A Quality-oriented Approach to Recommend Move Method Refactorings

Christian Marlon Souza Couto
Department of Computer Science
Federal University of Lavras (UFLA)
Lavras, Brazil
christiancouto@posgrad.ufla.br

Henrique Rocha
Inria Lille - Nord Europe
Lille, France
henrique.rocha@inria.fr

Ricardo Terra
Department of Computer Science
Federal University of Lavras (UFLA)
Lavras, Brazil
terra@dcc.ufla.br

ABSTRACT

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 paper, we propose a software refactoring approach based on quality attributes. We rely on the measurements of the Quality Model for Object Oriented Design (QMOOD) to recommend Move Method refactorings that improve software quality. In a nutshell, 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 systems, to find the best criteria to measure the quality improvement. We performed three types of evaluation to verify the usefulness of our implemented tool, named QMove. First, we applied our approach on 13 open-source systems achieving an average recall of 84.2%. Second, we compared QMove with two state-of-art refactoring tools (JMove and JDeodorant) on the 13 previously evaluated systems, and QMove showed better recall, precision, and f-score values than the others. Third, we evaluated QMove, JMove, and JDeodorant in a real scenario with two proprietary systems on the eyes of their software architects. As result, the experts positively evaluated a greater number of QMove recommendations.

CCS CONCEPTS

• **Software Architecture → Refactoring**; • **Quality Metrics →** *QMOOD Quality Model*;

KEYWORDS

Software Architecture; Refactoring; Move Method; Quality Metrics.

ACM Reference Format:
Christian Marlon Souza Couto, Henrique Rocha, and Ricardo Terra. 2018. A Quality-oriented Approach to Recommend Move Method Refactorings. In *17th Brazilian Symposium on Software Quality (SBQS), October 17–19, 2018, Curitiba, Brazil (Pre-print Author version)*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3275245.3275247

1 INTRODUCTION

The refactoring process changes the code to improve the internal structure without compromising its external behavior [8]. Currently, there are many refactoring approaches where the degree of automation can vary [4, 13, 20, 21]. Nevertheless, there are very few that consider their impact on software quality metrics. Consequently, a software system may be refactored into a version that worsens its overall quality.

In this paper, 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 Quality Model for Object Oriented Design (QMOOD) [3] to recommend Move Method refactorings that improve software quality. In a nutshell, 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 $quality(S_{i+1}) > quality(S_i)$. Indeed, our approach provides software architects a real grasp whether refactorings improve software quality or not.

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 if our approach would recommend the moved methods to return to their original place. After testing ten different calibration criteria, we calibrated the approach with the one that achieved the best recall average (57.5%, specifically).

We also implemented QMove, a prototype plug-in for Eclipse IDE that supports our proposed restructuring 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.

Finally, we performed three types of evaluation. First, we evaluated our approach on 13 open-source systems. Similar to our calibration method, we modified the original systems by randomly moving a subset of their methods to other classes. Next, we verified if our approach recommended the moved methods to return to their original classes. As result, QMove could move back 84.2% of the methods, on average. Second, we compared 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 performed a comparative evaluation of these tools in two proprietary systems that was oversaw by experts developers, and our approach obtained a greater number of positively evaluated recommendations.

The remainder of this paper is organized as follows. Section 2 presents the basic concepts to better understand our approach. Section 3 describes our approach, the calibration process, and the tool support. Section 4 reports three types of evaluation to verify our approach. Section 5 exhibit the threats to validity. Finally, Section 6 discusses the related work and Section 7 concludes.

SBQS, October 17–19, 2018, Curitiba, Brazil
2018. ACM ISBN 978-1-4503-6565-9/18/10…$?? (Pre-print author version)
https://doi.org/10.1145/3275245.3275247

2 BACKGROUND

This section presents fundamental concepts for understanding our proposed approach. Section 2.1 describes basic concepts on refactoring. Section 2.2 introduces the QMOOD model for quality assessment, and Section 2.3 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 of the system to be changed [19]. 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) [6].

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 [14]. From the several types of refactoring, we highlight the Move Method, which is the core of our proposed restructuring approach. A Move Method refactoring consists in moving a method from one class to another. The move can even occur to classes in different 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 from another class than its own (a bad smell named Feature Envy).

2.2 Quality Model for Object Oriented Design

There are several ways to measure quality of an object-oriented software, such as CK metrics (*Chidamber and Kemerer*) [5], MOOD (*Metrics for Object Oriented Design*) [1, 2], and QMOOD (*Quality Model for Object Oriented Design*) [3]. Our approach relies on the latter 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.

QMOOD quality model 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 the quality of a software as a whole or on a subset of the six mentioned attributes. QMOOD also helps to access object-oriented design property, provides search-based refactoring, and quantifies quality attributes with the help of equations [12].

This model defines 11 object-oriented design properties and links them to an appropriate design metric (Table 1). Then, it proposes equations using the design properties to measure the six quality attributes (Table 2) [3]. We employ these equations in our approach.

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 smaller number of repeated codes and a greater possibility of their reuse in other systems, compared to S.

Table 1: Design Metrics for Design Properties

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

Table 2: Equations for 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

2.3 Recall, Precision and F-Score

A view on the performance of a sample data can be given by the precision-recall curve, which is commonly summarized in a single indicator [9], e.g., 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 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 it is classified as positive, it is counted as a false positive (*fp*) [7].

We use these outcomes to calculate precision and recall values and consequently obtain the f-score value, as follows [16].

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

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

3 PROPOSED APPROACH

We propose a semi-automatic restructuring approach by using Move Method refactoring and six quality attributes defined by QMOOD (refer to Table 2). The main idea is, given a software system S, to move methods between classes in S in order to recommend a sequence of refactorings $R_1, R_2, \ldots, R_n$ that result in system versions $S_1, S_2, \ldots, S_n$, where $quality(S_{i+1}) > quality(S_i)$.

First, the approach calculates the six quality attributes for the analyzed software. Second, we detect every method that could be automatically moved to another class. Third, for each method, we move it to different classes that can receive it automatically, recalculate the quality attributes, and return it to its original class. Fourth, we include the refactoring that achieved better quality improvement to the recommendation list and repeat the third step for the remaining methods.

After we have processed every method, we present a recommendation list showing the sequence of Move Method refactorings ordered by the first to last recommendation found on fourth step.

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 two classes: `A` and `B`. Class `A` has two methods: `methodA1` and `methodA2`. For this example, method `methodA2`—which receives a `B` object as a parameter—should be in class `B`. Therefore, this creates a new system version `S′` when we move the method. Figure 1 shows a UML diagram of the classes described in our example, before and after we move the aforementioned method.

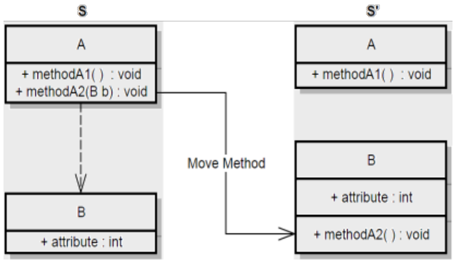


Figure 1: Move Method applied to our Example

When we apply our approach to system `S`, we first compute the QMOOD quality attributes for `S`. Then, we detect `methodA2` as a method that could be moved to another class. The method is moved to class `B` creating the new system version `S′`. We recompute the quality metrics for `S′` and then we return the method to class `A`, where it originally came from. In this particular case, since there are only two classes, our approach finishes the analysis. However, when there are other classes, our approach would repeat the process by moving the method to another class and recalculating the quality metrics again.

Table 3 shows the QMOOD quality attributes for `S` and `S′`, and the difference between `S′` and `S`. Even though *flexibility*, *understandability*, and *functionality* values remain the same, the values for the other three quality attributes—*reusability*, *extendibility*, and *effectiveness*—improve. Since it shows better quality attributes, our approach would recommend `methodA2` to be moved to class `B` (as previous illustrated in Figure 1).

Table 3: Quality Attributes for our Example

Quality Attribute	S	S’	S’ - S
Reusability	1.4	1.5	0.1
Flexibility	-0.12	-0.12	0
Understandability	0.2	0.2	0
Functionality	0.25	0.25	0
Extendibility	0.69	0.72	0.03
Effectiveness	-1.4	-1.31	0.09

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  Input: methods, a list with every method and their respective class from the
     analyzed system
2  Output: recommendations, an ordered sequence of Move Method refactoring
     that can be applied to the analyzed system
3  begin
4      potRefactor := $\emptyset$
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 := $\emptyset$
12     metrics := $\emptyset$
13     while potRefactor $\neq \emptyset$ 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 from highest to lowest according to the quality measurements.

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) (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 (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 Systems. Table 4 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. 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.

Table 4: Systems used in calibration process

System	Version	# of classes	# of methods	LOC
JHotDraw	4.6	674	6,533	80,536
JUnit	r4.12	1092	2811	26,111
MyAppointments	1.0	22	99	1,213
MyWebMarket	1.0	18	107	1,034

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.

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 1 uses absolute values of each metric before and after the refactoring, while Equation 2 uses relative values.

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

$$\text{Rel\#1: } \forall m\% \in M\%,\ m\% \geq 0 \land \exists m\% \in M\%,\ m\% > 0 \qquad (2)$$

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 and 4, respectively).

$$\begin{aligned} \text{Abs\#2: } &\forall m \in M \setminus \{\text{effectiveness}\},\ m(S') \geq m(S) \\ \land \quad &\exists m \in M \setminus \{\text{effectiveness}\},\ m(S') > m(S) \end{aligned} \qquad (3)$$

$$\begin{aligned} \text{Rel\#2: } &\forall m\% \in M\% \setminus \{\text{effectiveness}\},\ m\% \geq 0 \\ \land \quad &\exists m\% \in M\% \setminus \{\text{effectiveness}\},\ m\% > 0 \end{aligned} \qquad (4)$$

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 5 represents absolute version of the function `fitness` and Equation 6 represents the relative one.

$$\begin{aligned} \text{Abs\#3: } &s = sum(M), \\ &s(S') > s(S) \end{aligned} \qquad (5)$$

$$\text{Rel\#3: } sum(M\%) > 0 \qquad (6)$$

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 [18] stated that *Move Method* refactoring usually increases the values for *flexibility*, *understandability*, and *extensibility*. Therefore, particularly in this calibration, we consider only these three attributes: *flexibility*, *understandability*, and *extensibility*; 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 7 and 8 represent absolute and relative versions of the `fitness` function, respectively.

$$\begin{aligned} \text{Abs\#4: } &\forall m \in \text{M'},\ m(S') \geq m(S) \\ \land \quad &\exists m \in \text{M'},\ m(S') > m(S) \end{aligned} \qquad (7)$$

$$\begin{aligned} \text{Rel\#4: } &\forall m\% \in \text{M'\%},\ m\% \geq 0 \\ \land \quad &\exists m\% \in \text{M'\%},\ m\% > 0 \end{aligned} \qquad (8)$$

In the fifth and last calibration type, we used the following three design metrics (Table 1): CAM (cohesion), DCC (coupling), and CIS (messaging). We chose these metrics because they are the QMOOD design metrics that usually change when a method is moved. 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 9 and 10 represent absolute and relative `fitness` functions, respectively.

$$\begin{aligned} \text{Abs\#5: } & \forall m \in \text{DM},\ m(S') \geq m(S) \\ \wedge \quad & \exists m \in \text{DM},\ m(S') > m(S) \end{aligned} \tag{9}$$

$$\begin{aligned} \text{Rel\#5: } & \forall m\% \in \text{DM\%},\ m\% \geq 0 \\ \wedge \quad & \exists m\% \in \text{DM\%},\ m\% > 0 \end{aligned} \tag{10}$$

3.3.3 Results. Table 5 reports the calibration types, 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.

Table 5: Calibration Results

		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.

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 of 38.8% for both Abs#1, Rel#1, Abs#2, and Rel#2. Since the number of recommendations being the same, consequently resulting in the same recall values.

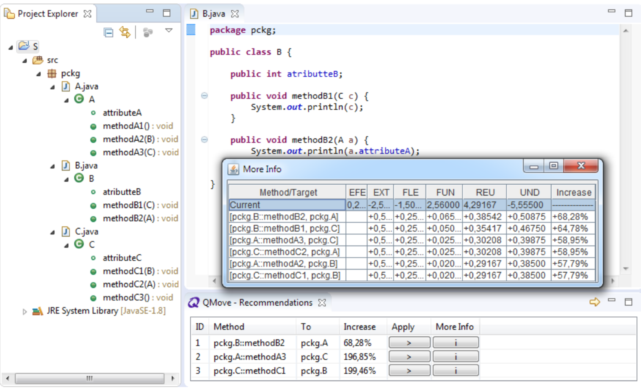
For the third calibration, Abs#3 had the average recall of 56.2%, the best result so far, since is an increase of 17.4% w.r.t. the previous strategies. However, for Rel#3, the average recall rose to 57.5%, i.e., a subtle increase of 1.3%. It occurs due exclusively to the difference of recall values calculated 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 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]. By default, our tool relies on calibration Rel#3 but it provides ways to define the preferable calibration option based on users' experience. Figure 2 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 when the refactoring is applied. 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 reusability remains the same, recommendation ID 1 improves flexibility from -2.5 to -2 (+0.5), effectiveness from -1.5 to -1.25 (+0.25), extendibility from 2.56 to 2.625 (+0.065), functionality 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.



```
package pckg;

public class B {

    public int atributteB;

    public void methodB1(C c) {
        System.out.println(c);
    }

    public void methodB2(A a) {
        System.out.println(a.attributeA);

}
```

Method/Target	EFE	EXT	FLE	FUN	REU	UND	Increase
Current	0,2...	-2,5...	-1,50...	2,56000	4,29167	-5,55500	------------
[pckg.B::methodB2, pckg.A]		+0,5...	+0,25...	+0,065...	+0,38542	+0,50875	+68,28%
[pckg.B::methodB1, pckg.C]		+0,5...	+0,25...	+0,050...	+0,35417	+0,46750	+64,78%
[pckg.A::methodA3, pckg.C]		+0,5...	+0,25...	+0,025...	+0,30208	+0,39875	+58,95%
[pckg.C::methodC2, pckg.A]		+0,5...	+0,25...	+0,025...	+0,30208	+0,39875	+58,95%
[pckg.A::methodA2, pckg.B]		+0,5...	+0,25...	+0,020...	+0,29167	+0,38500	+57,79%
[pckg.C::methodC1, pckg.B]		+0,5...	+0,25...	+0,020...	+0,29167	+0,38500	+57,79%

ID	Method	To	Increase	Apply	More Info
1	pckg.B::methodB2	pckg.A	68,28%	>	i
2	pckg.A::methodA3	pckg.C	196,85%	>	i
3	pckg.C::methodC1	pckg.B	199,46%	>	i

Figure 2: `QMove` plug-in[2]

4 EVALUATION

Section 4.1 reports a synthesized analysis we performed 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.

[1] https://github.com/pqes/QMove, verified 2018-07-04. The execution time of our tool is mostly lengthy. For example, QMove running on JHotDraw (largest system of our calibration) takes about 1 day and on the evaluated systems takes between 1 and 7 days. Nevertheless, we claim that our approach should be used in remodularization tasks, which occur once in a while, and it is indeed not suitable for continuous application.

[2] One could question rec. ID 1 improves 68.28% and rec. ID 2 improves 196.85%. However, rec. ID 1 is the best in version S_i and rec. ID 2 is the best in version S_{i+1}. Particularly in this example, rec. ID 2 in S_i would improve 58.95%, which is less than 68.28%.

4.1 Synthesized Evaluation

This section evaluates our proposed refactoring approach through QMove in open-source systems.

Subject systems: We rely on 13 open-source systems (Table 6) that possess well-defined architectures and are active projects. These systems have been used in the evaluation of a third-party work [20], which facilitates comparing our approach to JMove and JDeodorant.

Table 6: ***Subject Systems***

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

Methodology: 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.[3]

Results: Table 7 reports the evaluation results for each system, the total number of recommended methods, the *Gold Set* (GS) size, the recommendations from the *Gold Set* (GS), and the achieved f-score, precision, and recall. 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 3 and 4 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.

Table 7: Evaluation Results[4]

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%**

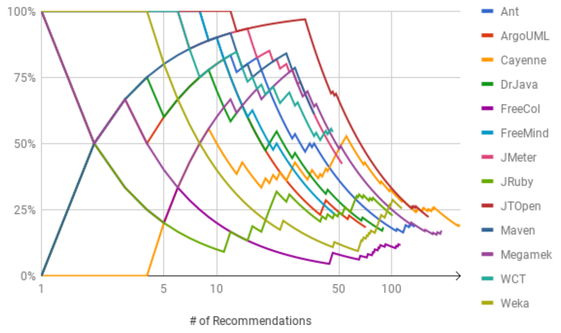


Figure 3: Precision Results

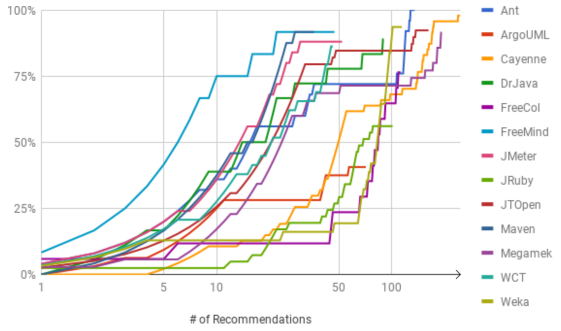


Figure 4: Recall Results

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 (*Top*3), five (*Top*5), ten (*Top*10), 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 7).

Figure 5 graphically illustrates the precision behavior and shows the data referring to the average precision of *Top*3 to *TopN*, the

[3] We do not rely on the same *Gold Set* from [20] 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 [20].

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.

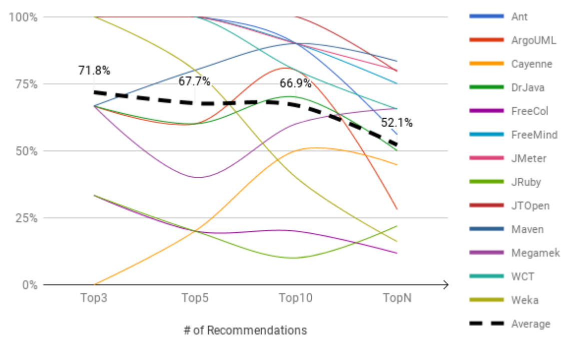


Figure 5: Precision Tops

Figure 6 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. 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.

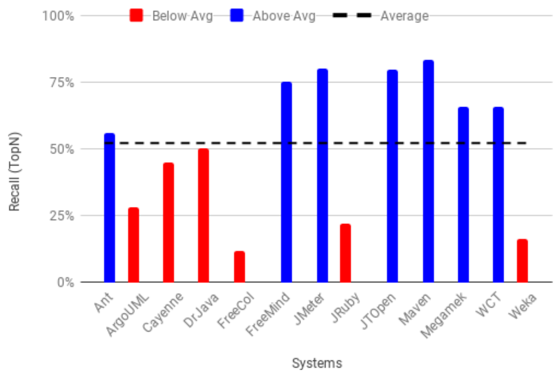


Figure 6: Recall Tops

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 7). Our goal is to observe the point that the f-score has its highest value, which consequently sets the highest values of precision and recall, before the f-score values begin to decline. This allows us to find 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. All the graphs are in logarithmic scale, and the abscissa axis has the size of 250, to simplify the comparison between them.

By analyzing the graphs represented in Figure 7, we detected the points where the 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 8 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 8: Best F-Score at Recommendation

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	-	**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.

Subject systems: We used the same systems used in our synthesized evaluation (refer to Table 6).

Methodology: We ran JMove and JDeodorant on these 13 systems to verify if they recommend moving back the methods from our *Gold Set*.

Results: Table 9 reports the number of refactoring recommendations by each tool (Recs.), the number of refactoring related to methods from the *Gold Set* (Recs. GS), and the f-score, precision, and recall values calculated through the mean of the precision and recall values for each analyzed system.

Table 9: Comparative Results

System	Recs.	Recs. GS	F-score	Prec.	Recall
QMove	1,404	313	39.1%	27.1%	84.2%
JMove	2,091	113	10.1%	6.6%	30.1%
JDeodorant	1,364	117	15.7%	13.4%	29.5%

As we can see in Table 9, QMove performed better than the other tools for all metrics. QMove f-score was 39.1%, which is more than twice as JDeodorant (15.7%) and almost four times as JMove (10.1%).

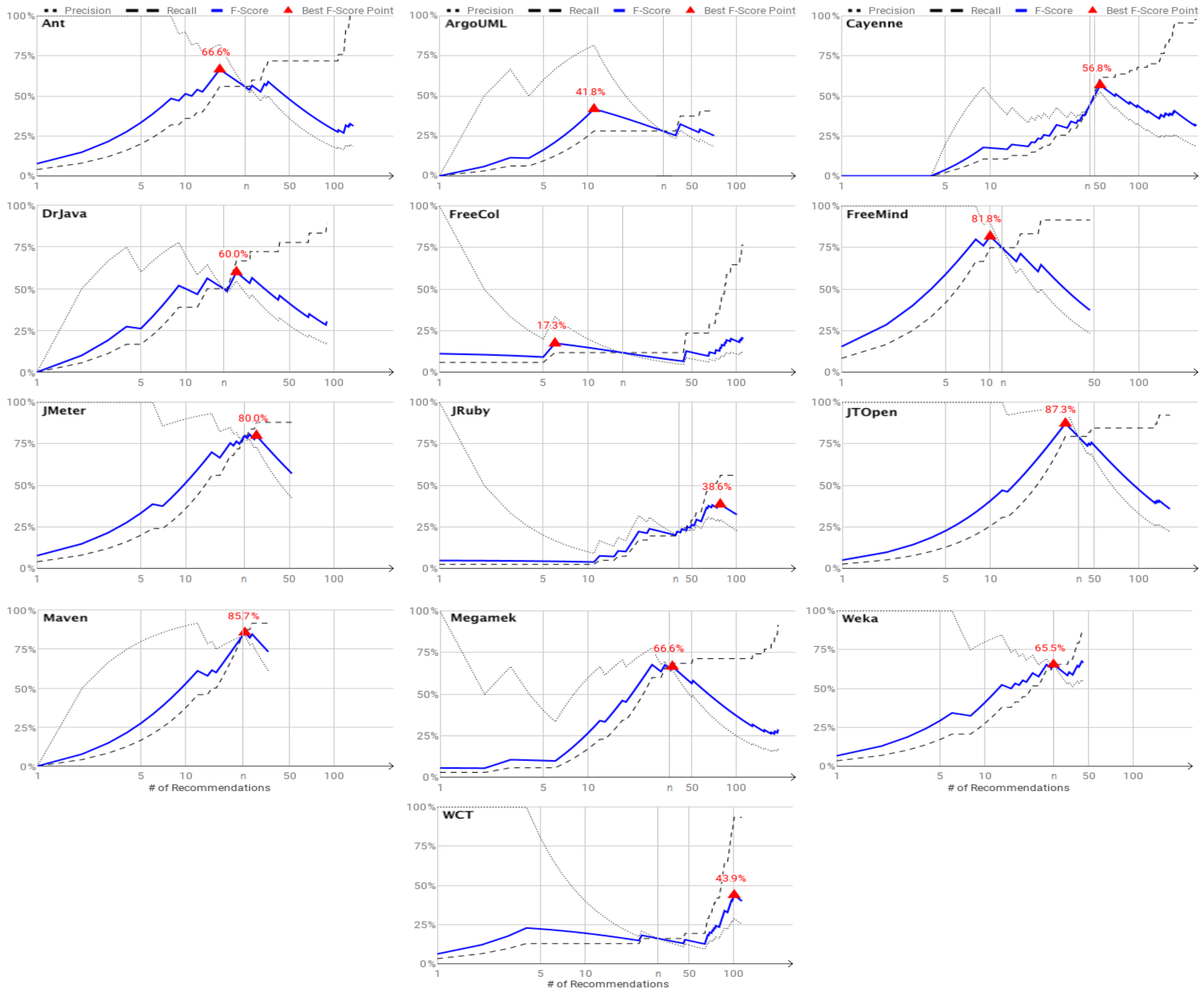


Figure 7: Recall, precision and f-score results

Considering precision and recall, QMove also performed at least twice as much as the other tools.

Figure 8 shows the overlap of the Move Method recommendations belonging to the *Gold Set* of the three evaluated tools.

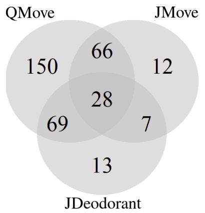


Figure 8: Overlapping between comparative results

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.

4.3 Real Scenario Evaluation

This section evaluates our proposed refactoring approach through QMove in real-world systems.

Subject systems: We rely on two proprietary systems developed by GT4W - Geo Technology for Web, a IT company located in Lavras, Minas Gerais, Brazil. Table 10 reports data about these systems.[5] ReMent is a demand management system and Cyssion a concession management system. We chose these systems because

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

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 10: Proprietary Systems

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

Methodology: 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 [11], 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 justify the chosen option, thus gathering useful information that could contribute to our evaluation.

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

For Cyssion, Table 11 reports the evaluation results of the 41, five, and six recommendations triggered by QMove, JMove, and JDeodorant, respectively.

Table 11: 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 neutrality, QMove would have eight recommendations against three from JMove and one from JDeodorant. Figure 9 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).

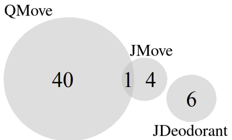


Figure 9: Overlapping between Cyssion results

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.

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.

5 THREATS TO VALIDITY

Next, we identify at least two threats to validity in our study.

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 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.

6 RELATED WORK

In this section, we highlight and discuss some studies that are closely related to our proposed restructuring approach.

Terra et al. [20] propose JMove, a tool for Move Method refactorings. Given a method m located in a class C, it calculates two similarities using Jaccard: (i) the average similarity between m and the remaining methods in C and (ii) the average similarity between m and the methods in another class C'. If the similarity measured in C' is greater than the one measured in C, then 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 potential recommendations.

Tsantalis and Chatzigeorgiou [21] present JDeodorant, a tool that suggests Move Method refactorings as solutions to the Feature Envy design problem. JDeodorant 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. Whereas JDeodorant is based only on cohesion and coupling metrics, our approach, using the QMOOD model, provides a broad set of metrics that measure improvements to overall quality of the system.

Mkaouer et al. [15] propose an approach that searches for a sequence of refactoring actions to maximize the six QMOOD quality attributes while minimizing the number of refactoring actions. Although their approach covers more refactoring types, it does not analyze all refactoring options, which may lead to a sub-optimum restructuring. In contrast, our approach analyzes every possibility but currently considers only the Move Method refactoring.

O'Keeffe and Cinnéide [17] present a tool for refactoring based on three QMOOD quality attributes, called Code-Imp. They propose four search algorithms to maximize *flexibility*, *understandability*, and *extensibility*. Even though their tool supports many refactoring types, it does not support the Move Method refactoring, which is the refactoring we used in our approach.

Griffith et al. [10] describe an approach to detect code smells by using CK (Chidamber-Kemerer) and size-oriented metrics, whereas our approach uses a more solid model. They employ a genetic algorithm to find the best refactoring sequence that removes the most number of code smells. While their approach outputs the refactoring in a UML class diagram, our approach allows the software architect to perform the recommended refactorings automatically in the source code.

7 FINAL REMARKS

Even though there are many refactoring approaches, very few consider their impact on the software quality. In this paper, 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.

In our first evaluation, 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 evaluation, 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. 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 evaluation, we evaluate 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.

Future work includes: (i) to incorporate other types of refactorings, such as Extract Class and Extract Method, (ii) to improve QMove preconditions to avoid false positives, and (iii) to rely on other metrics and other search-based algorithms to allow users to set up their own fitness function and search-based algorithm.

Acknowledgements: Our research has been supported by CAPES, FAPEMIG, and CNPq. We also would like to thank the developers Alisson José Oliveira de Faria and Mário de Carvalho Joaquim Filho, and the IT company GT4W - Geo Technology for Web for the valuable collaboration in the real scenario evaluation.

REFERENCES

[1] Fernando Brito Abreu and Rogério Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *4th International Conference on Software Quality*, pages 1–8, 1994.

[2] Fernando Brito Abreu, Miguel Goulão, and Rita Esteves. Toward the design quality evaluation of object-oriented software systems. In *5th International Conference on Software Quality*, pages 44–57, 1995.

[3] Jagdish Bansiya and Carl Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[4] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. Methodbook: Recommending Move Method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694, 2014.

[5] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[6] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[7] Tom Fawcett. An introduction to ROC analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[8] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley, USA, 1999.

[9] Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In *27th European Conference on Information Retrieval (ECIR)*, pages 345–359, 2005.

[10] Isaac Griffith, Scott Wahl, and Clemente Izurieta. TrueRefactor: An automated refactoring tool to improve legacy system and application comprehensibility. In *24th International Conference on Computer Applications in Industry and Engineering (CAINE)*, pages 316–321, 2011.

[11] Susan Jamieson. Likert scales: how to (ab) use them. *Medical Education*, 38(12):1217–1218, 2004.

[12] Bhavna Katoch and Lovepreet Kaur Shah. A systematic analysis on MOOD and QMOOD metrics. *International Journal of Current Engineering and Technology*, 4(2):620–622, 2014.

[13] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. *Electronic Notes in Theoretical Computer Science*, 82(3):483–499, 2003.

[14] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[15] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545, 2016.

[16] David L Olson and Dursun Delen. *Advanced data mining techniques.* Springer Science & Business Media, Germany, 2008.

[17] Mark O'Keeffe and Mel O Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[18] Raed Shatnawi and Wei Li. An empirical assessment of refactoring impact on software quality using a hierarchical quality model. *International Journal of Software Engineering and Its Applications*, 5(4):127–149, 2011.

[19] Ricardo Terra, Marco Tulio Valente, and Nicolas Anquetil. A lightweight remodularization process based on structural similarity. In *10th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, pages 111–120, 2016.

[20] Ricardo Terra, Marco Tulio Valente, Sergio Miranda, and Vitor Sales. JMove: A novel heuristic and tool to detect Move Method refactoring opportunities. *Journal of Systems and Software*, 138:19–36, 2017.

[21] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of Move Method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.