CHRISTIAN MARLON SOUZA COUTO

A QUALITY-ORIENTED APPROACH TO RECOMMEND MOVE METHOD REFACTORING

LAVRAS – MG

2018

CHRISTIAN MARLON SOUZA COUTO

A QUALITY-ORIENTED APPROACH TO RECOMMEND MOVE METHOD REFACTORING

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para a obtenção do título de Mestre.

Prof. Dr. Ricardo Terra Nunes Bueno Villela

Orientador

LAVRAS – MG

2018

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Couto, Christian Marlon Souza
A Quality-oriented Approach to Recommend Move Method Refactoring / Christian Marlon Souza Couto. – Lavras : UFLA, 2018.
64 p. : il.

Dissertação (mestrado acadêmico)–Universidade Federal de Lavras, 2018.
Orientador: Prof. Dr. Ricardo Terra Nunes Bueno Villela.
Bibliografia.

1. Software Architecture. 2. Refactoring. 3. Quality Metrics. I. Terra, Ricardo. II. Título.

CHRISTIAN MARLON SOUZA COUTO

A QUALITY-ORIENTED APPROACH TO RECOMMEND MOVE METHOD REFACTORING

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para a obtenção do título de Mestre.

APROVADA em 23 de Agosto de 2018.

Prof. Dr. Henrique Santos Camargos Rocha	Inria Lille - Nord Europe
Prof. Dr. Paulo Afonso Parreira Junior	UFLA

Prof. Dr. Ricardo Terra Nunes Bueno Villela
Orientador

LAVRAS – MG
2018

This dissertation is dedicated to my grandmothers in memoriam, Izaura and Geralda, who are inspirations to me of character and love.

ACKNOWLEDGMENTS

Firstly, I would like to thank God for guiding me and for the given opportunities and knowledge to finish my master's degree.

I would like to thank my family—especially Carlito and Rosa—who have always been supporting me.

I would like to thank my girlfriend Kelly for always being on my side, despite the distance between us.

I would like to thank Federal University of Lavras, specially the Department of Computer Science, for the opportunity to obtain a master's degree.

I would like to express my gratitude to my advisor Dr. Ricardo Terra for all support, motivation, patience, and knowledge which help me during these two years of researching.

I would also like to thank the rest of my dissertation committee—Dr. Henrique Rocha and Dr. Paulo Afonso Parreira Junior—for the disposition to read this master dissertation and to participate in my master's defense.

Lastly, I would like to thank my fellow labmates—specially to my friend Arthur Ferreira—for having always supported me in my academic activities.

”Those who can imagine anything, can create the impossible.”
(Alan Turing)

RESUMO

Processos de refatoração são comuns em sistemas de software de grande porte, principalmente quando desenvolvedores negligenciam o processo de erosão arquitetural por longos períodos. Embora existam uma ampla gama de processos de refatoração, poucos são automatizados e levam em consideração os impactos na qualidade do software resultante.

Diante desse cenário, esta dissertação de mestrado propõe uma abordagem de refatoração de sistemas de software orientada a métricas de qualidade de software. Com base no modelo QMOOD (*Quality Model for Object Oriented Design*), a ideia central é mover métodos entre classes de forma a maximizar os valores das métricas de qualidade. Utilizando uma notação formal, o problema pode ser descrito da seguinte forma. Dado um sistema de software S, a abordagem recomenda uma sequência de refatorações $R_1, R_2, \ldots, R_n$ que resulta em versões do sistema $S_1, S_2, \ldots, S_n$, onde $qualidade(S_{i+1}) > qualidade(S_i)$.

Uma calibração empírica foi conduzida utilizando quatro sistemas de código aberto, de modo a encontrar o melhor critério para medir a melhora da qualidade. Dentre dez estratégias diferentes, foi escolhida a que alcançou uma média de *recall* de 57%, cujo critério é comparar as métricas pela porcentagem de melhora da soma dos atributos de qualidade QMOOD.

Três tipos de avaliações foram realizadas para verificar a utilidade da ferramenta implementada, chamada QMove. Primeiro, a abordagem proposta foi aplicada em 13 sistemas de código aberto que foram modificados movendo aleatoriamente um subconjunto de seus métodos para outras classes, verificando posteriormente se a abordagem proposta recomendaria que esses métodos movidos retornassem ao seu local original, e foi alcançado 84% de *recall*. Segundo, foi realizada uma comparação do QMove com duas ferramentas de refatoração do estado-da-arte (JMove e JDeodorant) nos 13 sistemas previamente avaliados, e QMove demonstrou melhor valor de *recall* (84%) que os outros dois (30% e 29%, respectivamente). Terceiro e último, foi feita a mesma comparação utilizando QMove, JMove e JDeodorant em dois sistemas proprietários onde desenvolvedores experientes avaliaram a qualidade das recomendações. QMove obteve oito recomendações avaliadas positivamente pelos desenvolvedores, contra duas e nenhuma do JMove e JDeodorant, respectivamente.

Palavras-chave: Arquitetura de Software; Refatoração; Métricas de Qualidade.

ABSTRACT

Refactoring processes are common in large software systems, especially when developers neglect architectural erosion process for long periods. Even though there are many refactoring approaches, very few consider the refactoring impact on the software quality.

Given this scenario, this master dissertation proposes a refactoring approach to software systems oriented to software quality metrics. Based on the QMOOD (Quality Model for Object Oriented Design), the main idea is to move methods between classes in order to maximize the values of the quality metrics. Using a formal notation, we describe the problem as follows. Given a software system S, our approach recommends a sequence of refactorings $R_1, R_2, \ldots, R_n$ that result in system versions $S_1, S_2, \ldots, S_n$, where $quality(S_{i+1}) > quality(S_i)$.

We empirically calibrated our approach, using four open-source systems, to find the best criteria to measure the quality improvement. By testing ten different strategies, we chose the one that achieved a recall average of 57.5%, whose criterion is to compare the metrics by improvement percentage of the sum of QMOOD quality attributes.

We performed three types of evaluation to verify the usefulness of our implemented tool, called QMove. First, we applied our approach on 13 open-source systems that we modified by randomly moving a subset of its methods to other classes, then checking if our approach would recommend the moved methods to return to their original place, and we achieve 84% recall, on average. Second, we compared QMove against two state-of-art refactoring tools (JMove and JDeodorant) on the 13 previously evaluated systems, and QMove showed better recall value (84%) than the other two (30% and 29%, respectively). Third, we conducted the same comparison among QMove, JMove, and JDeodorant applied in two proprietary systems where experts evaluated the quality of the recommendations. QMove obtained eight positively evaluated recommendations from the experts, against two and none of JMove and JDeodorant, respectively.

Keywords: Software Architecture; Refactoring; Quality Metrics.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

This chapter is organized as follows. Section 1.1 presents the problems the approach proposed in this master dissertation addresses. Section 1.2 lists the main objectives of this dissertation. Section 1.3 introduces the proposed approach. Section 1.4 shows the structure of this dissertation by the organization of its chapters. Finally, Section 1.5 presents the publications generated by this dissertation.

1.1 Problem

The refactoring process changes the code to improve the internal structure without compromising its external behavior (FOWLER, 1999). In the context of software evolution, the use of refactoring (or restructuring) is to improve software quality subcharacteristics, such as maintainability, reusability, complexity, and efficiency (MENS; TOURWÉ, 2004).

Refactoring is also used to remove bad smells. For instance, to treat a Feature Envy bad smell—which occurs when a method `m` in class `C` accesses more components from another class C' rather than its own class—we can use Move Method refactoring for moving `m` from `C` to C' and hence removing this bad smell (FOWLER, 1999).

Studies in the literature address the relationship of quality metrics with software refactoring, such as the fact that coupling and size metrics decrease after refactorings (STROULIA; KAPOOR, 2001) and cohesion metrics increase (BOIS; DEMEYER; VERELST, 2004). The use of metrics during the refactoring process may be important to correct a software system, in order to monitor the metric values to verify that the measures are having a positive impact on the cohesion and coupling in the project, for example.

Currently, there are many refactoring approaches where the degree of automation can vary (MENS et al., 2003; TSANTALIS; CHATZIGEORGIOU, 2009; BAVOTA et al., 2014; TERRA et al., 2017). The majority of these tools search for the best sequence of refactorings using search-based algorithms, with genetic algorithms being the most used (MARIANI; VERGILIO, 2017).

However, with the use of these algorithms, the choice of objectives to be maximized may not impact the quality of software as a whole. Moreover, there is the possibility of not analyzing all the possible refactorings, which can generate a non-optimal sequence of refactorings. This is due to the large number of possibilities, making it impossible to verify all of them.

Besides, there are very few approaches that consider refactoring impact on software quality metrics. Some approaches consider other types of metrics, as number of bad smells (KEBIR; BORNE; MESLATI, 2016; KESSENTINI et al., 2011), Jaccard similarity coefficient (TERRA et al., 2017; TSANTALIS; CHATZIGEORGIOU, 2009), and number of modifications needed to apply refactorings (JENSEN; CHENG, 2010; WANG et al., 2015). Consequently, a software system refactored by one of the aforementioned approaches may result in a system version that worsens its overall quality.

1.2 Objectives

Considering the problems reported in the previous section, the main objective of this master dissertation is to propose a quality-oriented approach that identifies Move Method refactoring opportunities in software systems and recommend these refactorings through a sequence that, when applied, improves software quality metrics values. Our ideia is to verify all Move Method refactorings that can be automatically applied and recommend those that improve the quality of the system w.r.t. the six quality attributes from Quality Model for Object Oriented Design (QMOOD) (BANSIYA; DAVIS, 2002).

To achieve this main objective, we elaborated the following specific objectives:

- To design the underlying algorithm for a quality-oriented approach to recommend Move Method refactorings;
- To calibrate the proposed approach in open-source systems;
- To develop a tool that supports the proposed approach;
- To evaluate the approach in open-source and proprietary systems; and
- To compare the proposed approach with state-of-the-art approaches.

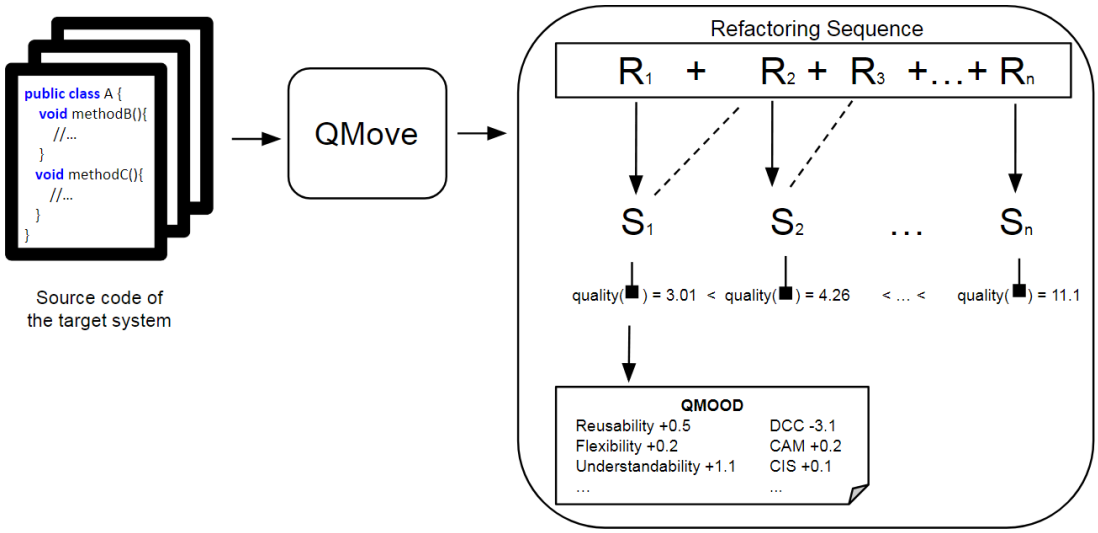
1.3 Proposed approach

On the context of a search-based software engineering research, we propose a semi-automatic software refactoring approach based on software quality metrics. We rely on the measurements of the QMOOD model to recommend Move Method refactorings that improve software quality.

Using a formal notation, we describe the problem as follows. Given a software system S, our approach recommends a sequence of refactorings $R_1, R_2, \ldots, R_n$ that result in system versions $S_1, S_2, \ldots, S_n$, where $quality(S_{i+1}) > quality(S_i)$, being the strategy of comparing the quality by improvement percentage of the sum of QMOOD quality attributes. Indeed, our approach provides software architects with a real grasp whether refactorings improve software quality or not.

We implemented QMove, a prototype plug-in for the Eclipse IDE that supports our proposed approach. The plug-in receives as input a Java system and outputs the better sequence of Move Method refactorings that improves the overall software quality. Figure 1.1 presents an illustrative representation of our approach.

Figure 1.1 – Illustrative representation of the proposed approach



We divide the process of our approach into three phases. In the first phase, our approach through QMove gets the source code of target system. In the second phase, QMove (i) calculates the six quality attributes of the system; (ii) detects every method that could be automatically moved to another class; (iii) moves each method to different classes that can receive it automatically, recalculate the quality attributes, and return it to its original class; and (iv) includes the refactoring that achieved better quality improvement to the recommendation list and come back to step (iii) for the remaining methods, being now the six quality attributes calculated in (i) changed for the new quality attributes values calculated after applying the best found refactoring. In third and last phase, after QMove have processed every method, it presents a recommendation list showing the sequence of Move Method refactorings $R_1, R_2, \ldots, R_n$ ordered by the first to last found recommendation. After performing all the refactorings in sequence (from R_1

to R_n), we expected that $quality(S_n) >>> quality(S)$, being $quality(S_i) > quality(S_{i-1})$ and S_i the system version with the sequence of refactorings R_1 to R_i performed.

We empirically calibrate our approach to find the best strategy to assess software quality improvement. First, we modify four systems by randomly moving a subset of its methods to other classes. Second, we verify if our approach would recommend the moved methods to return to their original place. After testing ten different calibration strategies, we calibrate the approach with the strategy of comparing the metrics by improvement percentage of the sum of QMOOD quality attributes, which achieved the best recall average (57.5%, specifically).

Through QMove, we perform three experiments. First, we evaluate our approach on 13 open-source systems. Similar to our calibration method, we modify the original systems by randomly moving a subset of their methods to other classes. Next, we verify if our approach recommend the moved methods to return to their original classes. As result, QMove could move back 84.2% of the methods, on average. Second, we compare QMove with JMove and JDeodorant on the same 13 systems used before. As result, the state-of-the-art tools showed lower precision, recall, and hence f-score values than the ones achieved by QMove. Last, we perform a comparative evaluation of these tools in two proprietary systems that has been overseen by experts developers, and our approach obtained a greater number of positively evaluated recommendations.

1.4 Outline of the dissertation

The remainder of this dissertation is structured as follows.

- Chapter 2 presents a literature review of the main concepts involved in this dissertation, such as refactorings with a focus on Move Method, quality metrics, QMOOD quality model, and the impact on QMOOD metrics when occurs a Move Method refactoring. Finally, we describe concepts of precision, recall, and f-score, used in the calibration and evaluation processes of our proposed approach.

- Chapter 3 describes our proposed approach, presenting its high-level algorithm. We also describe the calibration process, where we evaluate ten different strategies to find the best criteria to measure the quality improvement. Finally, we describe the tool that implements our proposed approach, which receives as input a software system and outputs a list of Move Method refactoring recommendations.

- Chapter 4 reports three evaluations of our proposed approach. First, we evaluate our approach in 13 open-source systems. Second, we compare the previous results with two state-of-the-art tools (JMove and JDeodorant). Third and last, we evaluate our approach and the two state-of-the-art tools in real-world systems.

- Chapter 5 discusses related works, presenting studies that address different refactorings, in addition to Move Method, and quality metrics, in addition to the QMOOD model.

- Chapter 6 presents the final remarks of this dissertation, highlighting the contributions, limitations, and future work.

1.5 Publications

This dissertation generated the following publications and therefore contains material from them:

- Christian Marlon Souza Couto, Henrique Rocha and Ricardo Terra. Quality-oriented Move Method Refactoring. In *16th BElgian-NEtherlands software eVOLution symposium (BENEVOL)*, pages 13-17, 2017.

- Christian Marlon Souza Couto, Henrique Rocha and Ricardo Terra. A Quality-oriented Approach to Recommend Move Method Refactoring. In *17th Simpósio Brasileiro de Qualidade de Software (SBQS)*, pages 1-10, 2018.

2 BACKGROUND

This chapter presents basic concepts for understanding the approach proposed in this dissertation. Section 2.1 describes basic concepts on refactoring. Section 2.2 details Move Method refactoring. Section 2.3 comments about software quality metrics. Section 2.4 introduces the QMOOD model for quality assessment. Section 2.5 presents the impact of Move Method refactoring on QMOOD quality attributes. Finally, Section 2.6 shows precision, recall, and f-score concepts.

2.1 Refactoring

In the literature, there are different terms for refactoring, such as remodularization and restructuring, and the concepts of each term are interrelated.

Remodularization is a process that changes the modular design of a software for purposes of adaptation, evolution or correction, and this process does not require the behavior preservation of the system to be preserved (TERRA; VALENTE; ANQUETIL, 2016).

Restructuring is the transformation of one form of representation into another at the same level of relative abstraction, while preserving the external behavior of the system (functionality and semantics) (CHIKOFSKY; CROSS, 1990).

Refactoring is basically restructuring applied to object-oriented programming, which can be described as transformations in a software that preserve its behavior, with the main idea to redistribute classes, methods, and attributes by class hierarchy to facilitate future adaptations and extensions (MENS; TOURWÉ, 2004). In practice, however, this concept is broader because developers realize that refactoring involves costs and risks, and they need other types of support in addition to automated refactoring (KIM; ZIMMERMANN; NAGAPPAN, 2012).

There are several types of refactorings (FOWLER, 1999), such as method composition (e.g., Extract Method, Replace Method with Method Object, and Substitute Algorithm), move resources between objects (e.g., Move Method, Move Field, and Extract Class), data organization (e.g., Replace Data Value with Object, Change Value to Reference, and Replace Array with Object), etc. From the several types of refactoring, we highlight the Move Method, which is the core of our proposed approach.

2.2 Move Method

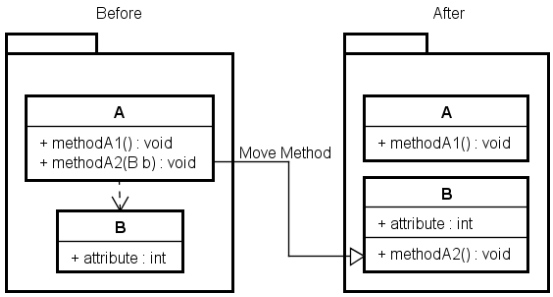
A Move Method refactoring consists in moving a method from one class to another, which can even occur to classes in other packages. There are many reasons to move a method to a different class. A common scenario for this refactoring is when developers realize that a method depends more from members of another class than its own. Fowler named this bad smell as Feature Envy (FOWLER, 1999).

Consider the method `methodA2` for example (Code 2.1). This method, belonging to class `A`, has an object from class `B` as parameter. The statements within the method calls only to methods of class `B` through object `b` (lines 2 and 3). More precisely, the value of one of the attributes in class `B` is printed if its value is not null (line 3). Thus, it makes more sense to move this method from class `A` to class `B`. Therefore, it is used the Move Method refactoring to move method `methodA2` to class `B`. After being moved, the method no longer needs the parameter of type `B`. Figure 2.1 illustrates by a UML class diagram the behavior of the refactoring in our example. It is noted that the dependence of class `A` in class `B` disappeared after the Move Method refactoring. Thus, we observed reduction of coupling and increase of cohesion. The calls of `methodA2` are now adapted to be called by an object of class `B`, eliminating the need for an object of class `A` to perform the call and thus reducing the coupling in classes that contains these method calls.

```
1 public void methodA2(B b){
2    if(b.getAttribute() != NULL){
3       System.out.println(b.getAttribute());
4    } else {
5       System.out.println("Empty");
6    }
7 }
```

Code 2.1 – Method example

Figure 2.1 – Move Method refactoring represented by a UML class diagram



Other reasons for using Move Method involve the following contexts (FOWLER, 1999):

- When there is a type of change in the code that causes the need to make more changes in methods of different classes. Move Method is an alternative to move these methods into a single class, making it easy to perform changes;

- When the system has several `switch` or `case` instructions spread across different classes, and when adding a new clause to the `switch`, there is a need to add it also in others `switch` instructions. With this, one can use Move Method to apply the idea of polymorphism and move the `switch` instruction where necessary, validating this polymorphism;
- To correct the parallel inheritance hierarchy problem, i.e., when creating a subclass of a class, there arises the need to create a subclass for another class as well. One way to remove this hierarchy is by using Move Method;
- To correct a messages chain, i.e., when an object, to be called, must first be called by several other objects. In this case, Move Method is used to break this chain by removing the object to be called from the chain;
- When classes depend on private parts of each other very often. Move Method is one of the strategies to decrease this dependence; and
- When there are alternative classes with different interfaces. Move Method is used until the classes' protocol be the same.

The proposed approach in this dissertation aims to suggest a sequence of Move Method refactorings that improve quality attributes, i.e., without a prior concern to specifically address one of the aforementioned problems. However, the recommended refactorings can indirectly correct these problems, mainly the Feature Envy bad smell, contributing to the improvement of quality of the system.

2.3 Software quality metrics

Metrics are products of quality measurement, productivity, and improvement, and their use is a strong maturity indication of an independent developer or organization (HEVNER, 1997). In the context of software, metrics are widely used and extremely important to measure several factors, one of the most important being software quality.

There are several metrics related to software quality, and in the context of object-oriented systems, there are examples of metrics such as Depth of Inheritance Tree (DIT), Number Of Children (NOC), Coupling Between Object classes (CBO), Weighted Methods per Class (WMC), Response For a Class (RFC), and Lack of Cohesion on Methods (LCOM) (CHIDAMBER; KEMERER, 1994).

Some studies report the refactoring impact on the measurement of quality metrics. Stroulia and Kapoor (STROULIA; KAPOOR, 2001) reported that size and coupling decrease after refactoring. Bois et al. (BOIS; DEMEYER; VERELST, 2004) proposed refactoring instructions to improve cohesion and coupling metrics, obtaining promising results when applying these instructions in open-source projects.

Although there is a wide range of proposed metrics in the literature, this dissertation presents the following 11 metrics contained in QMOOD quality model and elaborated by Bansiya and Davis, which are fundamental to the complete understanding of the proposed approach (BANSIYA; DAVIS, 2002):

- DSC (Design Size in Classes): it counts the number of classes in the project (range from 1 to n);
- NOH (Number of Hierarchies): it counts the number of hierarchical classes in the project by independent inheritance trees (range from 1 to n);
- ANA (Average Number of Ancestors): it represents the average of the number of classes from which a class inherits information in the project. Considering x as the sum of the ancestors for each class and y as the count of all classes, $ANA = x/y$ (range from 1 to n);
- DAM (Data Access Metrics): it represents the ratio between the number of private attributes and the total number of attributes declared in the class. A high DAM value is desired, ranging from 0 to 1, i.e., the closer to 1, the better;
- DCC (Direct Class Coupling): it counts the different number of classes a particular class is directly related. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods. A low DCC value is desired (range from 0 to n);
- CAM (Cohesion Among Methods of Class): it computes the relationship between the methods of a class based on its parameter list. Considering x as the sum of different parameter types of all methods in a class, and y as the total number of methods in a class (except constructor and static methods in both cases), $CAM = x/(x * y)$. The preference is for a value close to 1 (range from 0 to 1);

- MOA (Measure of Aggregation): it measures the extent of the part-whole relationship, performed using attributes. The metric is a count of the number of data declarations whose types are user-defined classes (range from 0 to n);
- MFA (Measures of Functional Abstraction): it measures the proportion of the number of methods inherited by a class to the total number of methods accessible by member methods of the class (range from 0 to n);
- NOP (Number of Polymorphic Methods): it counts the number of methods that may exhibit polymorphic behavior (range from 0 to n);
- CIS (Class Interface Size): it counts the number of public methods in a class (range from 0 to n); and
- NOM (Number of Methods): it counts the total number of methods in a class (range from 0 to n).

Our approach relies on the latter metrics and the QMOOD quality model due to its coverage achieved through its six quality attributes and 11 design properties, which together provide a broader overview of the quality of the software compared to other quality metrics for object-oriented design.

2.4 Quality Model for Object Oriented Design (QMOOD)

QMOOD quality model, proposed by Bansiya and Davis (BANSIYA; DAVIS, 2002), measures software quality aspects in object-oriented projects by six quality attributes based on ISO 9126, namely reusability, flexibility, understandability, functionality, extensibility, and effectiveness. Calculating values for each attribute provides an analysis on software quality as a whole or on a subset of the six mentioned attributes. QMOOD also helps to assess object-oriented design properties, provides search-based refactoring, and quantifies quality attributes with the help of equations (KATOCH; SHAH, 2014).

For the calculation of these six quality attributes, Bansiya and Davis formulated a methodology in four steps: (i) quality attributes definition of a design, (ii) design properties definition of object-oriented projects, (iii) design metrics definition of object-oriented projects, and (iv) relationship of these metrics with design properties and quality attributes.

Quality attributes definition (1st Stage): In this stage, the following six quality attributes are defined to be used to measure the quality of object-oriented systems:

- Reusability: it reflects the presence of features of object-oriented projects that allows a project to be reapplied to a new problem without significant effort;
- Flexibility: it reflects the easiness to incorporate changes in the project, i.e., the ability of the project to be adapted to provide features related to the functionalities;
- Understandability: it refers to the project property that allows it to be easily learned and understood, these characteristics being directly related to the complexity of the project structure;
- Functionality: it represents the responsibilities assigned to the project classes, which are made available by the classes through their public interfaces;
- Extendibility: it measures the ability of the project to achieve desired functionality and behavior using object-oriented design concepts and techniques; and
- Effectiveness: it reflects the presence and use of properties in an existing project that allows the incorporation of new requirements in the project.

The following steps formulate equations in order to obtain the measurements of these quality attributes.

Design properties definition (2nd Stage): Design properties are concepts that can be directly accessed by examination of the structure (internal and external), relationships and functionality of the components, attributes, methods, and classes of the project. For example, the evaluation of a class by its external relations with other classes and the examination of its internal components, attributes, and methods reveals significant information to objectively capture the structural and functional characteristics of a class and its objects.

This stage defines 11 design properties and separates them into two types of sets. The first set is composed by the characteristics of both structural and object-oriented designs: abstraction, encapsulation, coupling, cohesion, complexity, and size of the project. The second set is formed from concepts introduced for the object-oriented paradigm: messages, composition, inheritance, polymorphism, and hierarchy. Next, we describe the definitions of the 11 project properties, according to Bansiya and Davis:

- Design Size: measurement of the number of classes contained in the system;
- Hierarchies: measurement of the number of non-inherited classes that have children in a system, since hierarchies represent different generalization-specialization concepts;
- Abstraction: measurement of the generalization-specialization aspect of the system. System classes that have one or more descendants exhibit this abstraction property;
- Encapsulation: measurement of classes that restricts access to attributes by defining them as private, thus protecting the internal representation of objects;
- Coupling: measurement of the interdependence of an object with other objects in the system. It counts the number of other objects that would have to be accessed by an object in order to execute its functionalities properly;
- Cohesion: measurement of the relation of methods and attributes in the class. A strong intersection of method parameters and attribute types is an indication of strong cohesion;
- Composition: measurement of *part-whole* relationship, which are aggregation relations in an object-oriented system;
- Inheritance: measurement of *is-a* relationship between classes. This relationship is related to the level of nesting of classes in a inheritance hierarchy;
- Polymorphism: measurement of the functionalities of an object that are dynamically determined at run time;
- Messaging: measurement of the number of public methods that are available to other classes. It represents the functionality that a method provides; and
- Complexity: measurement of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.

Design metrics definition (3rd Stage): This stage uses the design properties of the previous stage to formulate metrics to quantitatively measure each of the properties. The metrics are the same already mentioned in Section 2.3, i.e., also elaborated for Bansiya and Davis. Table 2.1 reports each design property related to its equivalent metric (see Section 2.3 for more details about the metrics).

Table 2.1 – QMOOD design properties and its corresponding design metric

Design Property	Design Metric
Size	DSC (Design Size in Classes)
Hierarchies	NOH (Number of Hierarchies)
Abstraction	ANA (Average Number of Ancestors)
Encapsulation	DAM (Data Access Metrics)
Coupling	DCC (Direct Class Coupling)
Cohesion	CAM (Cohesion Among Methods of Class)
Composition	MOA (Measure of Aggregation)
Inheritance	MFA (Measures of Functional Abstraction)
Polymorphism	NOP (Number of Polymorphic Methods)
Messaging	CIS (Class Interface Size)
Complexity	NOM (Number of Methods)

Relationship of design metrics with design properties and quality attributes (4th Stage): In this stage, the metrics related to the design properties and defined in the previous stage were used to formulate equations for each of the quality attributes defined in the first stage. In this way, for each attribute, it is possible to measure its value by means of combinations of the design properties, the value of each property being calculated by means of the metrics defined in the third stage. Table 2.2 reports the attributes and the equations defined to calculate each quality attribute.

Table 2.2 – Equations for QMOOD quality attributes

Quality Attribute	Equation
Reusability	-0.25*Coupling +0.25*Cohesion +0.5*Messaging +0.5*Size
Flexibility	+0.25*Encapsulation -0.25*Coupling +0.5*Composition +0.5*Polymorphism
Understandability	-0.33*Abstraction +0.33*Encapsulation -0.33*Coupling +0.33*Cohesion -0.33*Polymorphism -0.33*Complexity -0.33*Size
Functionality	+0.12*Cohesion +0.22*Polymorphism +0.22*Messaging +0.22*Size +0.22*Hierarchies
Extendibility	+0.5*Abstraction -0.5*Coupling +0.5*Inheritance +0.5*Polymorphism
Effectiveness	+0.2*Abstraction +0.2*Encapsulation +0.2*Composition +0.2*Inheritance +0.2*Polymorphism

For the weighted values defined for each design property, Bansiya and Davis performed an extensive literature review in object-oriented programming books to incorporate ideas of how each design property can influence quality attributes. From this analysis, weighted value of $+1$ or $+0.5$ were initially defined for properties that positively influenced the value of the attribute, and weighted value of -1 or -0.5 for negative influences. The initial weighted values were then

proportionally changed to ensure that the sum of the new weighted values for each design property in a quality attribute added to -1 and 1 range. They chose this weighting scheme through the influences of each design property on quality attributes for its simplicity of application.

Each calculated quality attribute serves as a parameter to provide a notion of the current quality of the software, i.e., greater its value, better is the characteristic assigned to it, opposite to other metrics that provide values between 0 and 1. For example, a system S has reusability attribute value equals to 10. Assume that developers change the source code of S, generating a new version of the system, S', and reusability attribute value increased to 15. Therefore, there was an increase of 50% in reusability value, which means that S' has a greater possibility of their reuse in other systems, compared to S.

2.5 Move Method and QMOOD quality attributes

In order to properly conduct the process of formulating the proposed approach in this dissertation, we identified in Shatnawi and Li's study data about what would be the impact of the Move Method refactoring on the 11 project properties (see Table 2.1) and the six QMOOD quality attributes (see Table 2.2).

Table 2.3 reports an impact analysis on the desgin metrics when applying the Move Method refactoring. A positive impact is represented by $+$, negative impact by $-$, and an impact represented by 0 means that the metric has little or no impact (SHATNAWI; LI, 2011).

Table 2.3 – Move Method refactoring impact on QMOOD design metrics

Design Metric	Impact
DSC (Design Size in Classes)	0
NOH (Number of Hierarchies)	0
ANA (Average Number of Ancestors)	0
DAM (Data Access Metrics)	0
DCC (Direct Class Coupling)	**-**
CAM (Cohesion Among Methods of Class)	**+**
MOA (Measure of Aggregation)	0
MFA (Measures of Functional Abstraction)	0
NOP (Number of Polymorphic Methods)	0
CIS (Class Interface Size)	**-**
NOM (Number of Methods)	0

It is noticed that only DCC, CAM, and CIS metrics have relevant impacts. This means that a Move Method refactoring tends to improve cohesion and coupling properties, and worsen message, with the others properties little changed or unchanged.

Table 2.4 presents an analysis, also conducted by Shatnawi and Li, on the impact of Move Method refactoring on QMOOD quality attributes. It is observed that only reusability and effectiveness do not change, while functionality tends to worsen and the remaining quality attributes tend to improve.

Table 2.4 – Move Method refactoring impact on QMOOD quality attributes

Quality Attributes	Impact
Reusability	0
Flexibility	**+**
Understandability	**+**
Functionality	**-**
Extensibility	**+**
Effectiveness	0

Previous knowledge on the impacts presented in Tables 2.3 and 2.4 is relevant in the calibration process of the proposed approach in this dissertation (Section 3.3). However, some design properties or design metrics may have a different impact than those reported in Shatnawi and Li's study. Such impacts are further discussed in Section 4.3.

2.6 Recall, precision, and f-score

The precision-recall curve can provide a view on the performance of a sample data and is commonly summarised in a single indicator (GOUTTE; GAUSSIER, 2005), such as the f-score value. Given a classifier and an instance, there are four possible outcomes: (i) if the instance is positive and it is classified as positive, it is counted as a true positive (*tp*); (ii) if the instance is positive and it is classified as negative, it is counted as a false negative (*fn*); (iii) if the instance is negative and it is classified as negative, it is counted as a true negative (*tn*); and (iv) if the instance is negative and it is classified as positive, it is counted as a false positive (*fp*) (FAWCETT, 2006).

We use these outcomes to calculate precision and recall values and with them we can obtain the f-score value, as the following equations (OLSON; DELEN, 2008).

$$precision = \frac{tp}{tp + fp} \qquad recall = \frac{tp}{tp + fn}$$

$$f - score = 2 \times \frac{precision \times recall}{precision + recall}$$

These equations were used during the calibration and evaluation process of our work, and details about which outputs we considered and how we define them will be detailed in Section 3.3 and Chapter 4.

3 PROPOSED APPROACH

This dissertation proposes a semi-automatic refactoring approach using Move Method refactorings and the six QMOOD quality attributes used as quality metrics (see Table 2.2). Given a software system S, our approach recommends a sequence of refactorings $R_1, R_2, \ldots, R_n$ that result in system versions $S_1, S_2, \ldots, S_n$, where $quality(S_{i+1}) > quality(S_i)$. By applying the sequence of recommendations, we expect that both the architectural structure and the software quality are improved. Thus, the approach follows a set of steps, which are:

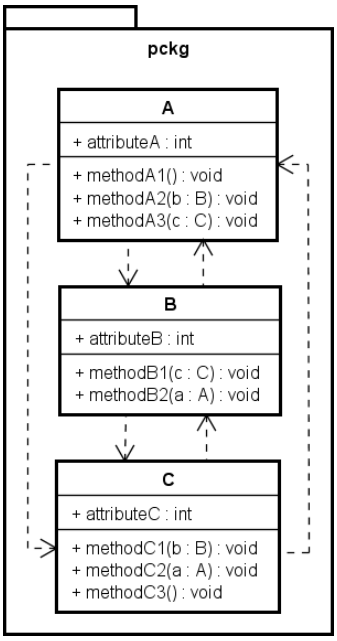
1. Calculate the values of the six quality attributes of the target system;
2. Identify all methods that can be automatically refactored with Move Method. Our approach uses for this identification JDT Eclipse, which provides a procedure to verify for each method of the target system whether it can move through Move Method refactoring and the possible classes that can receive the method under analysis;
3. Move each of the identified methods from step 2 to other classes, recalculate the new values of the quality attributes, and return it to its original class;
4. Select methods that have improved quality when comparing the values of six quality attributes calculated in step 1 with the new values after refactoring. Such improvement is measured by the strategy of comparing quality through the percentage improvement in the sum of QMOOD quality attributes;
5. Among the methods selected in step 4, identify the one that presented the highest increase in the values of the quality attributes, remove it and all of its possible refactorings from the list of methods formulated in step 2, and include it in a list of recommended refactorings;
6. Calculate the new values of the six quality attributes after refactoring the best method found in step 5 and replace these new values with the values calculated in step 1;
7. Repeat steps 3 to 6 as long as there is a method that improves the values of the quality attributes; and
8. If it does not have a method that improves the quality attribute values, the list of recommendations with Move Method refactorings is shown from the first to the last recommendation found in step 5.

This chapter is structured as follows. Section 3.1 presents a motivation example of the application of our proposed approach to a system. Section 3.2 shows the high-level algorithm of our approach. Section 3.3 reports the calibration process of our approach. Finally, Section 3.4 shows the implemented tool of our proposed approach.

3.1 Motivation example

This section illustrates a Move Method refactoring scenario where our approach could be useful. Suppose a small Java system `S` with three classes: `A`, `B`, and `C`. Class `A` has three methods: `methodA1`, `methodA2`, and `methodA3`; class `B` has two methods: `methodB1` and `methodB2`; and class `C` has three methods: `methodC1`, `methodC2`, and `methodC3`. A well-design architecture for this example would have the methods with parameters in the class of the parameter type. More specifically, `methodB2` and `methodC2` that receive an `A` object as a parameter should be in class `A`; `methodA2` and `methodC1` that receive a `B` object as a parameter should be in class `B`; and `methodA3` and `methodB1` that receive a `C` object as a parameter should be in class `C`. Figure 3.1 shows a UML diagram of the classes described in our example and the dependencies between them.

Figure 3.1 – UML class diagram of system `S` in our motivation example



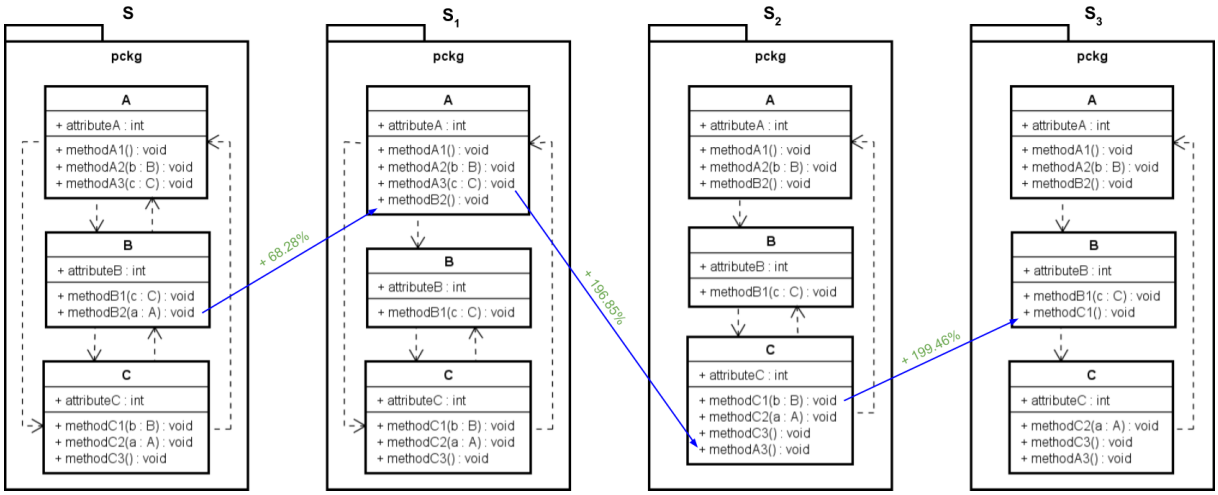
When we apply our approach to system `S`, we first compute the QMOOD quality attributes for `S`. Then, we detect `methodA2`, `methodA3`, `methodB1`, `methodB2`, `methodC1`, and `methodC2` as the methods that could be moved to other classes. Method `methodA2` is moved

to class B creating the new system version S_{A2}. We recompute the quality metrics for S_{A2} and then we return the method to class A, where it originally came from. Next, method methodA3 is moved to class C creating the new system version S_{A3}, then we recompute the quality metrics for S_{A3} and return the method to class A. We do the same to remain methods, where methodB1 generates S_{B1}, methodB2 generates S_{B2}, methodC1 generates S_{C1}, and methodC2 generates S_{C2}.

After, we verify from the generated systems the one that obtained the best quality improvement, and we found system S_{B2} as the best, with 68.28% of quality improvement, which was calculated by the following steps: (i) calculation of the sum of quality attributes values of system S; (ii) calculation of the sum of quality attributes values of system S_{B2}; and (iii) calculation of improvement percentage between the sums of steps (i) and (ii). The same is done to each generated system, and the improvement percentages are compared to find the best improvement, which in this case was 68.28% for system S_{B2}.

Therefore, we definitely move methodB2 from class B to class A, and the generated system S_{B2} is called now S_1. Figure 3.2 shows the architecture of S and S_1 systems and the remain systems generated during the execution of our approach. We can note that in system S_1 methodB2 does not have a parameter of type A and hence the dependency from class B to class A in system S no longer exists in system S_1, resulting in an improvement to cohesion and coupling from S to S_1, besides the quality improvement, i.e., $quality(S_1) > quality(S)$.

Figure 3.2 – System versions in UML class diagrams of our motivation example



Our approach then repeats the process to system S_1 by simulating to move the remain methods methodA2, methodA3, methodB1, methodC1, and methodC2 to other classes and recalculating the quality metrics again. After, we found that methodA3 has the best quality improvement (196.85%) regarding system S_1, then we definitively move it from class A to class C

on system version S_1, generating the new system version S_2 where the dependency from class A to class C no longer exists. Therefore, we have $quality(S_2) > quality(S_1) > quality(S)$.

Following, our approach continues the process to system version S_2 with methods methodA2, methodB1, methodC1, and methodC2. It is detected that methodC1 has the best quality improvement (199.45%) regarding system S_2, then we move it from class C to class B on system version S_2, generating the new system version S_3 where does not have the dependency from class C to class B. Thus, with the new system version S_3, we have $quality(S_3) > quality(S_2) > quality(S_1) > quality(S)$.

Last, our approach executes on system version S_3 by moving the remain methods methodA2, methodB1, and methodC2. However, no one of the moves improves the quality on system version S_3, then our approach finishes the analysis. Table 3.1 shows the QMOOD quality attributes for S, S_1, S_2, and S_3, and for each quality attribute from system S_1 to system S_3, we show the improvement compared to the previous version, e.g., reusability has the value of 4.29 in system S and 4.67 in system S_1, an improvement of 0.38 on its value. Besides, Table 3.1 shows the sum of quality attributes and the quality improvement of each system, which is calculated by the sum of the six quality attributes to previous system version, the sum of the six quality attributes to new system, and then the improvement percentage between these sums.

Table 3.1 – Variation of QMOOD quality attributes for our motivation example

Quality Attribute	**S**	**S_1**	**S_2**	**S_3**
Reusability	4.29	4.67 (+0.38)	5.01 (+0.34)	5.33 (+0.32)
Flexibility	-1.50	-1.25 (+0.25)	-1.00 (+0.25)	-0.75 (+0.25)
Understandability	-5.55	-5.04 (+0.51)	-4.60 (+0.44)	-4.18 (+0.42)
Functionality	2.56	2.62 (+0.06)	2.66 (+0.04)	2.70 (+0.04)
Extendibility	-2.50	-2.00 (+0.5)	-1.50 (+0.5)	-1.00 (+0.5)
Effectiveness	0.20	0.20 (0)	0.20 (0)	0.20 (0)
Sum (Improvement)	**-2.5**	**-0.8 (68.28%)**	**0.77 (196.85%)**	**2.3 (199.46%)**

Since system version S_3 shows better quality attributes, our approach would recommend methodB2, methodA3, and methodC1 to be moved to class A, C, and B, respectively and in that order (as previous illustrated in Figure 3.2).

3.2 Algorithm

Algorithm 1 describes our proposed approach. It is worth noting that before we execute the algorithm, we make a copy of the analyzed system, and the algorithm is executed in this copy (and not in the actual system).

Algorithm 1: Proposed Approach Algorithm

```
1  Require: methods, a list with every method and their respective class from the analyzed
     system
2  Ensure: recommendations, an ordered sequence of Move Method refactoring that can be
     applied to the analyzed system
3  begin
4      potRefactor := ∅
5      currentMetrics := calculateMetrics()
6      for each method m in methods do
7          if m can be automatically refactored to a class C then
8              potRefactor := potRefactor + {m,C}
9          end
10     end
11     candidates := ∅
12     metrics := ∅
13     while potRefactor ≠ ∅ do
14         for each refactoring ref in potRefactor do
15             applyRefactoring(ref)
16             metrics := calculateMetrics()
17             undoRefactoring(ref)
18             if fitness(metrics) > fitness(currentMetrics) then
19                 candidates := candidates + {ref,metrics}
20             end
21         end
           /* find the refactoring with the best metrics */
22         bestRefactoring := maxMetrics(candidates)
23         applyRefactoring(bestRefactoring)
24         potRefactor := potRefactor \ {bestRefactoring}
25         recommendations := recommendations + {bestRefactoring}
26         currentMetrics := bestRefactoring.metrics
27     end
28 end
```

The algorithm receives as input a list containing all methods with their respective class from the analyzed system. The output is a sequence of Move Method refactorings that resulted in better quality metrics, ordered by the first to last found recommendation.

First, it calculates the current six QMOOD quality metrics for the analyzed system (line 5). Second, it determines the methods of the system (m) that can be automatically moved to other classes (C) with support of JDT Eclipse (lines 6-10) and stores the pairs (m,C) in the list of potential refactorings (line 8).

The next loop (lines 13-27) finishes when the list containing the methods for potential refactoring is empty. Now, each method in the potential refactoring list is moved (line 15), the quality metrics are recalculated after moving the method (line 16), and the method returns to its

original class (line 17). If the quality measurements are better than the current ones (line 18), then the method is added to our list as a candidate for refactoring (line 19).

After we measure every method, we select the one that achieved the best quality metric improvement (line 22). The best refactoring is applied to the system copy (line 23), removed from the potential refactoring list (in fact, we remove all refactorings where it is the origin method) (line 24), and added to the recommendations (line 25). The new calculated metrics for the best refactoring becomes the system baseline now (line 26). After the execution of Algorithm 1, the sequence of refactorings is recommended to the user.

3.3 Calibration

Our calibration is related to the `fitness` function from Algorithm 1 (line 18). The `fitness` function defines how we compare the quality attributes to determinate if there is an improvement according to our requirements. Our objective is to identify the best set of requirements for the `fitness` function to make our approach recommend better refactoring options.

3.3.1 Subject systems

Table 3.2 reports information about the four systems we use in calibration process, such as size in terms of lines of code (LOC), and number of classes and methods.

Table 3.2 – Subject systems in the calibration process

System	Version	# of classes	# of methods	LOC
JHotDraw	4.6	674	6,533	80,536
JUnit	r4.12	1,092	2,811	26,111
MyAppointments	-	22	99	1,213
MyWebMarket	-	18	107	1,034

We chose these four systems because they were implemented following commonly architectural standards and hence most of their methods are probably located in the correct classes. We randomly moved *to different classes* 20 methods of JHotDraw and JUnit, and five methods of MyAppointments and MyWebMarket.

The information about these methods and classes (original and the one it has been moved to), we called *Gold Set*. We rely on the *Gold Set* to verify if our approach recommends moving those methods back to their original classes. In theory, it would be chosen the `fitness` function that recommends more methods from the *Gold Set*.

Next step of calibration process consists of elaborating different strategies for the `fitness` function configuration to observe which one is the most effective w.r.t. the larger number of methods from the *Gold Set* being moved back to their original classes.

3.3.2 Strategies

For this calibration process stage, we define five different types of fitness functions using two different kinds of metrics values—the absolute and relative values of QMOOD quality attributes—and we run our approach in the modified versions of the systems for each type of calibration.

Absolute values refer to original values of each calculated metric, and the relative values refer to improvement (or worsening) percentages of the metrics after a Move Method refactoring. For example, a system has absolute values of 1, 2, and 3 for reusability, flexibility, and understandability, respectively. After a refactoring, the new version of the system has the absolute values of 2, 3, and 4, respectively. In terms of absolute values, each metric increased by 1 its value, while in relative values, each metric increased by 100%, 50% and 33.3%, respectively.

We assume S as a system, S' as its version after a Move Method refactoring, and M as a metrics set consisting of reusability, flexibility, understandability, functionality, extendibility, and effectiveness. Also, we consider $M\%$ as the set with the percentage difference between the values of each metric of M in S and S'.

In first calibration type, our criterion is the more simplistic where we verified if none of the quality attributes decreased and at least one attribute increased. Therefore, Equation 3.1 uses absolute values of each metric before and after the refactoring, while Equation 3.2 uses relative values.

$$\text{Abs\#1: } \forall m \in M,\ m(S') \geq m(S) \wedge \exists m \in M,\ m(S') > m(S) \tag{3.1}$$

$$\text{Rel\#1: } \forall m\% \in M\%,\ m\% \geq 0 \wedge \exists m\% \in M\%,\ m\% > 0 \tag{3.2}$$

Nevertheless, we discovered that the *effectiveness* values get worse in the majority of the *Gold Set* and hence we discarded correct recommendations. In the second calibration, since the *effectiveness* rarely changed in the first calibration, we adjusted the `fitness` function to disregard this quality attribute, while maintaining the other criteria from the first calibration. Therefore, we altered the absolute and relative `fitness` functions (Equations 3.3 and 3.4, respectively).

$$\begin{aligned} \text{Abs\#2: } & \forall m \in M \setminus \{\text{effectiveness}\},\ m(S') \geq m(S) \\ \wedge \quad & \exists m \in M \setminus \{\text{effectiveness}\},\ m(S') > m(S) \end{aligned} \tag{3.3}$$

$$\begin{aligned} \text{Rel\#2: } & \forall m\% \in M\% \setminus \{\text{effectiveness}\},\ m\% \geq 0 \\ \wedge \quad & \exists m\% \in M\% \setminus \{\text{effectiveness}\},\ m\% > 0 \end{aligned} \tag{3.4}$$

In the third calibration, our criterion is as simplistic as the first one where we compare the overall sum of all six quality attributes. Thus, Equation 3.5 represents absolute version of the function `fitness` and Equation 3.6 represents the relative one.

$$\begin{aligned} \text{Abs\#3: } & s = sum(M), \\ & s(S') > s(S) \end{aligned} \tag{3.5}$$

$$\text{Rel\#3: } sum(M\%) > 0 \tag{3.6}$$

In the fourth calibration, we modified the `fitness` function based on the following two observations: (i) in the second calibration, *flexibility*, *understandability*, and *extensibility* improved but the remaining attributes (*reusability* and *functionality*) decreased; and (ii) Shatnawi and Li stated that *Move Method* refactoring usually increases the values for *flexibility*, *understandability*, and *extensibility* (SHATNAWI; LI, 2011). Therefore, particularly in this calibration, we consider only these three attributes, disregarding the others. Therefore, consider M' as a subset of M consisting of flexibility, understandability, and extensibility metrics and $M'\%$ as the percentage difference between the values of each metric of M' in S and S'. Equations 3.7 and 3.8 represent absolute and relative versions of the `fitness` function, respectively.

$$\begin{aligned} \text{Abs\#4: } & \forall m \in \text{M'},\ m(S') \geq m(S) \\ \wedge \quad & \exists m \in \text{M'},\ m(S') > m(S) \end{aligned} \tag{3.7}$$

$$\begin{aligned} \text{Rel\#4: } & \forall m\% \in \text{M'}\%,\ m\% \geq 0 \\ \wedge \quad & \exists m\% \in \text{M'}\%,\ m\% > 0 \end{aligned} \tag{3.8}$$

In the fifth and last calibration type, we used the following three design metrics (Table 2.1): CAM (cohesion), DCC (coupling), and CIS (messaging). We chose these metrics

because they are the QMOOD design metrics that usually change when a Move Method refactoring occurs. We then establish the criteria for the `fitness` function that cohesion, coupling, and messaging cannot decrease. Therefore, consider *DM* as a set with CAM, DCC, and CIS design metrics and *DM%* as the percentage difference between the values of each design metric of *DM* in S and S'. Equations 3.9 and 3.10 represent absolute and relative `fitness` functions, respectively.

$$\begin{aligned} &\text{Abs\#5: } \forall m \in \text{DM},\ m(S') \geq m(S) \\ &\wedge \quad \exists m \in \text{DM},\ m(S') > m(S) \end{aligned} \tag{3.9}$$

$$\begin{aligned} &\text{Rel\#5: } \forall m\% \in \text{DM\%},\ m\% \geq 0 \\ &\wedge \quad \exists m\% \in \text{DM\%},\ m\% > 0 \end{aligned} \tag{3.10}$$

3.3.3 Results

Table 3.3 reports the strategies for each calibration type (*ST*), the number of recommended methods (*RM*), the recommendations from the *Gold Set* (*GM*) and we also calculated precision, recall, and f-score, considering *GM* as true positives, $RM - GM$ as false positives, and $|Gold\ Set| - GM$ as false negatives.

We determined that we should focus our analysis on recall values. The measure of precision and f-score is jeopardized since we cannot ensure that recommendations that do not belong to the *Gold Set* are indeed wrong. In other words, we can mostly guarantee that the methods we moved around (i.e., those that belong to the *Gold Set*) are misplaced.

The first and second calibration strategies obtained an average recall of 38.8% for both Abs#1, Rel#1, Abs#2, and Rel#2. The methods recommended for each system in these strategies were the same, and hence they have same number of recommendations and recall values.

For the third calibration, Abs#3 had the average recall of 56.2%, the best result so far, since is an increase of 17.4% w.r.t. the previous strategies. However, for Rel#3, the average recall rose to 57.5%, i.e., a subtle increase of 1.3%. It occurs due exclusively to the difference of recall values calculated for JUnit, which in Abs#3 was 80% and Rel#3 was 85%. Thus, we now consider Rel#3 to be the best.

In the fourth calibration, the average recall for both Abs#4 and Rel#4 were 56.2% (the same found for Abs#3), so we keep Rel#3 as the best. Last, in the fifth calibration, Abs#5 had

Table 3.3 – Recall, precision, and f-score results for subject systems of calibration

JHotDraw						**JUnit**				
ST	RM	GM	P	R	F	RM	GM	P	R	F
Abs#1	33	5	15.1%	25.2%	18.8%	30	10	33.3%	50.0%	40.0%
Rel#1	33	5	15.1%	25.2%	18.8%	30	10	33.3%	50.0%	40.0%
Abs#2	33	5	15.1%	25.2%	18.8%	30	10	33.3%	50.0%	40.0%
Rel#2	33	5	15.1%	25.2%	18.8%	30	10	33.3%	50.0%	40.0%
Abs#3	43	13	30.2%	**65.0%**	41.2%	39	16	41.0%	80.0%	54.2%
Rel#3	43	13	30.2%	**65.0%**	41.2%	39	17	43.5%	**85.0%**	57.6%
Abs#4	40	13	32.5%	**65.0%**	43.3%	36	16	44.4%	80.0%	57.1%
Rel#4	40	13	32.5%	**65.0%**	43.3%	36	16	44.4%	80.0%	57.1%
Abs#5	36	4	11.1%	20.0%	14.2%	30	9	30.0%	45.0%	36.0%
Rel#5	37	5	13.5%	25.0%	17.5%	52	11	21.1%	55.0%	30.5%
MyAppointments						**MyWebMarket**				
ST	RM	GM	P	R	F	RM	GM	P	R	F
Abs#1	4	2	50.0%	**40.0%**	44.4%	2	2	100.0%	40.0%	57.1%
Rel#1	4	2	50.0%	**40.0%**	44.4%	2	2	100.0%	40.0%	57.1%
Abs#2	4	2	50.0%	**40.0%**	44.4%	2	2	100.0%	40.0%	57.1%
Rel#2	4	2	50.0%	**40.0%**	44.4%	2	2	100.0%	40.0%	57.1%
Abs#3	4	2	50.0%	**40.0%**	44.4%	2	2	100.0%	40.0%	57.1%
Rel#3	4	2	50.0%	**40.0%**	44.4%	2	2	100.0%	40.0%	57.1%
Abs#4	4	2	50.0%	**40.0%**	44.4%	2	2	100.0%	40.0%	57.1%
Rel#4	4	2	50.0%	**40.0%**	44.4%	2	2	100.0%	40.0%	57.1%
Abs#5	4	2	50.0%	**40.0%**	44.4%	2	2	100.0%	40.0%	57.1%
Rel#5	4	2	50.0%	**40.0%**	44.4%	3	3	100.0%	**60.0%**	75.0%
Acronyms: ST - Strategy, Abs - Absolute Calibration, Rel - Relative Calibration, RM - Recommended Methods, GM - Gold Set Methods, P - Precision, R - Recall, F - F-Score.										

an average recall of 36.2%, while Rel#5 had an average of 45%. These values are lower than the one of Rel#3, so they were discarded.

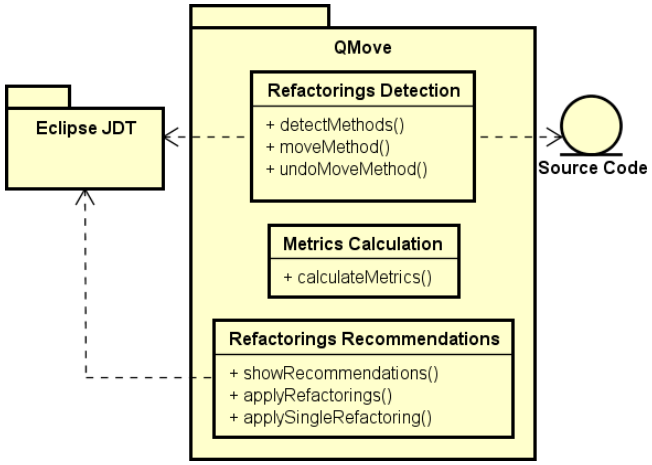
Thus, we chose calibration Rel#3, which obtained the highest average recall value of 57.5%, to be used by our approach as the `fitness` function, i.e., the criterion of comparing the metrics by improvement percentage of the sum of QMOOD quality attributes.

3.4 Tool support

We implemented our approach as a plug-in for the Eclipse IDE, called QMove[1]. We decided to implement the tool as a plug-in for Eclipse since this platform has automated refactoring features, including Move Method one. Figure 3.3 shows a high-level representation of QMove architecture with its following three main modules: (i) refactorings detection, (ii) metrics calculation, and (iii) refactorings recommendations.

[1] https://github.com/pqes/QMove, verified on July 4th, 2018

Figure 3.3 – UML class diagram of QMove's architecture



- **Refactorings Detection**: it receives and makes a copy of the system project under analysis, creates a list with all its methods, detects methods that can be automatically moved, moves the method to other classes, and moves it back to its original class. Eclipse JTD (Java Development Tools) assists in performing these refactorings by providing libraries that facilitate Move Method operations.

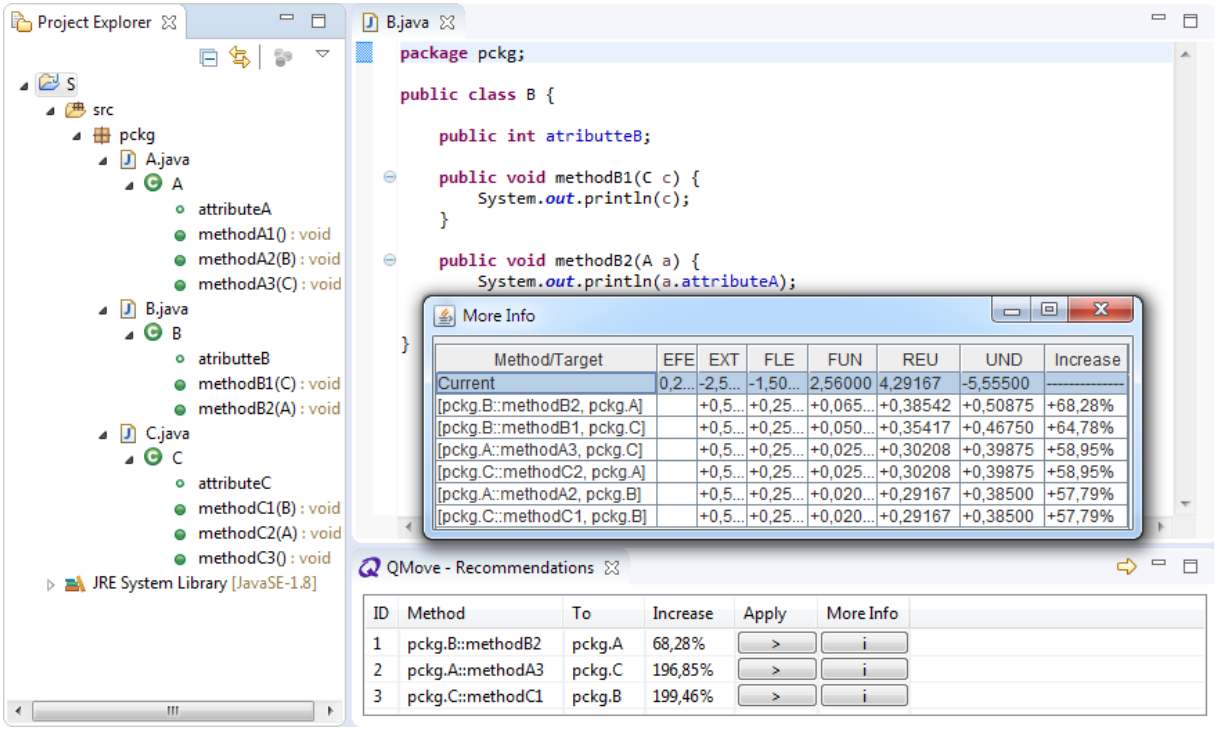
- **Metrics Calculation**: it calculates the six QMOOD metrics. We reused and adapted part of the source code of *Eclipse Metrics Plugin 3*[2] to calculate the metrics.

- **Refactorings Recommendations**: it displays the refactorings recommendations list to the user and applies the refactorings. The latter task is also assisted by Eclipse JDT.

Figure 3.4 demonstrates an example of using QMove on a system. When developers run the tool, it shows a view with the recommended refactoring sequence. The view shows the method's current location, the suggested class to move it, and the percentage increase in QMOOD quality metrics with the refactoring performed. Note, for instance, that recommendation ID 1 suggests to move `methodB2` from `pckg.B` to `pckg.A` since it improves 68.28% of metrics values. For each recommendation, QMove allows to apply it or check detailed information w.r.t. its QMOOD metrics values. Note again that, while effectiveness remains the same, recommendation ID 1 improves extendibility from -2.5 to -2 (+0.5), flexibility from -1.5 to -1.25 (+0.25), functionality from 2.56 to 2.625 (+0.065), reusability from 4.291 to 4.676 (+0.385), and understandability from -5.555 to -5.047 (+0.508). Finally, there is also the option to automatically apply all refactorings in the order our approach suggests.

[2] https://github.com/qxo/eclipse-metrics-plugin

Figure 3.4 – QMove plug-in screenshot



One could question rec. ID 1 improves 68.28% and rec. ID 2 improves 196.85%. However, as we previously detailed in our motivation example (Section 3.1), rec. ID 1 is the best in version S_1 and rec. ID 2 is the best in version S_2. Particularly in this example, rec. ID 2 in S_1 would improve 58.95%, which is less than 68.28%. However, if the user decides to apply only one refactoring out of sequence, e.g., recs. ID 2 or 3 applied first than rec. ID 1, QMove starts a re-execution to guarantee that the generated refactoring sequence improves quality as showed in recommendations view, which may not happen if a refactoring is applied outside the defined sequence.

4 EVALUATION

This section evaluates our approach, where we performed three types of evaluation. Section 4.1 reports a synthesized analysis running our QMove tool on 13 open-source systems. Section 4.2 compares QMove with two state-of-the-art tools similar to ours by running JMove and JDeodorant on the same 13 systems used in the previous evaluation. Finally, Section 4.3 reports an evaluation in a real scenario, where we ran our tool, JMove, and JDeodorant in two proprietary systems together with their software architects.

4.1 Synthesized evaluation

This section evaluates our proposed refactoring approach through QMove. We rely on 13 open-source systems (Table 4.1) that possess well-defined architectures and are active projects. These systems have been used in the evaluation of a third-party work (TERRA et al., 2017), which facilitates comparing our approach to JMove and JDeodorant.

Table 4.1 – Subject systems in the evaluation process

System	Version	# of classes	# of methods	LOC
Ant	1.8.2	1,474	12,318	127,507
ArgoUML	0.34	1,291	8,077	67,514
Cayenne	3.0.1	2,795	17,070	192,431
DrJava	r5387	788	7,156	89,477
FreeCol	0.10.3	809	7,134	106,412
FreeMind	0.9.0	658	4,885	52,757
JMeter	2.5.1	940	7,990	94,778
JRuby	1.7.3	1,925	18,153	243,984
JTOpen	7.8	1,812	21,630	342,032
Maven	3.0.5	647	4,888	65,685
Megamek	0.35.18	1,775	11,369	242,836
WCT	1.5.2	539	5,130	48,191
Weka	3.6.9	1,535	17,851	272,611

Similarly to our calibration, we modified the subject systems by randomly moving their methods to other classes. Those methods represent our *Gold Set* and our evaluation consists in verifying whether QMove recommend to move methods from our *Gold Set* back to their original classes.[1] Table 4.2 reports the evaluation results for each system, the total number of

[1] We do not rely on the same *Gold Set* from (TERRA et al., 2017) because we use a more recent version of Eclipse to implement and run our approach. We noticed, nevertheless, that Eclipse Photon has more preconditions to apply a Move Method than used to be when the nowadays version could not move back some methods from (TERRA et al., 2017).

recommended methods, the *Gold Set* (GS) size, the recommendations from the *Gold Set*, and the achieved f-score, precision, and recall.

Table 4.2 – Recall, precision, and f-score results for subject systems of evaluation

System	Recs.	GS Size	Recs. GS	F-score	Prec.	Recall
Ant	135	25	25	31.2%	18.5%	100.0%
ArgoUML	71	32	13	25.2%	18.3%	40.6%
Cayenne	245	47	46	31.5%	18.7%	97.8%
DrJava	90	18	16	29.6%	17.7%	88.8%
FreeCol	112	17	13	20.1%	11.6%	76.4%
FreeMind	47	12	11	37.2%	23.4%	91.6%
JMeter	52	25	22	57.1%	42.3%	88.0%
JRuby	101	41	23	32.3%	22.7%	56.1%
JTOpen	162	39	36	35.8%	22.2%	92.3%
Maven	36	24	22	73.3%	64.1%	91.6%
Megamek	193	35	32	28.0%	16.5%	91.4%
WCT	46	29	25	66.6%	54.3%	86.2%
Weka	114	31	29	40.0%	25.4%	93.5%
Average				39.1%	**27.1%**	**84.2%**

Our evaluation results shows 84.2% average recall for methods in the *Gold Set*. This result is similar to the one found in the calibration process, where the highest recall in our chosen strategy was 85%. On the other hand, the average f-score and precision in the evaluation were lower than the calibration. It is somehow expected since the systems used in the calibration process have been carefully implemented following commonly architectural standards.

We also performed a more detailed analysis of the recommendations, considering the precision and recall values for each recommendation, allowing the behavior observation of these values during the execution of our approach. Therefore, Figures 4.1 and 4.2 show a graph containing the precision and recall results, respectively, for all subject systems. We used the logarithmic scale for a better representation of data variation, mostly in relation to the first found recommendations.

We can note that precision values in general tend to be higher in the first recommendations, and throughout of the remaining recommendations, the precision undergoes a decline until the last recommended method. Regarding recall, the observed behavior is the opposite of precision, i.e., a low recall in the first recommendations and a high value in the last recommendations. This behavior shows a tendency that the first recommendations provided by QMove have high accurate in finding methods that are erroneously located, and throughout the remaining recommendations, most of these methods are recommended but with less precision.

Figure 4.1 – Precision graph of QMove for the evaluated systems

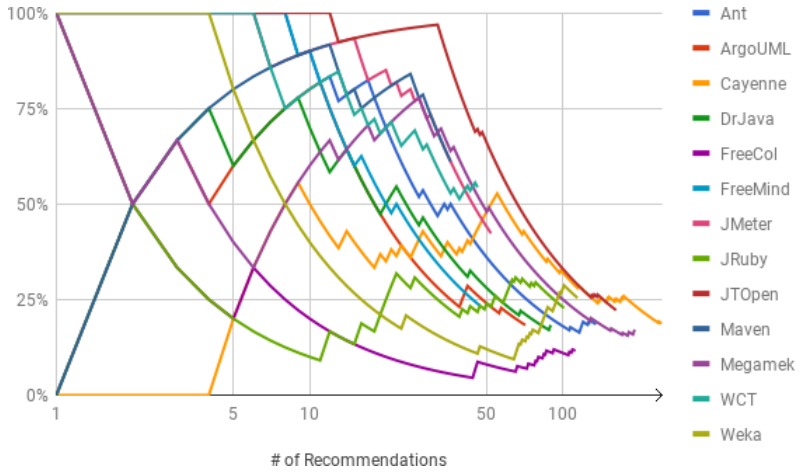
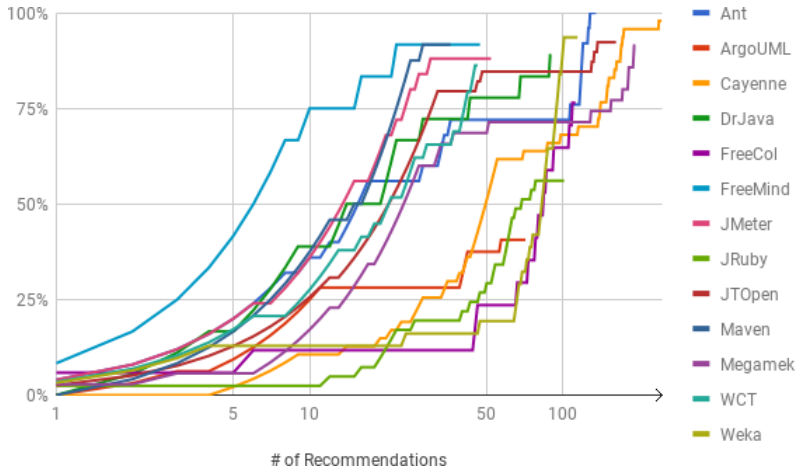


Figure 4.2 – Recall graph of QMove for the evaluated systems

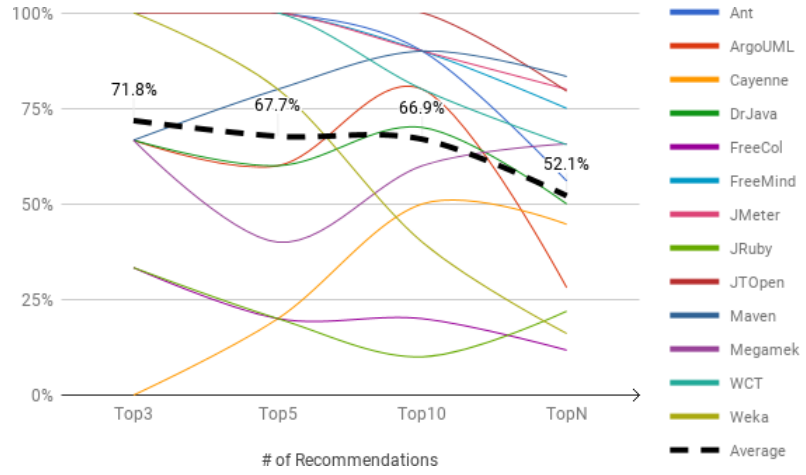


In order to find the situation where our approach provides the best possible precision and recall values, we made further analysis through precision and recall values at different stages of the Move Method recommendations, specifically when we set the number of recommendations as three ($Top3$), five ($Top5$), ten ($Top10$), and n ($TopN$), being n the size of the gold set for each system used in the evaluation (e.g., $n = 25$ for Ant according to Table 4.2).

Figure 4.3 graphically illustrates the precision behavior and shows the data referring to the average precision of $Top3$ to $TopN$, the letter represented by the dotted line. The average precision is 71.8% for the first three recommendations, 67.7% for the first five, 66.9% for the first ten, and 52.1% for the first n recommendations. Note that our approach is more precise in the first recommendations to move methods that improve QMOOD quality attributes.

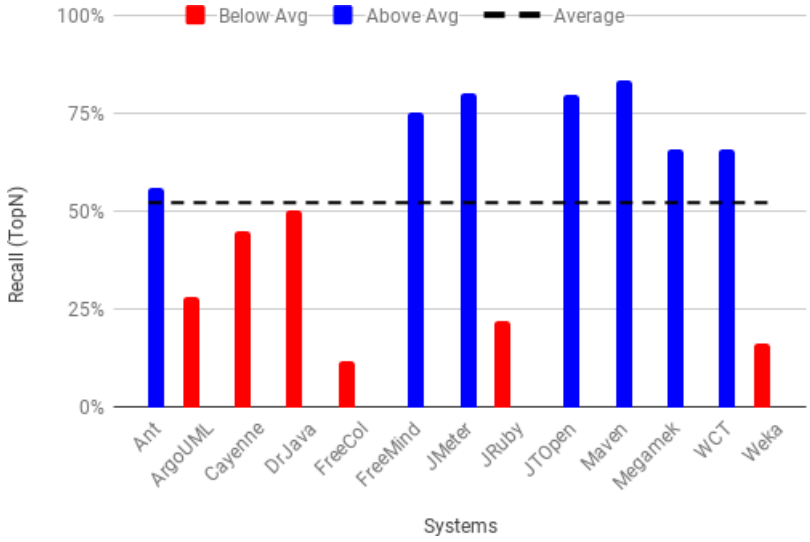
Figure 4.4 contains a graph representing the recall value for the $TopN$ recommendations, as well as containing a dotted line representing the average recall rate for all analyzed systems.

Figure 4.3 – Precision graph of QMove for *Top3* to *TopN* recommendations



It shows that the first *n* recommendations have an average recall of 53.1%, with a standard deviation of 25.6%. Thus, considering the number of randomly moved methods for each system, the tendency is that 53.1% of them are recommended to return to their original classes with a 52.1% precision.

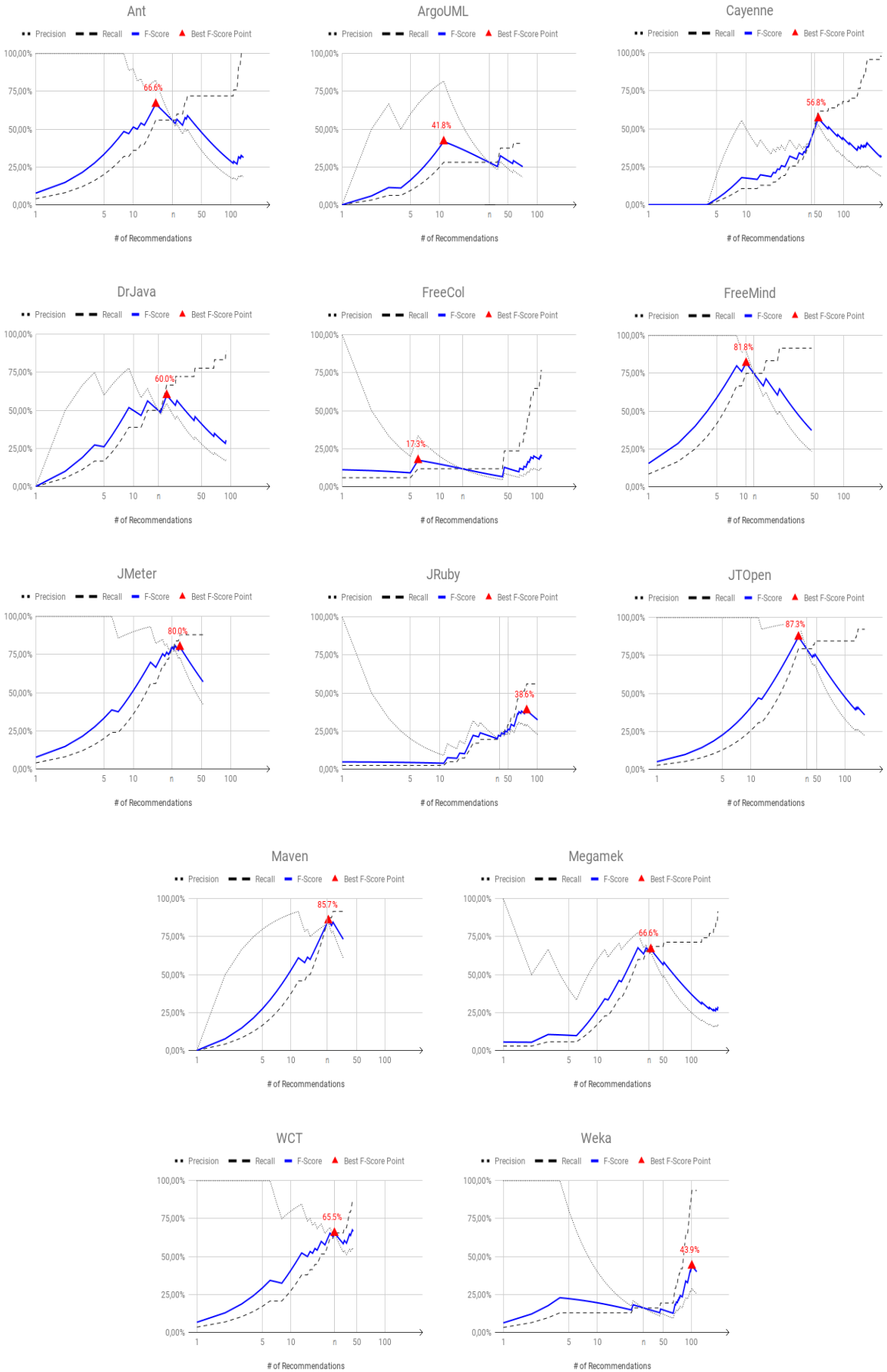
Figure 4.4 – Recall graph of QMove for *TopN* recommendations



Another strategy to verify the results is using f-score value in the same way that was used in the calibration (Section 3.3). Thereupon, we generate graphs for each of the 13 systems used in the evaluation, containing the precision, recall, and f-score behavior (Figure 4.5). All the graphs are in logarithmic scale, and the abscissa axis has the size of 250, to simplify the comparison between them.

Our goal is to observe the point that the f-score has its highest value, which means the highest values of precision and recall, before the f-score values begin to decline. This indicates

Figure 4.5 – Recall, precision, and f-score graph for each evaluated systems



the number of recommendations necessary for our approach to detect as many *Gold Set* methods as possible, while maintaining a high precision and recall rate.

By analyzing the graphs represented in Figure 4.5, we detected the points where f-score has its highest value, and for each point we extracted the corresponding precision, recall, and recommendation number. We also calculated the positions of the recommendations in function of *n*, where *n* is the size of the *Gold Set* for that system. Table 4.3 reports the results of this analysis showing for each system the recommendation position, the recommendation position in function of *n*, the f-score, precision, and recall values.

Table 4.3 – Best f-score value for each subject system of evaluation

System	Rec. Pos.	Rec(n)	F-Score	Prec.	Recall
Ant	17th	0.68	**66.6%**	82.3%	56.0%
ArgoUML	11th	0.34	**41.8%**	81.8%	28.1%
Cayenne	55th	1.17	**56.8%**	52.7%	61.7%
DrJava	22nd	1.22	**60.0%**	54.5%	66.6%
FreeCol	6th	0.35	**17.3%**	33.3%	11.7%
FreeMind	10th	0.83	**81.8%**	90.0%	75.0%
JMeter	30th	1.2	**80.0%**	73.3%	88.0%
JRuby	78th	1.90	**38.6%**	29.4%	56.1%
JTOpen	32nd	0.82	**87.3%**	96.8%	79.4%
Maven	25th	1.04	**85.7%**	84.0%	87.5%
Megamek	37th	1.05	**66.6%**	64.8%	68.5%
WCT	29th	1	**65.5%**	65.5%	65.5%
Weka	101th	3.25	**43.9%**	28.7%	93.5%
Average	**35th**	**1.14**	**62.6%**	**64.4%**	**64.5%**

As can be seen, the extracted data resulted in an average of 62.6% of the f-score values, with a standard deviation of 20.93%. Consequently, to find the number of recommendations needed to have the highest precision and recall values of 64.4% and 64.5%, respectively, are 1.14 *x n*. Therefore, considering all the systems used in the evaluation, our approach is able to detect, among the first 1.14 *x n Move Method* recommendations, 64.5% of methods contained in the *Gold Set* with 64.4% precision.

4.2 Comparative evaluation

We perform a comparative analysis between our approach and the JMove and JDeodorant tools. We used the same systems used in our synthesized evaluation (refer to Table 4.1). In this section, we ran JMove and JDeodorant on these 13 systems to verify if they recommend moving back the methods from our *Gold Set*.

Table 4.4 reports the comparative results for QMove, JMove, and JDeodorant (JDeo). It is shown the *Gold Set* size of evaluated systems, the number of refactoring recommendations (Recs.), the number of refactorings related to methods from the *Gold Set* (Recs. GS), and the total summation of each these mentioned values.

Table 4.4 – Comparative between QMove, JMove, and JDeodorant recommendations

		Recs.			Recs. GS		
System	**GS Size**	**QMove**	**JMove**	**JDeo**	**QMove**	**JMove**	**JDeo**
Ant	25	135	139	63	25	0	6
ArgoUML	32	71	48	9	13	6	4
Cayenne	47	245	158	199	46	11	11
DrJava	18	90	106	50	16	5	7
FreeCol	17	112	194	164	13	13	3
FreeMind	12	47	54	39	11	3	2
JMeter	25	52	59	46	22	2	3
JRuby	41	101	574	206	23	25	12
JTOpen	39	162	140	87	36	17	22
Maven	24	36	85	40	22	11	6
Megamek	35	193	244	131	32	9	18
WCT	29	46	44	51	25	5	12
Weka	31	114	246	279	29	6	11
Total	**375**	**1404**	**2091**	**1364**	**313**	**113**	**117**

As we can see in Table 4.4, QMove recommended 313 of 375 *Gold Set* methods, more than those recommended by JMove (113) and JDeodorant (117). In addition, QMove recommended a total of $1,404$ refactorings, while JMove and JDeodorant recommended $2,091$ and $1,364$, respectively.

By comparing the results for each evaluated system, QMove recommended more *Gold Set* methods in 11 of 13 systems, except FreeCol, where both QMove and JMove recommended 13 *Gold Set* methods, and JRuby, where JMove and QMove recommended 25 and 23 *Gold Set* methods, respectively. However, in both systems, JMove recommended a higher total number of refactorings for FreeCol (194) and JRuby (574), while QMove recommended 112 for FreeCol and 101 for JRuby.

Since Table 4.4 has absolute values on the results obtained, we performed a complementary analysis about the relation of the total recommended methods and the *Gold Set* methods. Therefore, Table 4.5 reports the recall, precision, and f-score values calculated for each evaluated system analyzed by QMove, JMove, and JDeodorant.

By comparing the average values reported in Table 4.5, QMove performed better than the other tools for all metrics. QMove f-score was 39.1%, which is more than twice as JDeodorant

Table 4.5 – Recall, precision, and f-score values for QMove, JMove, and JDeodorant tools

System	Recall			Precision			F-score		
	QMove	JMove	JDeo	QMove	JMove	JDeo	QMove	JMove	JDeo
Ant	100.0%	0.0%	24.0%	18.5%	0.0%	9.5%	31.2%	0.0%	13.6%
ArgoUML	40.6%	18.7%	12.5%	18.3%	12.5%	44.4%	25.2%	15.0%	19.5%
Cayenne	97.8%	23.4%	23.4%	18.7%	6.9%	5.5%	31.5%	10.7%	8.9%
DrJava	88.8%	27.7%	38.8%	17.7%	4.7%	14.0%	29.6%	8.0%	20.5%
FreeCol	76.4%	76.4%	17.6%	11.6%	6.7%	1.8%	20.1%	12.3%	3.3%
FreeMind	91.6%	25.0%	16.6%	23.4%	5.5%	5.1%	37.2%	9.0%	7.8%
JMeter	88.0%	8.0%	12.0%	42.3%	3.3%	6.5%	57.1%	4.7%	8.4%
JRuby	56.1%	60.9%	29.2%	22.7%	4.3%	5.8%	32.3%	8.1%	9.7%
JTOpen	92.3%	43.5%	56.4%	22.2%	12.1%	25.2%	35.8%	18.9%	34.9%
Maven	91.6%	45.8%	25.0%	61.1%	12.9%	15.0%	73.3%	20.1%	18.7%
Megamek	91.4%	25.7%	51.4%	16.5%	3.6%	13.7%	28.0%	6.4%	21.6%
WCT	86.2%	17.2%	41.3%	54.3%	11.3%	23.5%	66.6%	13.7%	30.0%
Weka	93.5%	19.3%	35.4%	25.4%	2.4%	3.9%	40.0%	4.3%	7.1%
Average	**84.2%**	**30.1%**	**29.5%**	**27.1%**	**6.6%**	**13.4%**	**39.1%**	**10.1%**	**15.7%**

(15.7%) and almost four times as JMove (10.1%). Considering precision and recall, QMove also performed at least twice as much as the other tools, with 84.2% and 27.1% of recall and precision, respectively, while JMove had 30.1% of recall and 6.6% of precision, and JDeodorant had 29.5% and 13.4% of recall and precision, respectively.

By analyzing the results of Table 4.5 for each system individually, QMove obtained better recall, precision, and f-score values on most systems. However, for FreeCol and JRuby systems, we had some different results. We can note that QMove and JMove obtained the same recall value for FreeCol system (76.4%), but the precision value was 11.6% for QMove and 6.7% for JMove, resulting in a better f-score value for QMove (20.1%) than JMove (12.3%). For JRuby, JMove had a better recall value (60.9%) compared to QMove (56.1%), but precision and f-score values were better for QMove (22.7% and 32.3%, respectively) than for JMove (4.3% and 8.1%, respectively). Therefore, in both cases where JMove found the same amount or more *Gold Set* methods, QMove achieved greater precision in its results.

Figure 4.6 shows the overlap of the Move Method recommendations belonging to the *Gold Set* of the three evaluated tools (QMove, JMove, and JDeodorant) for each of the 13 evaluated systems. We can see that in 12 systems QMove obtained a greater number of exclusive recommendations than the exclusive ones of JMove and JDeodorant, even in JRuby system, where QMove had a lower number of *Gold Set* methods recommended, but exclusively recommended 2 more than JMove. Only in FreeCol system, QMove did not have the most exclusive recommendations (2), with JMove having the largest number (3).

Figure 4.6 – Overlapping between results of each evaluated systems

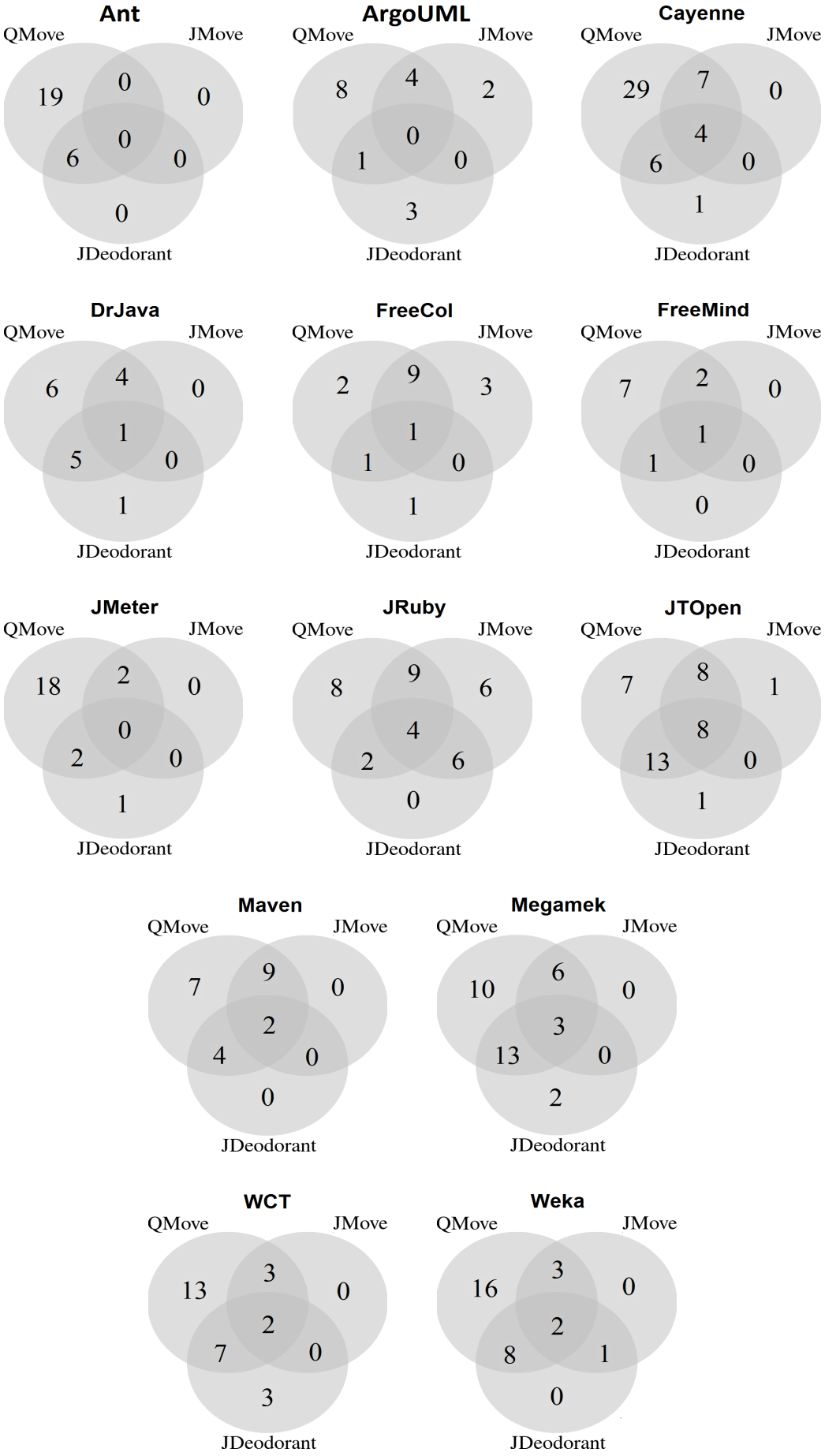
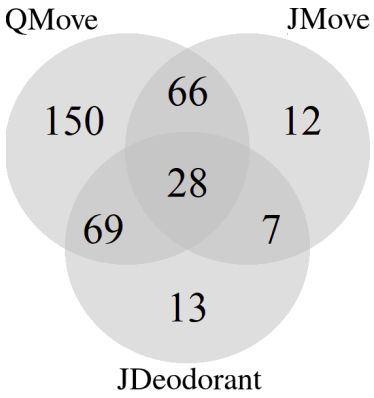


Figure 4.7 presents the overlap between the total number of *Gold Set* methods recommended by QMove, JMove, and JDeodorant, considering all 13 systems evaluated. QMove exclusively recommended 150 *Gold Set* methods, while JMove and JDeodorant exclusively recommended 12 and 13, respectively. QMove and JMove together recommended 66 methods, and QMove and JDeodorant recommended 69 *Gold Set* methods. The three tools together recommended 28 methods, and the refactorings that JMove and JDeodorant both recommended were only seven. We argue that the techniques are complementary since the state-of-the-art tools could indicate 32 correct recommendations that QMove could not.

Figure 4.7 – Overlapping between results of all the evaluated systems



We could use other evaluation metrics such as feedback, which is the ratio of all recommendations with the number of methods that can move automatically, and likelihood, which checks whether there are useful refactorings among the recommendations, i.e., *Gold Set* methods included in a k number of recommendations. Whereas likelihood is important for providing an indication of how often QMove provides at least one potentially useful result, feedback is a useful metric for recommendation systems, because a recommender that rarely gives recommendations may not be practical (ROCHA et al., 2016). However, we decided to not use these evaluation metrics since (i) precision and recall are more accurate metrics and obtained positive results, (ii) we did not extracted from results the number of methods that could move automatically, and (iii) likelihood in real scenario evaluation could not be useful since the objective was to verify if the recommendations were well evaluated for expert developers.

4.3 Real scenario evaluation

In order to evaluate our approach in a real scenario, we performed an analysis through the use of QMove in two proprietary systems developed by a IT company located in Lavras, Mi-

nas Gerais, Brazil. Table 4.6 reports data about these systems.[2] ReMent is a demand management system and Cyssion a concession management system. We chose these systems because they are in an advanced phase of implementation, have well-defined architectures through the MVC (Model-View-Controller) model, and are developed in Java.

Table 4.6 – Proprietary systems in the real scenario evaluation

System	# of classes	# of methods	LOC
ReMent	140	222	7,484
Cyssion	216	574	16,021

Besides executing QMove, we also executed JMove and JDeodorant on these systems in order to compare the results. Two expert developers (one for each system) evaluated recommendations provided by each tool. Using the Likert scale (JAMIESON, 2004), the evaluation methodology consisted of developers answering, for each recommendation, the question "How do you rank this Move Method recommendation?". The answer to this question could be one of five available: (1) Strongly not recommended, (2) Not recommended, (3) Neither recommended nor recommended, (4) Recommended, or (5) Strongly recommended. We leave free the option of the experts to comment on the reasons for the chosen option, thus gathering useful information that could contribute to our evaluation.

For ReMent, JMove and JDeodorant gave no recommendations. QMove, on the other hand, recommended five methods for ReMent, where two were evaluated as “Neither recommended nor recommended” and three as “Strongly not recommended”.

For Cyssion, Table 4.7 reports the evaluation results of the 41, five, and six recommendations triggered by QMove, JMove, and JDeodorant, respectively.

Table 4.7 – Proprietary system Cyssion experts’ evaluation

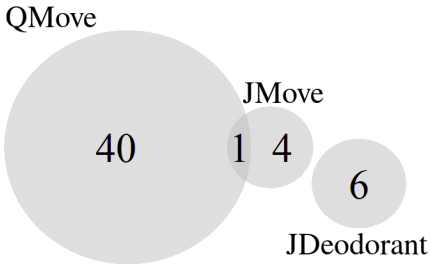
Rec. Classification	QMove	JMove	JDeodorant
(5) Strongly Recommended	2	0	0
(4) Recommended	4	2	0
(3) Neither Recommended Neither Not Recommended	2	1	1
(2) Not Recommended	6	0	0
(1) Strongly Not Recommended	27	2	5
Total	**41**	**5**	**6**

Considering the positive evaluations, which are those recommendations that were evaluated as (4) and (5), QMove had six, against two and zero of JMove and JDeodorant, respectively. Also, including as positive the recommendations that were evaluated as (3) because their

[2] We changed the names of the systems for confidentiality purposes.

neutrality, QMove would have eight recommendations against three from JMove and one from JDeodorant. Figure 4.8 shows the overlap between the recommendations found for all three tools, making it clear that only one recommendation was found at the same time by QMove and JMove, which is evaluated as (4).

Figure 4.8 – Overlapping between results of proprietary system Cyssion



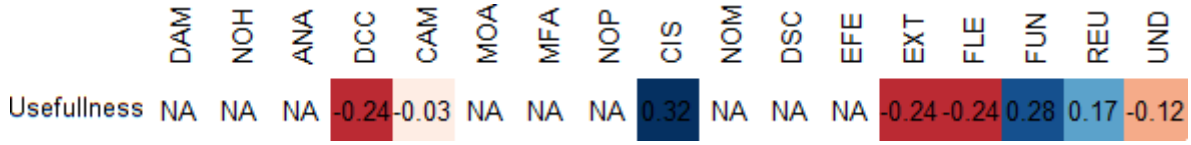
These results demonstrate that QMove was relatively more effective in finding practically useful recommendations. However, there were a high number of recommendations evaluated as (1) and (2) for the two systems, and the reasons for this can be explained by comments from the experts when we were conducting the evaluation.

First, most of the comments focused on justifying evaluations (1) and (2) because of the meaningless moves, such as moving an accessor method of a private attribute. Second, experts commented that some recommendations involved methods that were being overwritten from an interface (@override annotation), and moving them would cause compilation errors. Third and last, there were comments on methods used by frameworks, and moving them to other classes would hinder the functioning of the framework. These issues will be considered in our future work.

We performed an analysis of the correlation between the specialist rates and each QMOOD metric, i.e., the 11 design metrics (see Table 2.1) and the six quality attributes (see Table 2.2). We use Spearman's rank correlation coefficient, a nonparametric (distribution-free) rank statistic proposed as a measure of the strength of the association between two variables (HAUKE; KOSSOWSKI, 2011). The coefficient varies between -1 to 1, and the closer to zero, the lower the correlation between the variables. A coefficient greater than zero means that the two variables correlate as they grow at the same time, and a coefficient less than zero means a correlation in which as one variable grows, the other one decreases.

Figure 4.9 reports the Spearman's coefficients for each QMOOD metric, with positive correlations displayed in blue and negative correlations in red color. Color intensity are proportional to the correlation coefficients, and a non-correlation is represented by NA (Not Available).

Figure 4.9 – Correlation between proprietary system specialist rates and QMOOD metrics



We found five weak correlations between the rated recommendations and the DCC (-0.24), Flexibility (-0.24), and Extensibility (-0.24) metrics, which decrease while the recommendation rating increases, and CIS (0.32) and Functionality (0.28), which increase at the same time as the recommendation rating increases. We also find three very weak correlations between the rated recommendations and the CAM (-0.03), Reusability (0.17), and Understandability (-0.12). In a nutshell, we could not find any relevant correlation between the usefulness of a particular recommendation and any underlying metric from QMOOD model.

Concluding this evaluation, QMove was able to find positively evaluated methods in greater quantity than JMove and JDeodorant, and the number of negatively evaluated recommendations can be reduced with adjustments in the preconditions of QMove.

4.4 Threats to validity

We found two threats to validity and we divided them into an internal and an external type, and we also discuss the strategies used to mitigate these threats.

Internal Validity: We modified the subject systems to evaluate our approach by randomly moving methods from one class to another in order to verify whether these methods are recommended to return to their original classes or not. This fully-random methodology implies in the possibility of a moved method improves QMOOD quality attributes in its new class. In this case, our approach would not recommend such a method since returning to its original class would worsen our fitness function. However, since our approach achieved a recall rate of 84.2%, we can *at least* assume that most methods worsen quality metrics when they were moved.

External Validity: The subject systems used in the calibration and in the evaluation are implemented in Java. One could argue that this could affect the use of our proposed approach in other systems that use different programming languages. Nevertheless, our fitness function is based on the QMOOD model, whose quality metrics can be measured for any object-oriented project, regardless of the underlying programming language.

5 RELATED WORK

In this chapter, we discuss studies that are closely related to ours. Section 5.1 discusses approaches that search for a refactoring sequence that maximizes QMOOD metric values. Section 5.2 discusses approaches that search for a sequence of refactorings that maximizes or minimizes values of other metrics found in the literature or created by the in own authors. Section 5.3 discusses approaches that focus on suggestions for refactorings using other strategies regarding the use of metrics during the process. Finally, Section 5.4 discusses studies on the impact of refactorings on a certain set of metrics.

5.1 Refactorings and QMOOD quality attributes

Mkaouer et al. propose a solution that suggests a sequence of refactorings that optimize QMOOD quality attributes (MKAOUER et al., 2016). For this, the multi-objective genetic algorithm of optimization NSGA-III is used in order to find the sequence of refactorings that optimize the desired attributes. The algorithm considers possible refactor sequences and positive or negative weight of each type of refactoring on each QMOOD quality attribute. The suggested solution, in contrast to the approach proposed in this dissertation, does not have its own implementation (JDeodorant was used for refactoring) and, because it uses a genetic algorithm, it is possible to generate a non-optimal sequence of refactorings.

Moghadam and Cinnéide present Code-Imp, an automatic refactoring tool for Java language (MOGHADAM; CINNÉIDE, 2011). It supports fourteen refactorings, separated at the method level (e.g., Push Down Method, Pull Up Method, etc.), at the attribute level (e.g., Push Down Field, Pull Up Field, etc.) and at the class level (e.g., Extract Hierarchy, Make Superclass Abstract, etc.). With support for various types of research, Code-Imp implements 28 software quality metrics (e.g., QMOOD Quality Attributes flexibility, reusability and understandability, cohesion metrics, coupling metrics, interface class size, number of methods, etc.) that can be combined in various ways to form a fitness function. On its activities, Code-Imp first extracts the initial AST (Abstract Syntax Tree) from the source code. Then, it look for candidates for refactorings in AST. A refactoring is acceptable if it satisfies all pre and post-conditions, as well as meet the requirements of the search technique in use (e.g., improve project quality based on the total set of metrics, or with an acceptable fall in certain metrics). This process is repeated many times, and after the final refactoring is applied, the AST is included in the source files. Although it is a tool that works with several refactorings and metrics, its purpose is more fo-

cused on refactorings that perform corrections and adaptations in class hierarchies, thus limiting their use. However, the Move Method refactoring used in our proposed approach modifies the source code of the analyzed software system to solve problems in different contexts, as seen in Section 2.2.

Jensen and Cheng present REMODEL, an approach to refactor object-oriented software projects using a genetic algorithm to maximize QMOOD metrics, resulting in a set of refactoring recommendations for the user (JENSEN; CHENG, 2010). This approach has two objectives: (i) to improve the quality of the project by QMOOD metrics; and (ii) introduce design patterns, where appropriate, to improve the maintainability of the software project. The genetic algorithm used by REMODEL has as individuals pairs consisted of a graphical design (represents the design of the software being refactored) and a transformation tree (encoding a set of changes in graphical design of an individual), works with six different types of transformations—Abstraction, Abstract Access, Encapsulate Construction, Delegation, Partial Abstraction and Packer—and uses the 11 QMOOD design properties in the formulation of their fitness function. This approach works with types of refactorings different from those presented by Fowler in his book (FOWLER, 1999) and hence differ from the Move Method refactoring considered in our proposed approach.

Lee et al. propose an approach to suggest refactorings that remove duplicate codes while maximizing quality improvement in a software (LEE et al., 2011). Basically, the set of duplicate methods is first detected and a set of rules is proposed to determine the appropriate refactorings (Pull Up Method, Text Template Method, Move Method, Extract Class, Extract Superclass, and Extract Intermediate-Superclass) to the detected duplicate method sets. Subsequently, a genetic algorithm is applied to determine the most beneficial refactorings that maximize the fitness function determined by the sum of four QMOOD quality attributes (understandability, functionality, extensibility, and effectiveness). The scope of this approach is limited to duplicate methods, unlike the proposed approach in this dissertation that aims to verify all methods of a software system and to detect mainly those that are not correctly implemented in the class to which they belong, searching for a sequence of Move Method refactorings that maximize QMOOD quality attributes by moving the detected methods.

5.2 Refactorings and others metrics types

Terra et al. (TERRA et al., 2017) propose JMove, a tool for Move Method refactorings. The methodology for suggests refactorings is to first evaluate the set of static dependencies established by a given method m located in a class C. After that, it is computed two Jaccard similarity coefficients: (i) the average similarity between the set of dependencies established by method m and by the remaining methods in C; and (ii) the average similarity between the dependencies established by method m and by the methods in another class C'. If the similarity measured in the step (ii) is greater than the similarity measured in (i), it is inferred that m is more similar to the methods in C' than to the methods in its current class C. Therefore, C' is a potential candidate class to receive m. However, JMove deals with methods that have more than four dependencies, while our approach does not have this restriction, consequently increasing the scope of recommendation possibilities.

Tsantalis and Chatzigeorgiou (TSANTALIS; CHATZIGEORGIOU, 2009) present JDeodorant, a tool that suggests Move Method refactorings as solutions to the Feature Envy design problem. This approach defines some preconditions that preserve behavior and the design quality for recommendations, such as avoid compilation errors and assure behavior preservation. JDeodorant also defines a metric called Entity Placement that is used to evaluate whether a recommendation reduces coupling, defined by the Jaccard distance between the class itself and outer entities, and improves cohesion, defined by the Jaccard distance between the class itself and inner entities. JDeodorant is based only on cohesion and coupling metrics, while our approach, using the QMOOD quality model, provides a broad set of metrics that provides further improvements to the quality of the system.

Bavota el al. (BAVOTA et al., 2014) propose an approach named Methodbook to identify Move Method refactoring opportunities aimed at solving Feature Envy bad smell. The proposed approach follows the Facebook metaphor that analyzes users' profiles and suggests new friends or groups of people sharing similar interests using Relational Topic Model (RTM) (CHANG; BLEI, 2010). Thus, in Methodbook, methods and classes play the same role as people and groups of people, respectively, in Facebook. Methods' implementations, that is profiles, contain information about structural (e.g., method calls) and conceptual (i.e., textual) relationships (e.g., similar identifiers and comments) with other methods in the same class and in the other classes. Then, Methodbook uses RTM to identify "friends" of a method in order to suggest Move Method refactoring opportunities in software. In particular, given a method, Methodbook

exploits RTM to suggest as a target class the one containing the highest number of “friends” of the method under analysis. Methodbook uses in its approach a concept of methods recommendations that improves cohesion and coupling of a system, different from our approach that uses metrics with wider scope, aiming to improve several characteristics related to the quality of the software.

Napoli et al. propose an approach for suggestions of Move Method refactorings in large object-oriented systems, improving the modularity of several components at once by calculating metrics as Fan-in, Fan-out, LCOM, CBO, and the Jaccard similarity coefficient (NAPOLI; PAPPALARDO; TRAMONTANA, 2013). This approach uses different forms of metric combinations to suggest a Move Method refactoring. As it is focused on large systems, they adopted the parallel programming strategy to calculate the metrics through threads and hence improving the performance. Although the application of suggested refactorings indirectly increase cohesion and decrease coupling, its focus on using metrics to detect refactoring opportunities differs from our proposed approach, which uses QMOOD metrics to detect refactorings that directly increase the values of these metrics.

Meananeatra et al. propose an approach for the identification and selection of Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object, and Decompose Conditional refactorings that solve the long method problem (MEANANEATRA; RONGVIRIYAPANISH; APIWATTANAPONG, 2011). A method is proposed to select refactorings based on software metrics that are defined in terms of data flow and control flow charts. The method consists of four steps: (i) calculate the MCX (Complexity of Method), LOC and LCOM metrics to compare maintainability before and after refactorings; (ii) find candidates for refactoring using the previously measured metrics, (iii) apply each refactoring candidate and calculate the maintainability metrics later; and (iv) identify the refactoring that provides the greatest maintainability. The concept of using metrics to detect Code Smells while at the same time improving software quality features is limited to the concept of source code maintainability, whereas our proposed approach involves more concepts than maintainability, provided by QMOOD quality attributes, e.g., reusability, flexibility, understandability, etc.

Lee and Wu describe an approach of automatic restructuring using metrics that preserve the behavior of object-oriented projects (LEE; WU, 2001). They defined cohesion and coupling metrics to quantify the projects and to provide criteria for comparing alternative designs, and also defined eight types of primitive restructures in CUG (Call-Use Graph) and CAG (Class-

Association Graph), e.g., changing the name of a node in a CUG, changing a name of a node in a CAG, move a member of a class, etc. The authors implemented the approach as a tool for Java projects. This tool has a converter that extracts information from the analyzed source code and generates a CMM (Communication Matrix between Methods). Subsequently, a metric analyzer measures the values of the cohesion and coupling metrics. A genetic algorithm uses primitive restructurings to decompose or compose classes during the restructuring process, using metric values to improve design. In addition, a verifier model examines whether the project produced satisfies design constraints in each generation. This is an approach that automates the metric-based restructuring process, but if developers do not provide information about the context of the object-oriented system being parsed, the restructuring becomes less advantageous. Another weak point is the scope of the restructuring to be focused on cohesion and coupling metrics, whereas our proposed approach has focus on QMOOD metrics, which have a broader scope in relation to the quality of software systems.

5.3 Refactorings and different uses of metrics

Griffith et al. propose an approach based on machine learning and metric algorithms to automate refactorings (GRIFFITH; WAHL; IZURIETA, 2011). The goal of the approach is to modify object-oriented legacy systems to increase understandability, maintainability, and reusability. The approach has been implemented and named as TrueRefactor. This tool detects Code Smells by combining CK metrics (Chidamber-Kemerer), used for object-oriented programs, and size-oriented metrics. For each detected Code Smell, a refactoring sequence is generated. Thereafter, a genetic algorithm is applied to find the best refactoring sequence that removes as many Code Smells as possible. Based on the best set of refactoring sequences, a UML class diagram is produced. In short, this approach uses metrics to detect Code Smells and, by applying the best refactoring sequence detected, aims to optimize these metrics. In contrast, refactorings are used to create only a class diagram with code modifications, with the user having the responsibility to manually apply those refactorings. However, our proposed approach automatically applies the recommended refactorings on the system.

Hotta et al. propose CRat, a refactoring support tool for the Template Method design pattern (HOTTA et al., 2012). In this pattern, programmers create a (abstract) base class with similar methods and implement the detailed processes in each (concrete) derived class. CRat detects pairs of methods with duplicate codes in derived classes to be merged and moved to

the base class. For each pair of methods that can be refactored, a set of metrics are calculated, such as degree of similarity, size, number of statements that can be extracted for the base class, depth of inheritance of the common base class for the proprietary classes of the two methods, etc. CRat users can therefore filter out possible refactorings by the metric values they want. However, CRat does not modify the source code automatically, so users need to modify the source code manually. In addition, metrics only assist users to choose refactorings that have obtained the best values from them. Our proposed approach provides the option to automatically apply the suggested refactorings, as well as to use the QMOOD metrics to generate the sequence of refactorings that improve metric values in all refactorings.

Marinescu et al. (MARINESCU et al., 2005; MARINESCU, 2004) propose iPlasma, a tool that helps developers and maintainers to detect and localize design problems in a system. This tool uses a mechanism, called detection strategy, for formulating metrics-based rules that capture deviations from good design principles and heuristics, including the detection of methods displaying a Feature Envy behavior. The metrics can be divided into the following categories: (i) size metrics; (ii) complexity metrics; (iii) coupling metrics; and (iv) cohesion metrics. Our approach, in addition to these categories, provides support for other types of features through QMOOD metrics, which cover various aspects of software quality.

5.4 Impact of refactorings on metrics

Chaparro et al. propose an approach to predict the impact of 12 types of refactorings on 11 metrics (CHAPARRO et al., 2014). The objective of the approach is to show the variation of the metrics if a certain refactoring is applied. The 12 refactorings involved are separated into groups of different contexts, such as moving components (e.g., Move Method and Move Field), composition of methods (e.g., Extract Method), and generalization (e.g., Pull Up Method). The 11 metrics measure code properties such as coupling (RFC), size (LOC), complexity (CC), and inheritance (DIT). For each refactor-metric pair, a mathematical function was defined for the class codes involved in the refactoring and a function for the target class, when appropriate. Each function results in a value that indicates whether refactoring will impact positively or negatively on the analyzed metric. The purpose of the approach, therefore, is to use metrics only to predict the impact on their values if a refactoring is applied, whereas our proposed approach accurately calculates the variations in metric values by simulating method moves for each refactoring found in the analyzed system.

Higo et al. propose a method to estimate the effect of refactorings on source code based on complexity metrics (HIGO et al., 2008). The proposed method receives the system's source code and the refactorings to be applied. Subsequently, the effectiveness of each refactoring is reported based on the CK metrics calculated for the original and modified source code. This method was implemented as a tool for Java language. There are eight tool-supported refactorings (Pull Up/Down Field, Move Method, Pull Up/Down Method, Extract Class, and Extract Super/Subclass). The tool receives the Java source code and calculates the CK metrics; later, the developer defines the refactorings that will be performed. The structure of the source code is modified from the refactorings defined by the developer. The CK metrics are recalculated in the modified source code, and the tool returns the values of the metrics before and after the changes. Although this method is based on metrics to estimate refactorings that improve their values, the refactoring process is non-existent. It is up to the developer to inform which refactorings will be applied in the source code, whereas our proposed approach automatically detect the refactorings and apply them to the source code of the analyzed system.

6 CONCLUSION

Refactoring is an important activity to improve software internal structure. Even though there are many refactoring approaches, very few consider their impact on the software quality. In this dissertation, thereupon, we proposed a search-based approach to recommend Move Method refactorings that improve QMOOD quality attributes. QMove receive as input a given software system S and recommends a sequence of refactorings $R_1, R_2, \ldots, R_n$ that result in system versions $S_1, S_2, \ldots, S_n$, where the sum of all six QMOOD quality attributes is greater in S_{i+1} that in S_i.

We empirically calibrated our approach to find the best criteria to assess software quality improvement. First, we modified four systems by randomly moving a subset of its methods to other classes. Second, we verified which of the ten different strategies would recommend more of the moved methods to return to their original place. As the result, we calibrate our approach with the strategy of comparing the metrics by improvement percentage of the sum of QMOOD quality attributes, which achieved a recall average of 57.5%.

We also implemented QMove, a prototype plug-in for Eclipse IDE that supports our proposed refactoring approach with our current calibration. The plug-in receives as input a Java system and outputs the better sequence of Move Method refactorings that improves the overall software quality.

In our first experiment, we evaluated our approach in 13 open-source systems by randomly moving 375 methods. On average, our approach could move 84.2% of the methods back to their original classes. More important, our approach is able to detect, among the first $1.14 \times n$ recommendations (where n is the size of the *Gold Set* for each system), 64.5% of methods contained in the *Gold Set* with 64.4% precision, on average.

In our second experiment, we compared QMove with JMove and JDeodorant on the same 13 systems used in the first evaluation. As result, the state-of-the-art tools showed lower precision, recall, and hence f-score values than the ones achieved by QMove. While QMove recall value was 84.2%, JMove and JDeodorant recall values were 30.1% and 29.5%, respectively. Moreover, QMove recommended more *Gold Set* methods than JMove and JDeodorant in 11 of 13 evaluated systems. Nevertheless, we argue that the techniques are complementary since the state-of-the-art tools could indicate 32 correct recommendations that QMove could not.

In our third and last experiment, we evaluated QMove, JMove, and JDeodorant in a real scenario on the eyes of the software architects. As result, the software architects positively

evaluated six out of 46 recommendations from QMove, two out of five from JMove, and none out of six from JDeodorant. Although QMove found more correct Move Method opportunities, it triggered much more false positives than the state-of-the-art tools.

6.1 Contributions

This dissertation contains the following contributions:

- A new quality-oriented approach that identifies Move Method refactoring opportunities in software systems, whose purpose is to support refactoring process by verifying all possible automatic Move Method possibilities and recommending those that improve the quality of the system by the six quality attributes from QMOOD model;
- A tool called QMove that identifies Move Method refactorings that can be automatically applied and detects those that improve QMOOD quality metrics, returning to the user the best refactoring sequence in order to achieve the best system overall quality; and
- Evaluations that compared our approach with other similar state-of-the-art approaches, having better results in both synthesized and real scenario analysis, thus contributing to the academic community in the advancement of state-of-the-art.

6.2 Limitations

Next, we list the limitations in this dissertation:

- Our approach considers only refactorings performed automatically;
- Among the existing refactorings in the literature, we consider only Move Method refactoring. Although it is one of the most common refactorings, there are other automatic refactorings that we do not consider it in our approach; and
- The results obtained in our experiments are inserted in the context of Java systems. Therefore, although our proposed approach is suitable for any object-oriented programming language, we cannot extrapolate our results for other programming languages.

6.3 Future work

Future work to complement our proposed approach includes:

- To incorporate other types of refactorings, such as Extract Class and Extract Method. Increasing the number of refactoring types and applying then in a system can generate a new system with higher quality than a system version with one type of refactoring applied;
- To improve QMove preconditions to avoid false positives, in order to increase the precision of the recommended refactorings and to obtain better results in further evaluations of our proposed approach;
- To include other evaluation metrics used for recommendation systems, such as likelihood, recall rate@k, and feedback. The use of these new forms of evaluation can provide results that better reflect the use of refactorings in real scenarios, which may influence the choice of our proposed approach by software developers; and
- To rely on other metrics—such as number of bad smells (FOWLER, 1999), CK (CHIDAMBER; KEMERER, 1994) and Martin's metrics (MARTIN, 1994)—and other search-based algorithms—such as the multi-objective NSGA-II (DEB et al., 2002) and SPEA2 (ZITZLER; LAUMANNS; THIELE, 2001)—to allow the users to set up their own fitness function and the underlying search algorithm.

REFERENCES

BANSIYA, J.; DAVIS, C. G. A hierarchical model for object-oriented design quality assessment. **IEEE Transactions on Software Engineering**, v. 28, n. 1, p. 4–17, 2002.

BAVOTA, G.; OLIVETO, R.; GETHERS, M.; POSHYVANYK, D.; LUCIA, A. D. Methodbook: Recommending Move Method refactorings via relational topic models. **IEEE Transactions on Software Engineering**, v. 40, n. 7, p. 671–694, 2014.

BOIS, B. D.; DEMEYER, S.; VERELST, J. Refactoring - improving coupling and cohesion of existing code. In: **11th Working Conference on Reverse Engineering (WCRE)**. Delft, Netherlands: IEEE, 2004. p. 144–151.

CHANG, J.; BLEI, D. M. Hierarchical relational models for document networks. **The Annals of Applied Statistics**, v. 4, n. 1, p. 124–150, 2010.

CHAPARRO, O.; BAVOTA, G.; MARCUS, A.; PENTA, M. D. On the impact of refactoring operations on code quality metrics. In: **30th International Conference on Software Maintenance and Evolution (ICSME)**. Victoria, Canada: CPS, 2014. p. 456–460.

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

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: A taxonomy. **IEEE Software**, v. 7, n. 1, p. 13–17, 1990.

DEB, K.; PRATAP, A.; AGARWAL, S.; MEYARIVAN, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. **IEEE transactions on evolutionary computation**, v. 6, n. 2, p. 182–197, 2002.

FAWCETT, T. An introduction to ROC analysis. **Pattern recognition letters**, v. 27, n. 8, p. 861–874, 2006.

FOWLER, M. **Refactoring: improving the design of existing code**. Boston, USA: Addison-Wesley, 1999.

GOUTTE, C.; GAUSSIER, E. A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In: **27th European Conference on Information Retrieval (ECIR)**. Santiago de Compostela, Spain: ACM, 2005. p. 345–359.

GRIFFITH, I.; WAHL, S.; IZURIETA, C. TrueRefactor: An automated refactoring tool to improve legacy system and application comprehensibility. In: **24th International Conference on Computer Applications in Industry and Engineering (CAINE)**. Honolulu, USA: ISCA, 2011. p. 316–321.

HAUKE, J.; KOSSOWSKI, T. Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data. **Quaestiones geographicae**, v. 30, n. 2, p. 87–93, 2011.

HEVNER, A. R. Phase containment metrics for software quality improvement. **Information and Software Technology**, v. 39, n. 13, p. 867–877, 1997.

HIGO, Y.; MATSUMOTO, Y.; KUSUMOTO, S.; INOUE, K. Refactoring effect estimation based on complexity metrics. In: **19th Australian Software Engineering Conference (ASWEC)**. Perth, Australia: IEEE, 2008. p. 219–228.

HOTTA, K.; HIGO, Y.; IGAKI, H.; KUSUMOTO, S. CRat: A refactoring support tool for form template method. In: **20th International Conference on Program Comprehension (ICPC)**. Passau, Germany: IEEE, 2012. p. 250–252.

JAMIESON, S. Likert scales: how to (ab) use them. **Medical Education**, v. 38, n. 12, p. 1217–1218, 2004.

JENSEN, A. C.; CHENG, B. H. On the use of genetic programming for automated refactoring and the introduction of design patterns. In: **12th Genetic and Evolutionary Computation Conference (GECCO)**. Portland, USA: ACM, 2010. p. 1341–1348.

KATOCH, B.; SHAH, L. K. A systematic analysis on MOOD and QMOOD metrics. **International Journal of Current Engineering and Technology**, v. 4, n. 2, p. 620–622, 2014.

KEBIR, S.; BORNE, I.; MESLATI, D. Automatic refactoring of component-based software by detecting and eliminating bad smells. In: **11th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE**. Rome, Italy: SCITEPRESS, 2016. p. 210–215.

KESSENTINI, M.; KESSENTINI, W.; SAHRAOUI, H.; BOUKADOUM, M.; OUNI, A. Design defects detection and correction by example. In: **19th International Conference on Program Comprehension (ICPC)**. Kingston, Canada: IEEE, 2011. p. 81–90.

KIM, M.; ZIMMERMANN, T.; NAGAPPAN, N. A field study of refactoring challenges and benefits. In: **20th International Symposium on the Foundations of Software Engineering (FSE)**. Cary, USA: ACM, 2012. p. 1–11.

LEE, B.; WU, C. An automatic restructuring approach preserving the behavior of object-oriented designs. In: **8th Asia Pacific Software Engineering Conference (APSEC)**. Macau, China: IEEE, 2001. p. 400–407.

LEE, S.; BAE, G.; CHAE, H. S.; BAE, D.-H.; KWON, Y. R. Automated scheduling for clone-based refactoring using a competent GA. **Software: Practice and Experience**, v. 41, n. 5, p. 521–550, 2011.

MARIANI, T.; VERGILIO, S. R. A systematic review on search-based refactoring. **Information and Software Technology**, v. 83, p. 14–34, 2017.

MARINESCU, C.; MARINESCU, R.; MIHANCEA, P. F.; WETTEL, R. iPlasma: An integrated platform for quality assessment of object-oriented design. In: **21th International Conference on Software Maintenance (ICSM)**. Budapest, Hungary: IEEE, 2005. p. 77–80.

MARINESCU, R. Detection strategies: metrics-based rules for detecting design flaws. In: **20th International Conference on Software Maintenance (ICSM)**. Chicago, USA: IEEE, 2004. p. 350–359.

MARTIN, R. OO design quality metrics – an analysis of dependencies. In: **9th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)**. Portland, USA: ACM, 1994. p. 151–170.

MEANANEATRA, P.; RONGVIRIYAPANISH, S.; APIWATTANAPONG, T. Using software metrics to select refactoring for long method bad smell. In: **8th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications, and Information Technology (ECTI-CON)**. Khon Kaen, Thailand: IEEE, 2011. p. 492–495.

MENS, T.; DEMEYER, S.; BOIS, B. D.; STENTEN, H.; GORP, P. V. Refactoring: Current research and future trends. **Electronic Notes in Theoretical Computer Science**, v. 82, n. 3, p. 483–499, 2003.

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

MKAOUER, M. W.; KESSENTINI, M.; BECHIKH, S.; CINNÉIDE, M. Ó.; DEB, K. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. **Empirical Software Engineering**, v. 21, n. 6, p. 2503–2545, 2016.

MOGHADAM, I. H.; CINNÉIDE, M. O. Code-Imp: A tool for automated search-based refactoring. In: **4th Workshop on Refactoring Tools (WRT)**. Waikiki, USA: ACM, 2011. p. 41–44.

NAPOLI, C.; PAPPALARDO, G.; TRAMONTANA, E. Using modularity metrics to assist Move Method refactoring of large systems. In: **7th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)**. Taichung, Taiwan: CPS, 2013. p. 529–534.

OLSON, D. L.; DELEN, D. **Advanced data mining techniques**. Heidelberg, Germany: Springer Science & Business Media, 2008.

ROCHA, H.; VALENTE, M. T.; MARQUES-NETO, H.; MURPHY, G. C. An empirical study on recommendations of similar bugs. In: **23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. Osaka, Japan: IEEE, 2016. p. 46–56.

SHATNAWI, R.; LI, W. An empirical assessment of refactoring impact on software quality using a hierarchical quality model. **International Journal of Software Engineering and its Applications**, v. 5, n. 4, p. 127–149, 2011.

STROULIA, E.; KAPOOR, R. Metrics of refactoring-based development: An experience report. In: **7th International Conference on Object-Oriented Information Systems (OOIS)**. Calgary, Canada: Springer, 2001. p. 113–122.

TERRA, R.; VALENTE, M. T.; ANQUETIL, N. A lightweight remodularization process based on structural similarity. In: **10th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)**. Maringá, Brazil: IEEE, 2016. p. 111–120.

TERRA, R.; VALENTE, M. T.; MIRANDA, S.; SALES, V. JMove: A novel heuristic and tool to detect move method refactoring opportunities. **Journal of Systems and Software**, v. 138, p. 19–36, 2017.

TSANTALIS, N.; CHATZIGEORGIOU, A. Identification of Move Method refactoring opportunities. **IEEE Transactions on Software Engineering**, v. 35, n. 3, p. 347–367, 2009.

WANG, H.; KESSENTINI, M.; GROSKY, W.; MEDDEB, H. On the use of time series and search based software engineering for refactoring recommendation. In: **7th International Conference on Management of Computational and Collective Intelligence in Digital Ecosystems (MEDES)**. Caraguatatuba, Brazil: ACM, 2015. p. 35–42.

ZITZLER, E.; LAUMANNS, M.; THIELE, L. SPEA2: Improving the strength Pareto evolutionary algorithm. **TIK-report**, v. 103, p. 1–21, 2001.