Quality-oriented Move Method Refactoring

Christian Marlon Souza Couto
Department of Computer Science
Federal University of Lavras
Lavras, Brazil
Email: christiancouto@posgrad.ufla.br

Henrique Rocha
Inria Lille - Nord Europe
Lille, France
Email: henrique.rocha@gmail.com

Ricardo Terra
Department of Computer Science
Federal University of Lavras
Lavras, Brazil
Email: terra@dcc.ufla.br

***Abstract*—Restructuring is an important activity to improve software internal structure. Even though there are many restructuring approaches, very few consider the refactoring impact on the software quality. In this paper, we propose an semi-automatic software restructuring 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 to find the best criteria to measure the improvement of quality. In our preliminary evaluation on three open-source systems, our approach achieved an average recall of 57%.**

I. Introduction

The refactoring process changes the code to improve the internal structure without compromising its external behavior [2]. Currently, there are many restructuring approaches where the degree of automation can vary [4]. Nevertheless, there are very few that consider their impact in software quality metrics. Consequently, a software system may be restructured into a version that worsens its overall quality.

In this paper, on the context of a search-based software engineering research, we propose an semi-automatic software restructuring approach based on software quality metrics. We rely on the measurements of the Quality Model for Object Oriented Design (QMOOD) [1] 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 the `JHotDraw` system 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 five different calibration criteria, we calibrated the approach with the one that achieved the best f-score (38%), which corresponds to a precision of 27% and a recall of 65%.

We 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 evaluated our approach on three open-source systems: FreeMind, Maven, and WCT. 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 57% of the methods, on average.

The remainder of this paper is organized as follows. Section II presents the basic concepts to better understand our approach. Section III describes and calibrates our proposed approach. Section IV evaluates our approach. Finally, Section V discusses the related work and Section VI concludes.

II. Background

In this section, we present the basic concepts on refactoring (Section II-A) and introduce the QMOOD model for quality assessment (Section II-B).

A. Refactoring

Refactoring is basically restructuring applied to object-oriented programming [5], which can be described as transformations in a software that preserve its behavior.

From the several types of refactoring, we highlight the Move Method, which is used in 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 to improve the software quality. A common scenario for this refactoring is when we realize that a method depends more from members from another class than its own (a bad smell named Feature Envy).

B. Quality Model for Object Oriented Design

Bansiya and Davis [1] proposed QMOOD (*Quality Model for Object Oriented Design*) to measure software quality aspects in object oriented projects. This model defines 11 object-oriented design properties and links them to an appropriate design metric (Table I). Then, it identifies six qualities attributes based on the ISO 9126 and propose equations using the design properties to measure such qualities (Table II) [1]. We employ the equations proposed for the quality attributes in our proposed approach.

TABLE I: 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 II: 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

III. Proposed Approach

We propose an semi-automatic restructuring approach by using Move Method refactoring and six quality attributes defined by QMOOD (Table II). First, the approach calculates the six quality attributes for the analyzed software. Second, we detect every method that could be moved automatically to another class. Third, for each method, we move it to a different class, recalculate the quality attributes, and return it to its original place. Fourth, we include the refactoring that achieved better quality measurements to the recommendation list and repeat the third step for the remaining methods. After we processed every method, we present a recommendation list showing the sequence of Move Method refactorings ordered by their quality results (highest to lowest).

A. Example

This section illustrates a Move Method refactoring scenario where our approach could be performed. Suppose a small Java system `S` with two classes: `A` and `B`. Class `A` has two methods: `methodA1` and `methodA2`. For this example, we highlight method `methodA2` (Listing 1) that receives a `B` object as a formal parameter. The code inside the method accesses only the attribute from class `B` by using the object `b` (lines 2-3). More specifically, the method prints class `B` attribute if its value is not zero (lines 3) and "Empty" otherwise (line 5). Therefore, we can deduce that it would be more appropriate if we moved this method to class `B`, which creates a new system version `S'`. Figure 1 shows a UML diagram of the classes described in our example, before and after we move the aforementioned method.

Listing 1: Method Example

```
1 public void methodA2(B b){
2   if (b.attribute != 0){
3     System.out.println(b.attribute);
4   } else {
5     System.out.println("Empty");
6   }
7 }
```

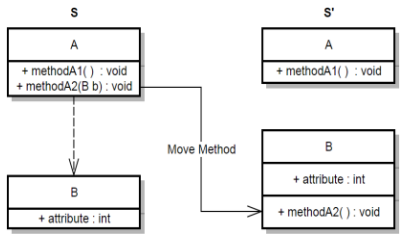

Fig. 1: *Move Method* applied to our Example

When we apply our approach to system `S`, first we 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`, which is its original place. In this particular case, since there are only two classes, our approach finishes its analysis. However, if other classes did exist, then our approach would repeat the process by moving the method to another class and recalculating the quality metrics again.

Table III shows the QMOOD quality attributes for `S` and `S'`, and the difference between `S'` and `S`. Even though, *flexibility*, *effectiveness*, and *extendibility* remain the same, the values for the other three quality attributes (*reusability*, *functionality*, *understandability*) 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 III: Quality Attributes for our Example

	S	S'	S' - S
Reusability	1.4375	1.5000	0.0625
Flexibility	-0.1250	-0.1250	0.0000
Understandability	0.2000	0.2000	0.0000
Functionality	0.2500	0.2500	0.0000
Extendibility	0.6900	0.7200	0.0300
Effectiveness	-1.4025	-1.3190	0.0835

B. Algorithm

Algorithm 1 describes our proposed approach in a high-level abstraction. 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 the actual system).

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 which methods of the system (m) that can be automatically moved to other classes (C) (lines 6-10) and store the pairs (m, C) in the list of potential refactorings (line 8).

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 := ∅
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 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 is used to as the system baseline now (line 26).

After the execution of Algorithm 1, the sequence of Move Method refactorings is recommended to the user. The larger the number of refactorable methods, the higher the execution time. We argue that performance is not critical since our approach is designed to be performed as part of night builds.

C. 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 they are an improvement according to our requirements. Our objective is to identify the best set of requirements for the `fitness` to make our approach recommend better refactoring options.

We chose the `JHotDraw`[1] software as the baseline for our calibration process. Our main reason for using JHotDraw in our calibration is due its methods are likely to be in their proper classes, since it is developed and maintained by a small number of expert developers. We used version 4.6, which is composed of 674 classes, 6,533 methods, and 80,536 lines of code. For the calibration, we randomly moved 20 methods from `JHotDraw` to other classes. The information about these methods and classes (original and newly moved), we called *Gold Set*. We employ the *Gold Set* to verify if our algorithm recommends moving those methods back to their original place. In theory, the `fitness` function that recommends more methods from the *Gold Set* performs better.

Table IV summarizes the calibrations showing the description, the total number of recommended methods, the recommendations from the *Gold Set* (GS), and we also calculated precision, recall, and f-score.

In the first calibration, our criterion was the more simplistic where we verified if none of the quality attributes decreased and at least one attribute increased. We used such criteria for our `fitness` function. Then, we executed our algorithm to the modified `JHotDraw` and we got 28 methods recommended. However, only two methods belonged to the *Gold Set*. Therefore, we investigated why the 18 remaining methods in the *Gold Set* where not recommended by our algorithm. We discovered that the *effectiveness* value would get worse for most the *Gold Set*, which discarded those methods from the 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. This resulted in our algorithm to recommend five methods from the *Gold Set*.

In the third calibration, our criterion was as simplistic as the first one where we compare the overall sum of all six quality attributes. By using this new `fitness` function, we managed to find 11 methods from the *Gold Set* but the total number of recommendations increased to 56.

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) Mkaouer et al. [6] stated that *Move Method* refactoring usually increases the values for *flexibility*, *understandability*, and *extensibility*. Therefore, here we focus only on these three attributes: *flexibility*, *understandability*, and *extensibility*. The remaining attributes were not considered for this calibration. This calibration recommended 48 methods in which 13 belonged to the *Gold Set*.

In the fifth calibration, we used the following three design metrics (Table I): 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. This calibration resulted in 45 recommended methods in which only five belonged to the *Gold Set*.

When we consider the f-score values, the fourth calibration achieved the best results. Recall is also very important, and

[1] JHotDraw is a Java framework for graphic objects, and its design relies on well-known design patterns. http://www.jhotdraw.org/, verified 2017-10-22.

TABLE IV: Calibration Results

#	Description	Total Recs.	Recs. from Gold Set	Precision	Recall	F-score
1	(i) no quality attribute decreases; (ii) improve at least one attribute	28	2	7.14%	10.00%	8.33%
2	Same as #1 but using five attributes, i.e., ignore the attribute *effectiveness*	37	5	13.51%	25.00%	17.54%
3	The sum of quality attributes > the sum of the previous measured attributes	56	11	19.64%	55.00%	28.94%
4	Same as #1 but using only *flexibility*, *understandability*, and *extensibility*	48	13	27.08%	65.00%	38.23%
5	Cohesion, coupling, and messaging cannot decrease	45	5	11.11%	25.00%	15.38%

the fourth calibration also achieved the best recall result. Therefore, our `fitness` function uses the criteria defined by the fourth calibration, i.e., the quality attributes of *flexibility*, *understandability*, and *extensibility* cannot decrease and at least one of them should increase.

IV. Evaluation

This section evaluates our proposed restructuring approach through QMove, a prototype plug-in for Eclipse IDE we implemented to support our proposed restructuring approach.[2] We chose three open-source systems (Table V) that possess a well-defined architecture and present a similar number of classes `JHotDraw` does. We can assume, differently of JHotDraw, that *most* methods of these systems are likely to be in proper classes and hence we only evaluate recall.

TABLE V: *Subject Systems*

System	Version	# of classes	# of methods	LOC
FreeMind	0.9.0	658	4,885	52,757
Maven	3.0.5	647	4,888	65,685
WCT	1.5.2	539	5,130	48,191

Similarly to our calibration using `JHotDraw`, we modified the subject systems by randomly moving ten methods of each to other classes. Those methods compose our *Gold Set*, and our evaluation consists in verifying if *Gold Set* methods are recommended back to their original place by our approach. Table VI reports the evaluation results for each system, the total number of recommended methods, the recommendations from the *Gold Set*, and the achieved recall.

TABLE VI: Recall Results

System	Total Recs.	Recs. from Gold Set	Recall
FreeMind	28	6	60%
Maven	15	6	60%
WCT	8	5	50%
Average	**17**	**5.7**	**57%**

Our evaluation results shows a 57% average recall for methods in the *Gold Set*. This result is quite lower than the calibration, where we achieved 65% recall for the *JHotDraw* system (Table IV). The FreeMind and Maven systems performed closer to our calibration (60% recall for both systems). Therefore, we claim that we still need to improve the approach to achieve better results. Although our approach considerably improved the quality of these systems, we cannot guarantee similar results in low-quality systems (further work).

V. Related Work

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

[2]https://github.com/rterrabh/QMove

Mkaouer et al. [6] 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 possible refactoring options, which may lead to a sub-optimum restructuring. By contrast, our approach analyzes every possibility for the Move Method refactoring.

Moghadam and Cinnéide [7] 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.

Napoli et al. [8] propose an approach to suggest Move Method refactoring opportunities in large object-oriented systems. They rely on conventional metrics—such as Fan-In, Fan-Out, LCOM, CBO, and a Jaccard similarity coefficient—to detect refactoring opportunities. However, their approach does not aim to recommend the refactorings that better improve the system quality, which contrasts the main goal of our approach.

Griffith et al. [3] 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.

VI. Final Remarks

In this paper, we proposed a search-based approach, to recommend Move Method refactorings based on QMOOD quality attributes. We calibrated our approach by testing five different criteria on the `JHotDraw` system. The best calibration achieved a f-score of 38%, which corresponds to a precision of 27% and a recall of 65%. Finally, we evaluated our approach in three open-source systems by randomly moving ten methods each. On average, our approach could move 57% of the methods back to their original class.

Future work includes: (i) fine-grained calibration strategies to increase the f-score; (ii) other evaluation metrics used for recommendation systems, such as likelihood, recall rate@k, and feedback; (iii) other types of refactorings, such as Extract Class and Extract Method; and (iv) work with normalized values of the QMOOD quality attributes.

REFERENCES

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

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

[3] I. Griffith, S. Wahl, and C. 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.

[4] 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.

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

[6] 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.

[7] Iman Hemati Moghadam and Mel Ó Cinnéide. Code-imp: A tool for automated search-based refactoring. In *4th Workshop on Refactoring Tools (WRT)*, pages 41–44, 2011.

[8] C. Napoli, G. Pappalardo, and E. Tramontana. Using modularity metrics to assist Move Method refactoring of large systems. In *7th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*, pages 529–534, 2013.