxRate;
        this.isDiscountApplicable = isDiscountApplicable;
    }

    public double calculateTotal() {
        double tax = amount * taxRate;
        double discount = isDiscountApplicable ? amount * 0.05 : 0;
        return amount + tax - discount;
    }

    public static void main(String[] args) {
        Invoice invoice = new Invoice(amount:1000.00, taxRate:0.2, isDiscountApplicable:true);
        System.out.println("Total Amount: " + invoice.calculateTotal());
    }
}
```

Explanation:
Before Refactoring: The Invoice class had two separate methods, calculateTax and applyDiscount, to calculate tax and apply discount, respectively. These methods were invoked within calculateTotal.

After Refactoring: The calculateTax and applyDiscount methods were inlined, that is, their contents were moved directly to the calculateTotal method. Now, calculateTotal performs the tax and discount calculation directly, without having to call additional methods.

6/12 Rename Package

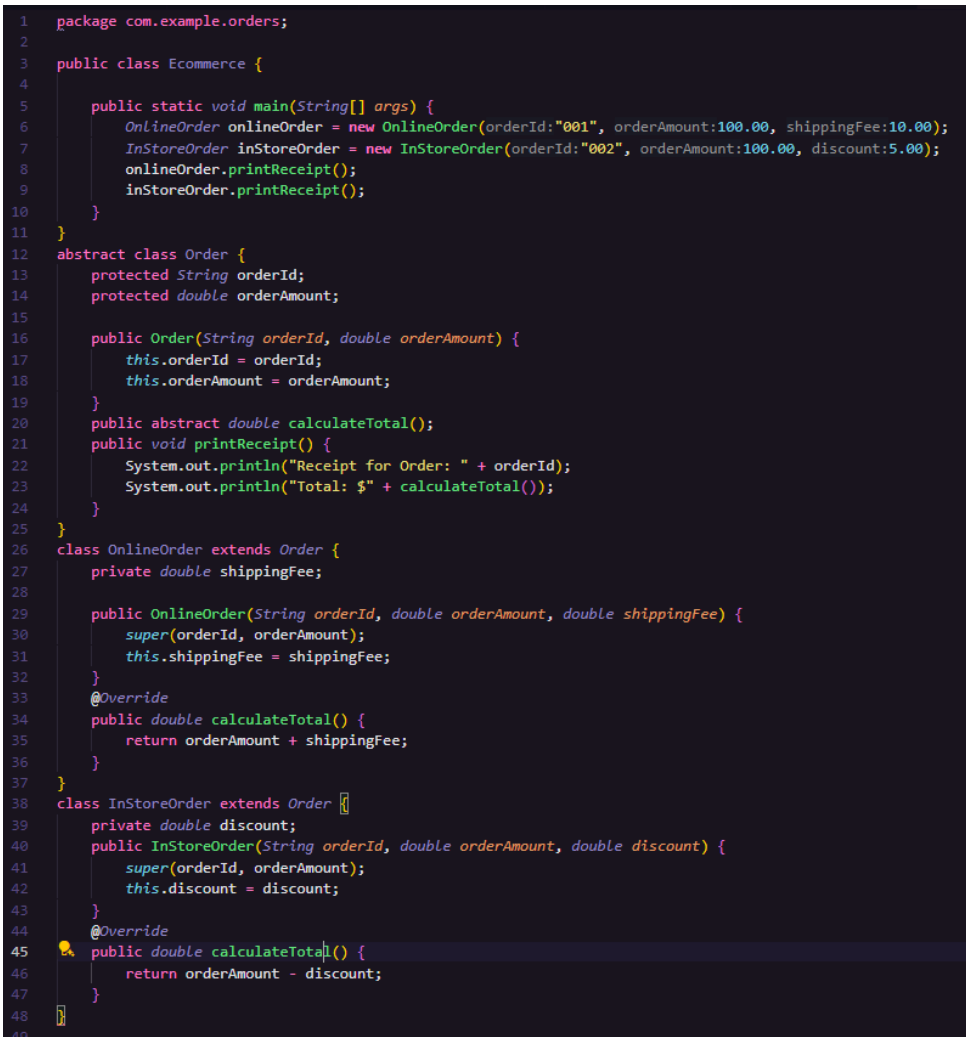


```java
package com.example.order;

import com.example.payment.PaymentService;

public class OrderProcessor {
    private PaymentService paymentService;

    public OrderProcessor(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void processOrder(String orderId) {
        System.out.println("Processing order: " + orderId);
        paymentService.processPayment(orderId, calculateTotal(orderId));
    }

    private double calculateTotal(String orderId) {
        // Lógica fictícia de cálculo
        return 100.00;
    }
}
```

Explanation:
Before Refactoring: The OrderProcessor class was located in the com.example.utilities package, which is very generic and does not adequately reflect the responsibility of this class, which is to process orders.

After Refactoring: OrderProcessor was moved to com.example. order, which more clearly reflects its order processing function.

7/12 Extract Superclass

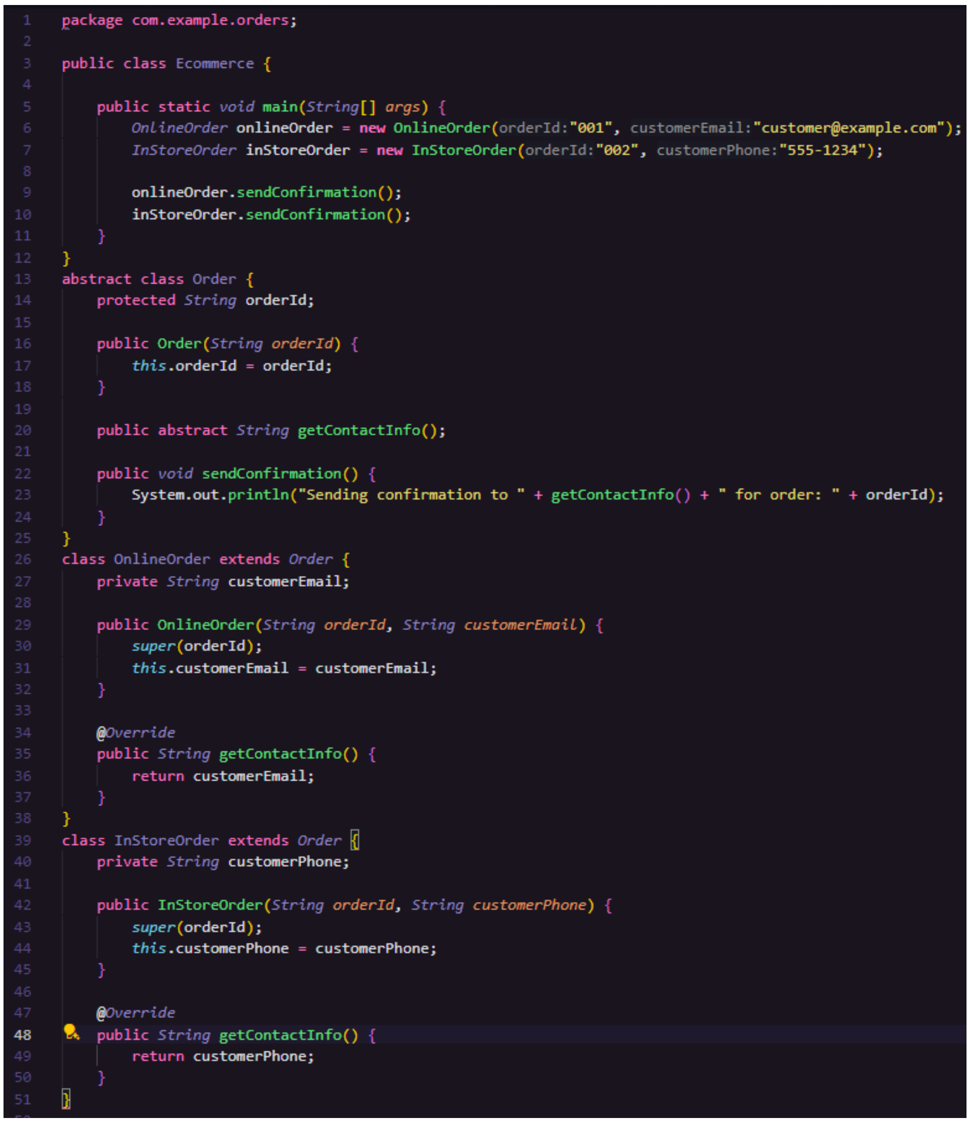


```java
package com.example.orders;

public class Ecommerce {

    public static void main(String[] args) {
        OnlineOrder onlineOrder = new OnlineOrder(orderId:"001", orderAmount:100.00, shippingFee:10.00);
        InStoreOrder inStoreOrder = new InStoreOrder(orderId:"002", orderAmount:100.00, discount:5.00);
        onlineOrder.printReceipt();
        inStoreOrder.printReceipt();
    }
}
abstract class Order {
    protected String orderId;
    protected double orderAmount;

    public Order(String orderId, double orderAmount) {
        this.orderId = orderId;
        this.orderAmount = orderAmount;
    }
    public abstract double calculateTotal();
    public void printReceipt() {
        System.out.println("Receipt for Order: " + orderId);
        System.out.println("Total: $" + calculateTotal());
    }
}
class OnlineOrder extends Order {
    private double shippingFee;

    public OnlineOrder(String orderId, double orderAmount, double shippingFee) {
        super(orderId, orderAmount);
        this.shippingFee = shippingFee;
    }
    @Override
    public double calculateTotal() {
        return orderAmount + shippingFee;
    }
}
class InStoreOrder extends Order {
    private double discount;
    public InStoreOrder(String orderId, double orderAmount, double discount) {
        super(orderId, orderAmount);
        this.discount = discount;
    }
    @Override
    public double calculateTotal() {
        return orderAmount - discount;
    }
}
```

Explanation:
Before Refactoring: The OnlineOrder and InStoreOrder classes had duplicate methods and attributes, such as orderId, orderAmount, calculateTotal(), and printReceipt(). This resulted in redundant code and made maintenance difficult.

After Refactoring: We created an abstract superclass called Order, which contains the common receipt printing logic (printReceipt()) and the orderId and orderAmount attributes. The calculateTotal() methods were defined as abstract in the superclass and implemented in the OnlineOrder and InStoreOrder subclasses, which now inherit from Order.

8/12 Pull Up Method

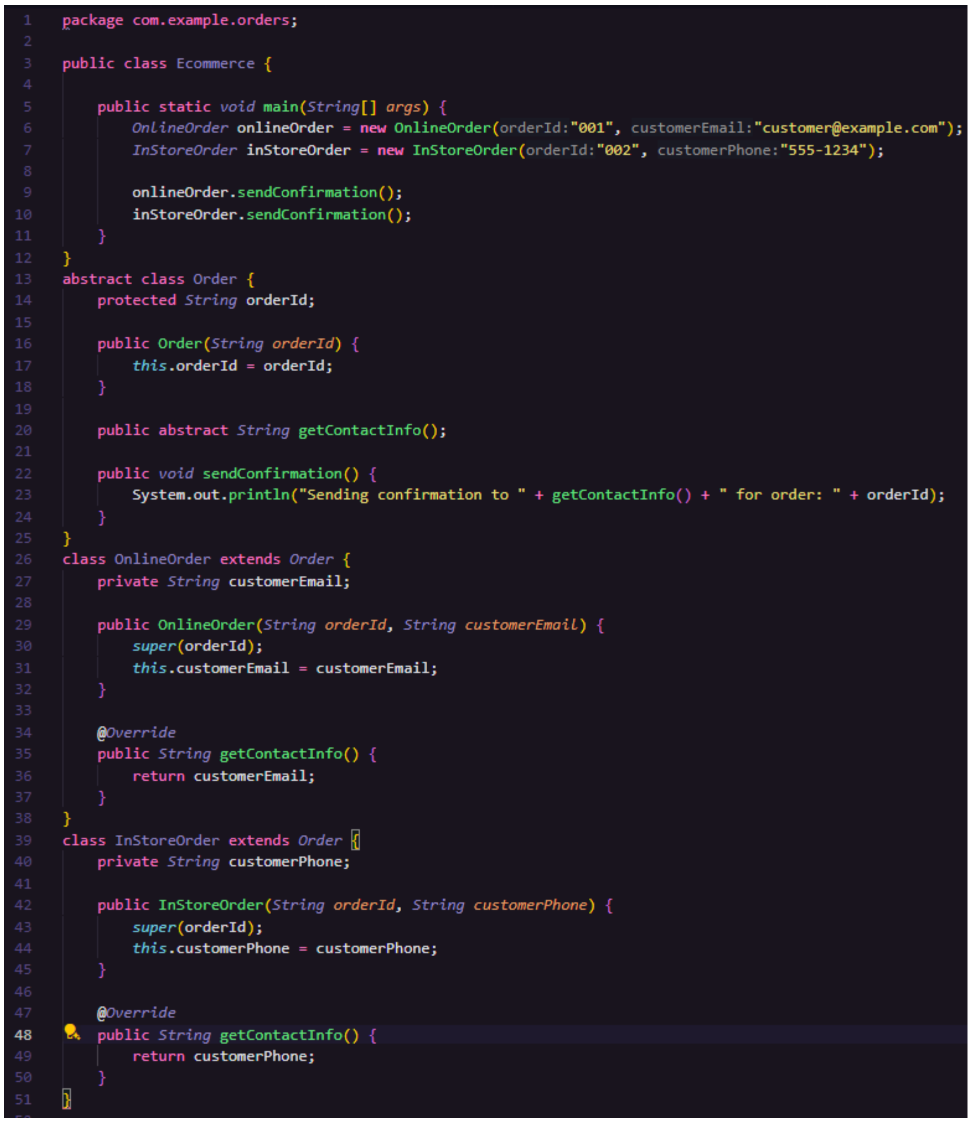


```java
package com.example.orders;

public class Ecommerce {

    public static void main(String[] args) {
        OnlineOrder onlineOrder = new OnlineOrder(orderId:"001", customerEmail:"customer@example.com");
        InStoreOrder inStoreOrder = new InStoreOrder(orderId:"002", customerPhone:"555-1234");

        onlineOrder.sendConfirmation();
        inStoreOrder.sendConfirmation();
    }
}
abstract class Order {
    protected String orderId;

    public Order(String orderId) {
        this.orderId = orderId;
    }

    public abstract String getContactInfo();

    public void sendConfirmation() {
        System.out.println("Sending confirmation to " + getContactInfo() + " for order: " + orderId);
    }
}
class OnlineOrder extends Order {
    private String customerEmail;

    public OnlineOrder(String orderId, String customerEmail) {
        super(orderId);
        this.customerEmail = customerEmail;
    }

    @Override
    public String getContactInfo() {
        return customerEmail;
    }
}
class InStoreOrder extends Order {
    private String customerPhone;

    public InStoreOrder(String orderId, String customerPhone) {
        super(orderId);
        this.customerPhone = customerPhone;
    }

    @Override
    public String getContactInfo() {
        return customerPhone;
    }
}
```

Explanation:
Before Refactoring: The OnlineOrder and InStoreOrder classes had the sendConfirmation() method, which was identical in both, resulting in duplicate code.

After Refactoring: The sendConfirmation() method was moved to the Order superclass. The abstract method getContactInfo() was created in the Order superclass, and implemented in subclasses to provide specific contact information (customerEmail or customerPhone).

9/12 Pull Up Attribute

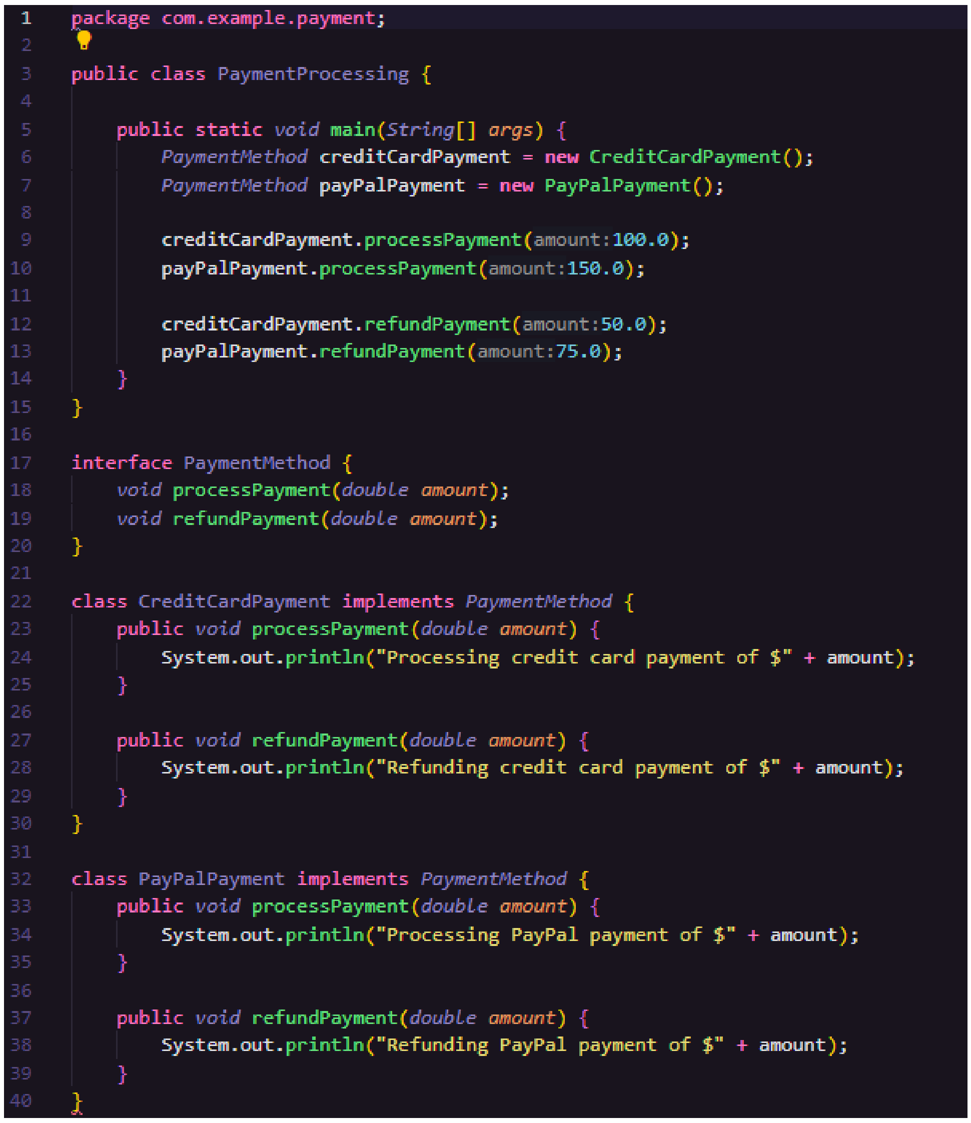


```java
package com.example.orders;

public class Ecommerce {

    public static void main(String[] args) {
        OnlineOrder onlineOrder = new OnlineOrder(orderId:"001", customerEmail:"customer@example.com");
        InStoreOrder inStoreOrder = new InStoreOrder(orderId:"002", customerPhone:"555-1234");

        onlineOrder.sendConfirmation();
        inStoreOrder.sendConfirmation();
    }
}
abstract class Order {
    protected String orderId;

    public Order(String orderId) {
        this.orderId = orderId;
    }

    public abstract String getContactInfo();

    public void sendConfirmation() {
        System.out.println("Sending confirmation to " + getContactInfo() + " for order: " + orderId);
    }
}
class OnlineOrder extends Order {
    private String customerEmail;

    public OnlineOrder(String orderId, String customerEmail) {
        super(orderId);
        this.customerEmail = customerEmail;
    }

    @Override
    public String getContactInfo() {
        return customerEmail;
    }
}
class InStoreOrder extends Order {
    private String customerPhone;

    public InStoreOrder(String orderId, String customerPhone) {
        super(orderId);
        this.customerPhone = customerPhone;
    }

    @Override
    public String getContactInfo() {
        return customerPhone;
    }
}
```

Explanation:
Before Refactoring: The OnlineOrder and InStoreOrder classes had the orderId attribute duplicated. Both classes had the logic to store and access the order ID independently.

After Refactoring: The orderId attribute was moved to the Order superclass, eliminating duplication. Now, both OnlineOrder and InStoreOrder inherit this attribute and the getOrderId() method from the superclass.

10/12 Extract Interface



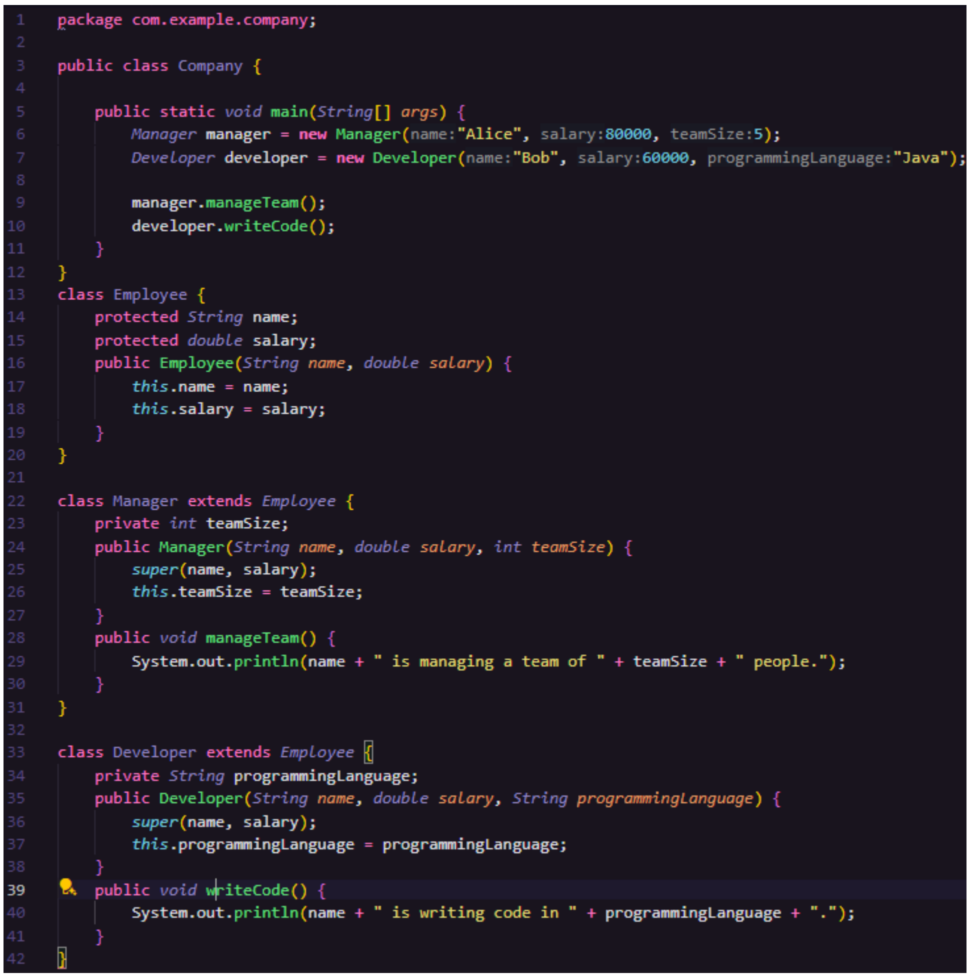
```java
package com.example.payment;

public class PaymentProcessing {

    public static void main(String[] args) {
        PaymentMethod creditCardPayment = new CreditCardPayment();
        PaymentMethod payPalPayment = new PayPalPayment();

        creditCardPayment.processPayment(amount:100.0);
        payPalPayment.processPayment(amount:150.0);

        creditCardPayment.refundPayment(amount:50.0);
        payPalPayment.refundPayment(amount:75.0);
    }
}

interface PaymentMethod {
    void processPayment(double amount);
    void refundPayment(double amount);
}

class CreditCardPayment implements PaymentMethod {
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment of $" + amount);
    }

    public void refundPayment(double amount) {
        System.out.println("Refunding credit card payment of $" + amount);
    }
}

class PayPalPayment implements PaymentMethod {
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }

    public void refundPayment(double amount) {
        System.out.println("Refunding PayPal payment of $" + amount);
    }
}
```

Explanation:

Before Refactoring: The CreditCardPayment and PayPalPayment classes had processPayment and refundPayment methods, which were similar but not formally related. This meant that the classes had common functionality, but there was no standardized way to treat them as interchangeable types.

After Refactoring: The common methods processPayment and refundPayment were extracted into the PaymentMethod interface.Now both CreditCardPayment and PayPalPayment implement this interface, allowing both classes to be treated uniformly.

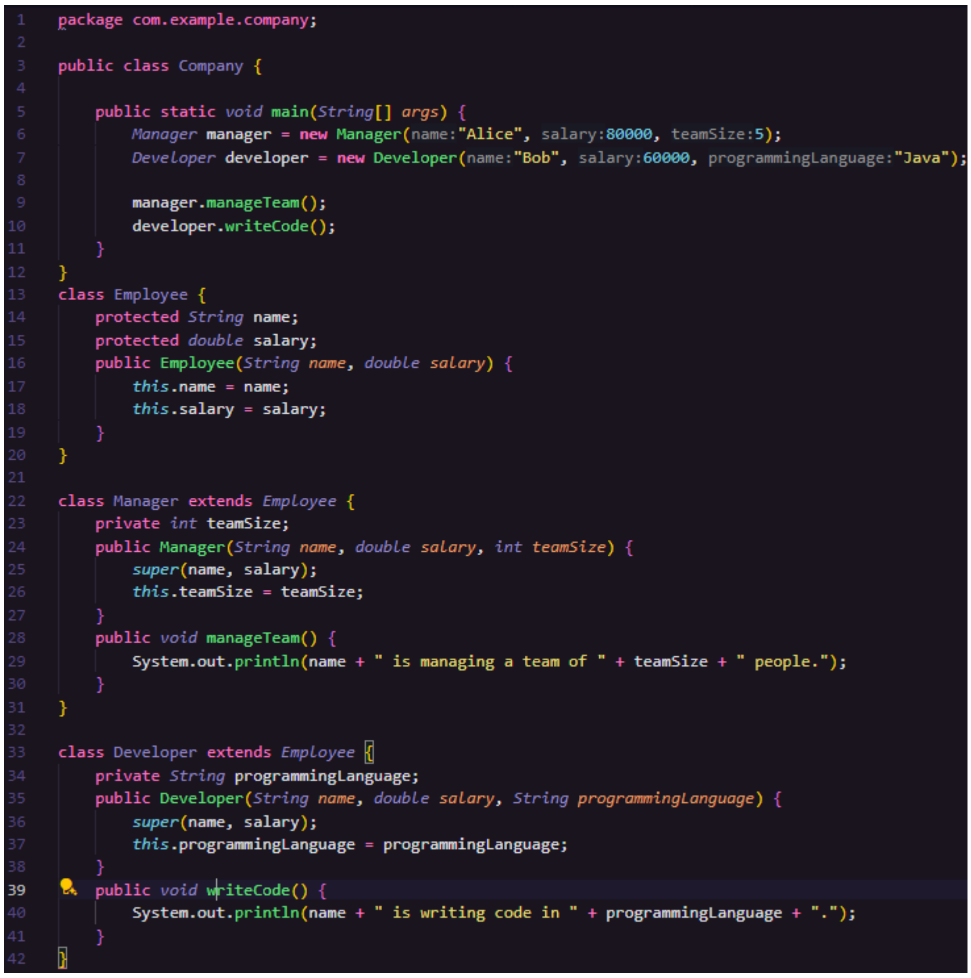
11/12 Push Down Attribute



```java
package com.example.company;

public class Company {

    public static void main(String[] args) {
        Manager manager = new Manager(name:"Alice", salary:80000, teamSize:5);
        Developer developer = new Developer(name:"Bob", salary:60000, programmingLanguage:"Java");

        manager.manageTeam();
        developer.writeCode();
    }
}
class Employee {
    protected String name;
    protected double salary;
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
}

class Manager extends Employee {
    private int teamSize;
    public Manager(String name, double salary, int teamSize) {
        super(name, salary);
        this.teamSize = teamSize;
    }
    public void manageTeam() {
        System.out.println(name + " is managing a team of " + teamSize + " people.");
    }
}

class Developer extends Employee {
    private String programmingLanguage;
    public Developer(String name, double salary, String programmingLanguage) {
        super(name, salary);
        this.programmingLanguage = programmingLanguage;
    }
    public void writeCode() {
        System.out.println(name + " is writing code in " + programmingLanguage + ".");
    }
}
```

Explanation:
The teamSize attribute has been moved from the Employee superclass to the Manager subclass, where it is more relevant. The programmingLanguage attribute has been moved from the Employee superclass to the Developer subclass, where it is more relevant.

12/12 Push Down Method



```java
package com.example.company;

public class Company {

    public static void main(String[] args) {
        Manager manager = new Manager(name:"Alice", salary:80000, teamSize:5);
        Developer developer = new Developer(name:"Bob", salary:60000, programmingLanguage:"Java");

        manager.manageTeam();
        developer.writeCode();
    }
}
class Employee {
    protected String name;
    protected double salary;
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
}

class Manager extends Employee {
    private int teamSize;
    public Manager(String name, double salary, int teamSize) {
        super(name, salary);
        this.teamSize = teamSize;
    }
    public void manageTeam() {
        System.out.println(name + " is managing a team of " + teamSize + " people.");
    }
}

class Developer extends Employee {
    private String programmingLanguage;
    public Developer(String name, double salary, String programmingLanguage) {
        super(name, salary);
        this.programmingLanguage = programmingLanguage;
    }
    public void writeCode() {
        System.out.println(name + " is writing code in " + programmingLanguage + ".");
    }
}
```

Explanation:
The manageTeam() method has been moved from the Employee superclass to the Manager subclass, because it is specific to managers. The writeCode() method has been moved from the Employee superclass to the Developer subclass, because it is specific to developers.

Free and Informed Consent Form (TCLE)

This document, called Free and Informed Consent Form (TCLE), aims to ensure your rights as a participant and will be made available electronically.

Please read carefully and calmly, trying to fully understand the research proposal. There will be no penalty or loss if you do not wish to participate or withdraw your authorization at any time.

To participate in this research you must be over 18 years old and work in the area of software development. Your participation will consist of answering a questionnaire about code metrics that are important in refactoring. The questionnaire will be answered online and will take approximately 7 minutes to complete. We guarantee the secrecy and confidentiality of all information you provide during your participation in the research.

INFORMATION ABOUT THIS RESEARCH:

Objectives: The objective of this research is to complement a study carried out to compose the master's thesis, collecting detailed information from developers about their practices and perceptions in relation to code metrics and the refactoring process.

Importance of the study: The project seeks to better understand how developers measure code quality, identify areas that need improvement and implement refactorings. The data collected will be used to identify trends, challenges and opportunities for improvement in development practices, with the aim of promoting the production of more efficient, sustainable and high-quality code.

Procedures and methodologies: This study is aimed at any developer profile that has some knowledge about code metrics, refactoring or even cares about software quality and/or studies that aim to improve this area. Your participation in the study will consist of answering an online questionnaire. The confidentiality and anonymity of participants will be strictly maintained, encouraging frank and honest responses to provide valuable insights into the topic in question.

Data processing: Throughout the process, respect for privacy and ethics standards will be ensured, as recommended by the General Data Protection Law (Law nº 13,709/2018), ensuring that participants' information is treated confidentially. Initially, the data collected through the de-identified online questionnaire will be anonymized, removing any information that could directly identify the participants, in accordance with the guidelines of the aforementioned legislation. The data will then be organized and coded to facilitate analysis. The results will be presented in an aggregated and non-individually identifiable form, thus preserving the privacy of participants in accordance with applicable legal provisions.
How to contact the researchers: Whenever you wish, you can contact us to obtain information about this research project, your participation or other matters related to the research. The researchers responsible are: Darwin Pinheiro (darwinfederal@alu.ufc.br),

Carla Bezerra (carlailane@ufc.br), Anderson Uchôa (andersonuchoa@ufc.br) and Alessandro Garcia (afgarcia@inf.puc-rio. br).

GUARANTEES TO PARTICIPANTS:

Right to refuse to participate: At any time, you can refuse to participate and withdraw from the research, without embarrassment, penalties or any loss. The information and materials obtained in this research will not be used for purposes other than within the context of this research.

Confidentiality and privacy: We guarantee the complete confidentiality and anonymity of your responses. All sensitive information, such as company or other people's names, will be completely anonymized.
Responsibility of the Researcher: We ensure to provide this document to the research participant and use the data obtained exclusively for the purposes described in this document or in accordance with the consent given by the participant.

APPENDIX C – SIMILARITY SCORE PROGRAM

Program written in Java language to obtain the Similarity Score metric based on the AST of the code using the Eclipse JDT library.

~\eclipse-workspace\Alpha\src\main\java\dw\project\Program.java

```java
package dw.project;

import com.github.javaparser.ast.expr.SimpleName;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.eclipse.jdt.core.dom.*;

public class Program {

    public static double calculateSimilarity(List<ASTNode> allNodes, List<ASTNode> allNodes2) {
        int commonNodes = getCommonNodes(allNodes, allNodes2);
        int totalNodes = getTotalNodes(allNodes, allNodes2);

        return (double) commonNodes / totalNodes;
    }

    private static int getTotalNodes(List<ASTNode> allNodes, List<ASTNode> allNodes2) {
        Set<ASTNode> uniqueNodes = new HashSet<>();

        uniqueNodes.addAll(allNodes);

        for (ASTNode node2 : allNodes2) {
            boolean nodeFound = false;

            for (ASTNode node1 : uniqueNodes) {
                if (node1.getNodeType() == node2.getNodeType() && node1.subtreeMatch(new ASTMatcher(), node2)) {
                    nodeFound = true;
                    break;
                }
            }

            if (!nodeFound) {
                uniqueNodes.add(node2);
            }
        }

        return uniqueNodes.size();
    }

    public static int getCommonNodes(List<ASTNode> allNodes, List<ASTNode> allNodes2) {
        List<ASTNode> commonNodes = new ArrayList<>();

        List<ASTNode> smallerArray = allNodes.size() <= allNodes2.size() ? allNodes : allNodes2;
        List<ASTNode> largerArray = allNodes.size() > allNodes2.size() ? allNodes : allNodes2;

        for (ASTNode node1 : largerArray) {
            for (ASTNode node2 : smallerArray) {
                if (node1 == null && node2 == null) {
                    commonNodes.add(node1);
                    break;
                } else if (node1 != null && node2 != null) {
                    if (node1.getNodeType() == node2.getNodeType() && node1.subtreeMatch(new ASTMatcher(), node2)) {
                        commonNodes.add(node1);
                        break;
                    }
                }
            }
        }

        return commonNodes.size();
    }

    public static void main(String[] args) throws IOException {
        if (args.length == 0) {
            System.out.println("Path of java file:");
            return;
        }

        String caminhoExplorerBefore = args[0];
        String caminhoExplorerAfter = caminhoExplorerBefore.replace("before", "after");

        Path filePath = Path.of(caminhoExplorerBefore);
        Path filePath2 = Path.of(caminhoExplorerAfter);
        String codeString = Files.readString(filePath);
        String codeString2 = Files.readString(filePath2);

        ASTParser parserCode1 = ASTParser.newParser(AST.JLS15);
        parserCode1.setSource(codeString.toCharArray());
        parserCode1.setKind(ASTParser.K_COMPILATION_UNIT);

        ASTParser parserCode2 = ASTParser.newParser(AST.JLS15);
        parserCode2.setSource(codeString2.toCharArray());
        parserCode2.setKind(ASTParser.K_COMPILATION_UNIT);

        CompilationUnit cu = (CompilationUnit) parserCode1.createAST(null);
        CompilationUnit cu2 = (CompilationUnit) parserCode2.createAST(null);

        MyVisitor visitor = new MyVisitor();
        cu.accept(visitor);

        MyVisitor visitor2 = new MyVisitor();
        cu2.accept(visitor2);

        List<ASTNode> allNodes = visitor.getAllNodes();
        List<ASTNode> allNodes2 = visitor2.getAllNodes();

        double score = calculateSimilarity(allNodes, allNodes2);
        System.out.println(score);

    }

}

```

~\eclipse-workspace\Alpha\src\main\java\dw\project\MyVisitor.java

```java
package dw.project;

import org.eclipse.jdt.core.dom.*;
import java.util.ArrayList;
import java.util.List;

public class MyVisitor extends ASTVisitor {
    private int count=0;

    private List<ImportDeclaration> importDeclarations;
    private List<TypeDeclaration> typeDeclarations;
    private List<FieldDeclaration> fieldDeclarations;
    private List<MethodDeclaration> methodDeclarations;
    private List<VariableDeclarationStatement> variableDeclarations;
    private List<SingleVariableDeclaration> singleVariableDeclaration;
    private List<ExpressionStatement> expressionStatement;

    private List<ASTNode> allNodes;

    public MyVisitor() {
        methodDeclarations = new ArrayList<>();
        variableDeclarations = new ArrayList<>();
        typeDeclarations = new ArrayList<>();
        importDeclarations = new ArrayList<>();
        fieldDeclarations = new ArrayList<>();
        singleVariableDeclaration = new ArrayList<>();
        expressionStatement = new ArrayList<>();

        allNodes = new ArrayList<>();

    }

    public List<MethodDeclaration> getMethodDeclarations() {
        return methodDeclarations;
    }

    public List<VariableDeclarationStatement> getVariableDeclarations() {
        return variableDeclarations;
    }

    public List<TypeDeclaration> getTypeDeclarations() {
        return typeDeclarations;
    }

    public List<ImportDeclaration> getImportDeclarations() {
        return importDeclarations;
    }

    public List<FieldDeclaration> getFieldDeclarations() {
        return fieldDeclarations;
    }

    public List<SingleVariableDeclaration> getSingleVariableDeclaration() {
        return singleVariableDeclaration;
    }

    public List<ExpressionStatement> getExpressionStatement() {
        return expressionStatement;
    }

    @Override
    public boolean visit(MethodDeclaration node) {
        methodDeclarations.add(node);
        count++;
        allNodes.add(node);
        return super.visit(node);
    }

    @Override
    public boolean visit(VariableDeclarationStatement node) {
        variableDeclarations.add(node);
        count++;
        allNodes.add(node);
        return super.visit(node);
    }

    @Override
    public boolean visit(TypeDeclaration node) {
        typeDeclarations.add(node);
        count++;
        allNodes.add(node);
        return super.visit(node);
    }

    @Override
    public boolean visit(ImportDeclaration node) {
        importDeclarations.add(node);
        count++;
        allNodes.add(node);
        return super.visit(node);
    }

    @Override
    public boolean visit(FieldDeclaration node) {
        fieldDeclarations.add(node);
        count++;
        allNodes.add(node);
        return super.visit(node);
    }

    @Override
    public boolean visit(SingleVariableDeclaration node) {
        singleVariableDeclaration.add(node);
        count++;
        allNodes.add(node);
        return super.visit(node);
    }

    @Override
    public boolean visit(ExpressionStatement node) {
        expressionStatement.add(node);
        count++;
        allNodes.add(node);
        return super.visit(node);
    }

    public int getTotalCount() {
        return count;
    }

    public List<ASTNode> getAllNodes() {
        return allNodes;
    }

}
```