A Context-Based Refactoring Recommendation Approach Using Simulated Annealing: Two Industrial Case Studies

Marouane Kessentini
CIS department
University of Michigan
Dearborn, MI, USA
marouane@umich.edu

Troh Josselin Dea
CIS department
University of Michigan
Dearborn, MI, USA
deatroh@umich.edu

Ali Ouni
College of Information Technology
UAE University
Al Ain, UAE
ouniali@uaeu.ac.ae

ABSTRACT

Refactoring is a highly valuable solution to reduce and manage the growing complexity of software systems. However, programmers are "opportunistic" when they apply refactorings since most of them are interested in improving the quality of the code fragments that they frequently update or those related to the planned activities for the next release (fixing bugs, adding new functionalities, etc.). In this paper, we describe a search based approach to recommend refactorings based on the analysis of the history of changes to maximize the recommended refactorings for recently modified classes, classes containing incomplete refactorings detected in previous releases, and buggy classes identified in the history of previous bug reports. The obtained results on two industrial projects show significant improvements of the relevance of recommended refactorings, as evaluated by the original developers of the systems.

CCS CONCEPTS

• CCS → Software and its engineering → Software creation and management → Search-based software engineering

KEYWORDS

Refactoring, search based software engineering, software quality

ACM Reference format:
M. Kessentini, J. Dea, and A. Ouni. 2017. In *Proceedings of GECCO '17,* Berlin, Germany, 8 pages.

DOI: http://dx.doi.org/10.1145/3071178.3071334

GECCO '17, July 15-19, 2017, Berlin, Germany © 2017 ACM. ISBN 978-1-4503-4920-8/17/07 $15.00
DOI: http://dx.doi.org/10.1145/3071178.3071334

1 INTRODUCTION

Several studies show that programmers are postponing software maintenance activities that improve software quality, even while seeking high-quality source code for themselves when updating existing projects. High-quality source code can be characterized using several quality attributes, but maintaining this high level of quality is expensive. One reason is that time, and monetary pressures force programmers to neglect to enhance the quality of their source code.

The challenge that programmers face when trying to improve the software design structure while preserving the behavior is termed the "software refactoring problem" [1][7][11]. A large portion of existing refactoring tools suggests refactorings improve the overall quality of systems without a concrete prioritization plan [2][13]. As a result, the number of refactorings to apply can be large, and developers may spend a long time to select relevant refactorings.

When a high number of refactorings are recommended, manual refactoring becomes error-prone and time-consuming. Murphy-Hill et al. [2] show that most developers do not use fully automated refactoring techniques because they want to mix refactorings with semantic changes, something that is not permitted by existing methods. Also, developers find fully automated refactoring risky because it can introduce bugs or undesired changes.

In the current literature, Search-based refactoring techniques obtained promising results based on the use of mono-objective and multi-objective algorithms to optimize quality metrics [5][6][7] [15][17][16][18]. However, most of these techniques explore a large search space of possible solutions and recommend a large sequence of refactorings to apply. In fact, developers are more interested, in general, to refactor recently modified entities related to their current tasks (e.g. features update, fixing bugs, etc.) [2]. Furthermore, recent empirical studies show that most of the refactored code fragments in practice are buggy classes.

In this paper, we propose a profile-based approach for refactoring recommendations to satisfy the following requirements: 1) programmers prefer to improve mainly the quality of recently modified code before a new release due to limited resources and time, 2) several empirical studies [8][9][19][20] identified correlation between bugs and refactoring opportunities, and 3)

recently introduced refactorings may give an indication of quality issues that should be fixed and show an interest from programmers to refactor these code fragments.

To consider the above observations, we propose a search-based refactoring approach, based on multi-objective simulated annealing [4], to find the best solution satisfying two objectives: maximizing the number refactorings applied to buggy or recently modified classes, and minimizing the number of antipatterns [1] using a set of antipatterns detection rules [5]. We implemented our proposed approach and evaluated it on a set of two industrial systems provided by our industrial partner from the automotive industry. We did the evaluation only on these two systems since it is critical to evaluate the relevance of recommended refactorings by the original developers of the systems. Statistical analysis of our experiments showed that our proposal performed significantly better than existing search-based refactoring approaches [6][7] and an existing refactoring tool not based on heuristic search, JDeodorant [8] regarding the relevance and importance of recommended refactorings. In our qualitative analysis, we conducted a survey with the software developers who participated in our experiments to evaluate the relevance of the fixed quality violations in their daily development activities.

The remainder of this paper is structured as follows. Section 2 provides an account of the related work. Section 3 describes our profile-based refactoring approach while the results obtained from our experiments are presented and discussed in Section 4. Finally, in Section 5, we summarize our conclusions and present some ideas for future work.

2 BACKGROUND AND RELATED WORK

Refactoring is the process of improving the code quality of an existing system while preserving its external behavior [2]. The refactoring process includes several steps, but the most important ones are the detection of refactoring opportunities and the recommendation of relevant refactorings to fix those detected quality issues. To identify refactoring opportunities, the majority of existing studies are based on the concept of code smells [1]. These code smells correspond to design practices that have a negative impact on the maintainability, understandability, and performance of the software[3].

Meananeatra [11] proposed a semi-automated graph-based algorithm to reduce the refactoring effort. The proposed algorithm is based on three objectives to reduce the number of detected code smells, the number of applied changes and number of refactored code fragments. Another tool is proposed, called JDeodorant [8], and implemented as an Eclipse plug-in based on the use of quality metrics to detect design quality violations. Several templates are proposed to cover different possible standard strategies to fix the detected code smells. Kessentini et al. [5] proposed a mono-objective genetic algorithm to identify the optimal sequence of refactorings that reduce the number of code smells using a set of detection rules.

Nevertheless, Refactoring studies were not only limited to fixing design defects, but also driven by the optimization of the software design through optimizing software quality attributes. For example, Du Bois et al. [14] has intended to find an optimal distribution of features within software modules through moving existing methods and classes while decreasing coupling and increasing cohesion. Seng et al. [15] used a genetic algorithm the generate refactoring sequences that optimize class level properties based on several quality metrics.

In contrast with combining metrics into one fitness function, Harman and Tratt [17] suggested a multi-objective optimization approach to generate refactoring operations that find the best tradeoff among two conflicting measures namely, the coupling and the standard deviation of methods per class.

3 CONTEXT-BASED REFACTORING USING LOCAL SEARCH

3.1 Approach Overview

The goal of our approach is to find the most relevant refactorings for software developers to refactor their systems based on their recent update of the system. The general structure of our approach is sketched in Figure 1.

Our technique comprises two main components. The first component is the pre-processing phase to rank the list of possible classes to refactor. During this phase, three different parsers are executed to extract classes that are recently modified or refactored in recent releases or those mentioned in previous bug reports. The classes mentioned in recent commits, are maybe important to refactor since they have a high probability to include bugs or to be updated in the future comparing to stable classes that were not modified for many releases. Several empirical studies show that correlation exists between buggy classes and poor quality symptoms […]. Thus, relevant refactorings for the developers' context could be identified in these classes based on this pre-processing phase.

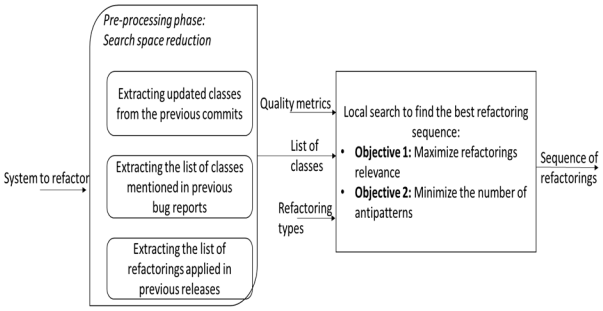

Figure 1: Approach overview

Developers, for example, may introduce bugs because of the complexity of the system and its poor design. Furthermore, the classes that are refactored recently by the developers but still contain quality issues can be recommended for further refactoring since already programmers expressed an interest in fixing them. The list of applied refactorings in previous releases are detected using the technique proposed in [18]. The outcome of this first phase is a list of classes that could be refactored based on the three main criteria detailed above.

The outcome of the first phase is used to reduce the search space to find the best refactoring sequence to recommend for developers. A multi-objective search algorithm is used to focus mainly on refactoring, if needed, the classes of the first phase while fixing some other quality issues as well. To this end, a multi-objective simulated annealing algorithm is executed for a number of iterations to find the solutions balancing the two objectives of 1) improving the relevance of recommended refactorings, which corresponds to maximize the number of refactoring recommendation in recently modified or buggy classes and 2) minimizing the number of antipatterns using a set of detection rules defined in the literature [5]. The first objective of the refactorings relevance is based on an average of three different measures of recently modified classes, recent classes mentioned in bug reports and recently refactored classes including incomplete refactoring activities or may need to be further refactored. The formalization of these measures will be described in the next section.

A multi-objective simulated annealing algorithm [4] is selected due to the small search space to explore after the pre-processing phase. A set of semantic constraints is used to check the correctness and feasibility of recommended refactorings based on textual similarities, call graphs and pre/post-conditions. These constraints are described in more details in [7]. The next section will discuss the formalization of our approach and the adaptation of the multi-objective simulated annealing algorithm to our problem.

3.2 Problem Formulation and Solution Approach

Simulated annealing is a local search heuristic inspired by the concept of annealing in metallurgy where metal is heated, raising its energy and relieving it of defects due to its ability to move around more easily [4]. As its temperature drops, the metal's energy drops and eventually it settles in a more stable state and becomes rigid. The local search algorithm of the Simulated Annealing is very suitable for exploring small search spaces. More details about Multi-Objective Simulated Annealing can be found in the following reference [4].

In the next sections, we described the three main steps of adaptation of MOSA to our problem.

Solution representation. A solution of our problem is defined as a sequence of a number of refactorings involving one or multiple source code fragments of the software to refactor. As described in Table I, the vector-based representation is used to define the refactoring sequence. Each dimension of the vector has a refactoring, and its index in the vector indicates the order in which it will be applied. For every refactoring, pre- and post-conditions are specified to guarantee the correctness of the operation.

The initial population is created by randomly selecting a sequence of operations to a randomly chosen set of code elements, or actors identified in the first phase of search space reduction. The type of actor usually depends on the type of the refactoring it is assigned to and also depends on its role in the refactoring operation. In our experiments, we used the following list of refactorings: *Extract class, Extract interface, Inline class, Move field, Move method, Push down field, Push down method, Pull up field, Pull up method, Move class, and Extract method*.

Fitness functions. The generated solutions are evaluated using two fitness functions as detailed in the following paragraphs.

Minimize the number of code smells: This fitness function is calculated based on the following equation:

$$\text{Min } f_1(s) = \frac{\#\text{code smells after refactoring}}{\#\text{code smells before refactoring}}$$

This function represents the proportion between the number of corrected defects (detected using bad smells detection rules) and the total number of possible defects that can be detected. The detection of defects is based on some metrics-based rules according to which a code fragment can be classified as a design defect or not (without a probability/risk score), i.e., 0 or 1, as defined in the detection rules of previous studies [5].

Maximize refactorings relevance: The main goal of the second fitness function is to evaluate the refactoring solutions based on their relevance to the developers. Formally, this function is defined as follows:

$$\text{Max } f_2(s) = \sum_{i=1}^{n} \frac{commitf(c_i) + bugreports\ f(c_i)}{2}{n},$$

where n is the number of classes to be refactored by the solution S, c is the class that contain at least one code smell and *commitf(c) and bugreportsf(c)* are respectively the functions to estimate the relevance of the class for refactoring based on previous changes in recent commits and previous bug reports.

The first function *commitf(c)* checks if a class was recently changed. In fact, a class that was modified recently has a high probability to be refactored comparing to stable classes. Thus, the function compares between the date of the last commit and the last date where the class was modified in the previous commit. If a suggested class was modified in the last commit, then the value of this function is 1. We define this normalized function, normalized in the range of [0, 1] as following:

$$commitf(c) = \frac{1}{commit.date(c) - lastcommit.date + 1}$$

The second function *bugreportsf(c)* counts the number of times a class was fixed to eliminate bugs based on the history of bug reports divided by the maximum number of times that a class in the system was fixed in previous bug reports. In fact, a class that was fixed several times has a high probability of being a buggy class and thus need to be refactored. Formally, this function, normalized between [0,1] is defined as:

$$bugreports\ f(c) = \frac{\frac{1}{lastbugreport.date(c) - lastcommit.date + 1} + \frac{NbFixedBugs(reports, c)}{MaxNbFixedBugs(reports)}}{2}$$

Change operators

MOSA is using a mutation operator to generate new solutions. For mutation, we use the bit-string mutation operator that selects one or more refactoring operations (or their controlling parameters) from the solution and replaces them by other ones from the list of possible operations to apply.

When applying the change operators, the different pre- and post-conditions are checked to ensure the applicability of the newly

generated solutions. We also apply a repair operator to randomly select new refactorings to replace those creating conflicts.

Table I. Example of first randomly generated operations.

Ref	Refactoring operation
RO001	MoveMethod(org.apache.xerces.xinclude.XIncludeTextReader, org.apache.xerces.xinclude.XIncludeTextReader, close())
RO002	MergePackage(org.apache.xerces.xpointer, org.apache.xerces.xs)
RO003	PullUpMethod(org.apache.html.dom.HTMLTableCaptionElementImpl,org.apache.html.dom.HTMLElementImpl, addEventListener())
RO004	ExtractInterface(org\apache.xml.serialize.SerializerFactory,apache.xml.serialize.SerializerFactoryInterface)

4 EVALUATION

4.1Research Questions and Evaluation Metrics

To evaluate and compare the performance and relevance of the recommended refactoring by our context-based multi-objective simulated annealing algorithm, we defined the following three research questions:

RQ1: To what extent can our approach recommends relevant refactorings to developers?

RQ2: To what extent can our approach reduces the number of refactorings and the execution time while improving the quality and recommending relevant refactorings compared to existing refactoring techniques?

RQ3: Can our approach be relevant for programmers in practice?

To address the first research question RQ1, we used both qualitative and quantitative evaluations of the recommended refactorings by our approach and existing studies.

For the quantitative validation, we asked a group of developers from our industrial partner to manually suggest a list of possible refactorings to apply based on the latest release source code of the system to refactor. Then, we used the precision (PR) and recall (RC) measures to evaluate the similarity between the recommended refactorings by our approach and those manually found by the original programmers of the industrial projects:

$$RC = \frac{|\text{set (recommended refactorings)} \cap \text{set (expected refactorings)}|}{|\text{set (expected refactorings)}|}$$

$$PR = \frac{|\text{set (recommended refactorings)} \cap \text{set (expected refactorings)}|}{|\text{set (recommended refactorings)}|}$$

Another metric that we considered for the quantitative evaluation is the percentage of fixed antipatterns (*NF*) by the refactoring solution. The code smells are detected on the new source code after refactoring based on the detection rules provided by [10]. Formally, *NF* is defined as

$$NF = \frac{\#\text{fixed code smells}}{\#\text{code smells}} \in [0,1]$$

The detection of antipatterns is very subjective and some developers prefer not to fix some smells because the code is stable or some of them are not important to fix. To this end, we considered another metrics the total gain in quality *G* for each of the considered QMOOD [3] quality attributes q_i before and after refactoring can be easily estimated as:

$G_{q_i} = q'_i - q_i$, where q'_i and q_i represents the value of the quality attribute *i* respectively after and before refactoring.

Since several good solutions can be relevant, it is important to check the relevance and correctness of recommended refactorings not only by comparing them with one expected solution (quantitative validation). Thus, we performed a qualitative evaluation where we asked the original programmers of the industrial projects to review, manually, if the recommended refactorings are relevant and correct or not from their perspectives. We define the metric Refactoring Relevance (*RR*) to mean the number of relevant refactorings divided by the total number of suggested refactorings. *RR* is given by the following equation:

$$RR = \frac{\#\text{relevant refactorings}}{\#\text{proposed refactorings}}$$

To answer RQ2, we compared our approach to random search (RS), mono-objective simulated annealing (SA) aggregating both objectives, another multi-objective evolutionary algorithm (NSGA-II) and an existing work based on search algorithms to fully-automate the refactoring recommendation process: O'Keeffe and Ó Cinnéide [11] and Ouni et al. [12].

O'Keeffe and Ó Cinnéide proposed a mono-objective formulation to automate the refactoring process by optimizing a set of quality metrics. Ouni et al. [7] proposed a multi-objective refactoring formulation that generates solutions to fix code smells. Both techniques are fully-automated and did not consider the personalization of refactoring recommendations. We have also compared our results with an existing tool, called JDeodorant, not based on heuristic search to fix quality issues by recommending refactorings. JDeodorant implements a set of templates to fix different design violations by providing a generic list of refactorings to apply. Since JDeodorant just recommends a few types of refactoring with respect to the ones considered by our tool. We restricted, in this case, the comparison to the same refactoring types supported by JDeodorant.

We used the metrics *PR, RC, NF, RC* and *G* to perform the comparisons and two new metrics related to the computational time (*CT*) and the number of refactorings (NR).

To answer RQ3, we asked the programmers to answer to a post-study questionnaire to get their opinions and feedback about our personalized refactoring recommendations.

4.2 Experimental Setup

To get feedback from the original developers of a system, we considered in our experiments two large industrial projects provided by our industrial partner, from the automotive industry. The first project is a marketing return on investment tool, called MROI, used by the marketing department to predict the sales of cars based on the demand, dealers' information, advertisements, etc. The tool can collect, analyze and synthesize a variety of data types and sources related to customers and dealers. It was

implemented over a period of more than eight years and frequently changed to include and remove new/redundant features.

The second project is a Java-based software system, JDI, which helps the Company to create the best schedule of orders from the dealers based on many business constraints. This system is also used by the company to find the best configurations of cars based on the requirements of dealers and customers. Software developers have developed several releases of this system at the company over the past 10 years. Due to the high number of changes introduced to this system over the years and its importance, it is critical to ensure that they remain of high quality and minimize the effort required by developers to fix bugs and extend the system in the future. Table IV described the statistics related to the two studied systems.

Our study involved 19 software developers from the company. Participants include 9 original developers of the MROI system and 10 original developers of the JDI one. All the developers who participated in the experiments are expert in Java, quality assurance and testing. The experience of these participants on these areas ranged from 7 to 18 years.

The questionnaire includes five main questions to be answered by the participants. Some of the questions are related to the background of the participants to evaluate their experience and ability to evaluate the results of our technique. Furthermore, we organized a lecture for all the participants about different concepts and examples related to software refactoring then they took six tests about evaluating the relevance of recommended refactorings on code fragments extracted from open source systems.

We formed two groups. Each of the two groups (A and B) is composed of the original developers of each system. We selected the participants of each group based on the collected background information to make sure that both groups have, in average, the same level of expertise with software refactoring and quality assurance. We provided to all the participants the questionnaire, the guidelines about the different steps to perform the experiments, the different used tools and source code of the systems to evaluate. After the first step of the quantitative evaluation, we provided to the participants the list of recommended refactorings by the different tools and asked them to evaluate their relevance and correctness. The participants are not aware of the tools used to get the different results. We counted the votes of the programmers for every of the recommended refactorings then we considered the highest number of votes to evaluate the correctness/relevance of the evaluated operations.

In the first scenario, we asked every participant to manually apply refactorings after reviewing the code of their systems. As an outcome of the first scenario, we estimated the similarity between the suggested refactorings and the expected ones as defined by the programmers.

In the second scenario, we asked the developers to manually evaluate the relevance of every recommended refactoring by our approach. In the third scenario, we collected the opinions of the developers about our tool based on a post-study questionnaire that will be detailed later. The programmers commented on the different evaluated refactorings and these comments/justifications were discussed later with the organizers of the study.

We used different population sizes of the used algorithms to evaluate their performance ranging from 100, 200, 300 and 500 individuals per population.

The maximum number of iterations is 100,000 evaluations for all the studied systems. We used the Wilcoxon test to compare between the different algorithms considered in our experiments. For each algorithm and project, we use the trial and error strategy to find the good parameters setting. For all the systems and algorithms, the obtained results in our experiments are statistically significant on 30 independent executions using the Wilcoxon rank sum test with a confidence level of 95% ($\alpha < 5\%$).

Table 2. The Evaluated Industrial Projects

Syst.	Release	Avg. #classes	Avg. KLOC	Avg. #code smells	#manual Refactorings
JDI	V1.0 - V5.8 (26 releases)	694	252	88	94
MROI	V1.0- V6.4 (31 releases)	827	269	116	119

To evaluate the difference in magnitude, we used the Vargha-Delaney A measure as a non-parametric effect size metric. Based on the different evaluation measures used in our experiments (such as *PR, RC, RR*, etc.), the *A* statistic estimates the probability that the execution of an algorithm *B1* (MOSA) has better performance than executing another algorithm *B2* (other existing refactoring studies). In the validation of this work, we found the following results: a) On the JDI system, the performance of our MOSA algorithm based on all the different evaluation metrics is better than existing studies with an *A* effect size more than 0.91; and b) On the MORI system, the performance of our MOSA algorithm based on all the different evaluation metrics is better than existing studies with an *A* effect size more than 0.88.

We used in our experiments, eight different types of code smells [1]: Blob, Long Parameter List (LPL), Functional Decomposition (FD), Spaghetti Code (SC), Data Class (DC), Feature Envy (FE), Shotgun Surgery (SS), and Lazy Class (LC). We selected these code smells because they are the most frequent and hard to fix defects based on recent empirical studies[2].

For the starting temperature and alpha value, we used respectively the following values 0.0003 and 0.999. When randomly generating a mutation, each type of mutation had the same probability of being generated; there was a one-third chance of adding a refactoring, modifying a refactoring, or removing a refactoring.

4.3 Results and Discussions

Results for RQ1. Figure 2 (*RR*) summarized the results of our approach of the qualitative evaluation when programmers manually evaluated the relevance and correctness of the recommended refactorings. Most of the solutions recommended by our personalized approach are relevant and correct from the perceptive of the programmers.

On average, for the two studied projects, around 88% of the proposed refactoring operations are found to be useful by the software developers of our experiments. The highest MC score is 89% for the JDI project and the *RR* score is 87% for the second system MROI. Thus, it is clear the obtained results are not dependent on the size of the systems and the number of recommended refactorings. Most of the refactorings that were not manually approved by the developers were found to be either fixing non-relevant quality issues or introducing design incoherence.

We also compared the proposed refactoring solutions with the ones that are provided manually by the programmers of these industrial systems. Figures 3-4 show that the majority of the proposed refactorings, with an average of 84% in terms of precision and 87% of recall, are equivalent to those manually found by the programmers when trying to refactor the system. The higher score of the recall comparing to the precision can be explained by the fact that our approach proposes a complete list of refactorings comparing to the manually recommended operations by the programmers due to the time-consuming process of code refactoring. Also, we found that the slight deviation with the expected refactorings is not related to incorrect operations but to the fact that the developers were interested mainly in fixing the severest quality issues or those related more to find better ways to extend the current design.

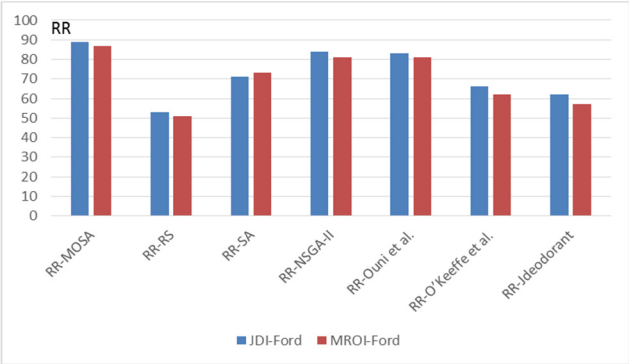

Figure 2. Median refactoring relevance (RR) value for 30 executions on the two systems with a 95% confidence level (α<5%).

Figure 6 shows that the refactorings recommended by the approach and applied by developers improved the quality metrics value (*G*) of the two systems. The average quality gain for the two industrial systems was the highest among the systems with more than 0.2. The improvements in the quality gain confirm that the recommended refactorings helped to optimize different quality metrics by fixing the most severe quality issues. Although the average quality gain is lower comparing to existing techniques, it is still comparable to them due to the much lower number of refactorings recommended by our technique.

Result for RQ2. Figures 3, 4, 5, 6, 7 and 8 confirm the average superior performance of our personalized refactoring approach compared to existing refactoring approaches. Figure 3 describes that our approach provides better refactoring relevance results (*RR*) than existing approaches having *RR* scores between 55% and 79%, as *RR* scores, on average, on the two different systems. The same results are similar for the precision and recall as described in Figure 4 and 5. However, the quality gain is slightly lower than most of the existing techniques as showed in Figure 6. This can be explained by the reason that the main goal of developers is not to fix the maximum number the quality issues detected in the system (which was the goal of most of the existing studies). Also, our approach is based on a multi-objective algorithm to find a trade-off between improving the quality and reducing the number of refactorings.

Figure 7 clearly shows that our personalized refactoring approach converges much faster to acceptable refactoring solutions comparing to most of the existing studies. For example, the work of Ouni et al. required at least 20 minutes to converge to a good quality of solutions however our approach was able to recommend good refactoring opportunities within two minutes. One reason of the low execution time of our approach is the number of recommended refactorings as described in Figure 9.

To conclude, our interactive approach provides better results, on average, than existing fully-automated refactoring techniques (answer to RQ2).

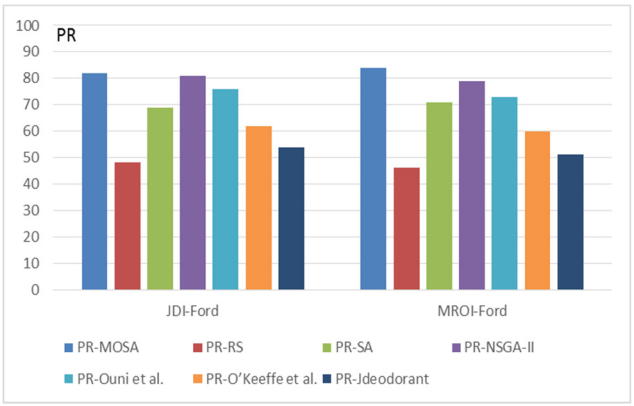

Figure3. Median precision (PR) value for 30 executions on all the two systems with a 95% confidence level (α<5%).

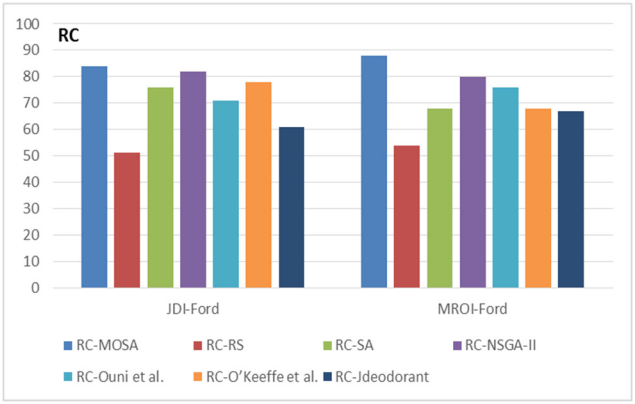

Figure 4. Median recall (RC) value for 30 executions on all the two systems with a 95% confidence level (α<5%).

Results for RQ3. In the first component of the post-study questionnaire, the participants were asked to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following statements: 1. The proposed personalized refactoring technique is a desirable feature in integrated development environments. 2. The reduced number of recommended relevant refactorings may help developers performing every-day design, implementation and maintenance activities.

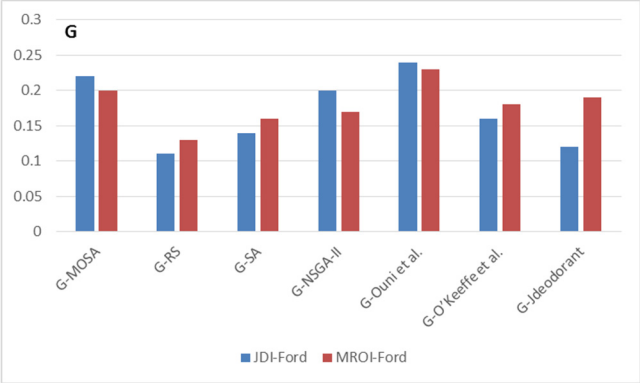

Figure 5. Median quality gain (G) value for 30 executions on all the two systems with a 95% confidence level (α<5%).

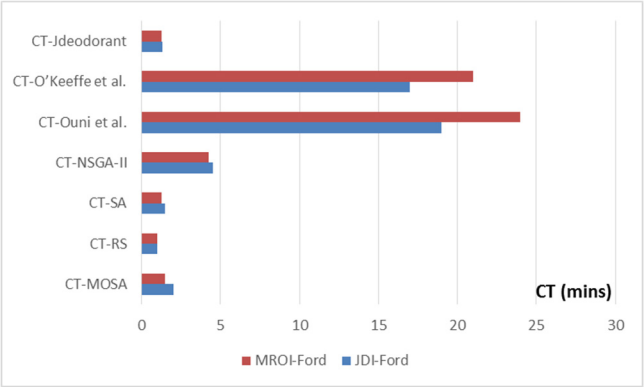

Figure 6.Median execution time (CT) for 30 executions on all the two systems with a 95% confidence level (α<5%).

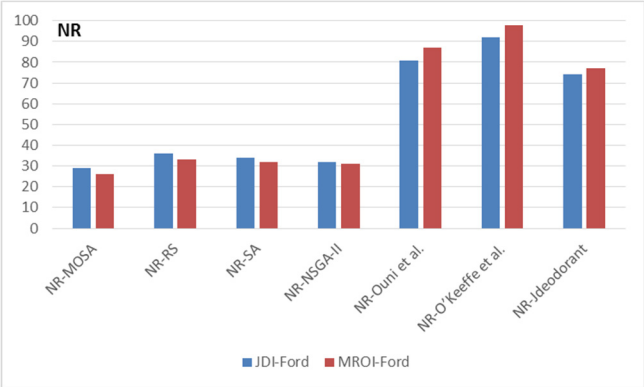

Figure 7.Median number of refactorings (NR) for 30 executions on all the two systems with a 95% confidence level (α<5%).

In the second component of the questionnaire, the subjects were asked to specify the possible usefulness of the suggested refactorings to perform some activities such as quality assurance/assessment, regression testing, effort prediction, code inspection, and features extension. In the third part, we asked the programmers about possible improvements of our personalized refactoring tool.

As described in Figure 7, the agreement of the participants was 4.6 and 4.3 for the first and second statements respectively. This confirms the usefulness of our approach for the software developers. Regarding the possible usefulness to perform some activities, the developers agreed that quality assurance/assessment and features extension are the three main activities where the personalized refactorings could be very helpful with an agreement of more than 4.3.

The three other activities of effort prediction, regression testing and code inspection are considered less relevant for our tool with an agreement of around 3.8. The majority of the programmers we interviewed found that the personalized refactorings give interesting quick advices about possible refactoring opportunities to improve the quality and mainly facilitate extending the design of the system to update recently introduced features.

The remaining questions of the post-study questionnaire were about the benefits and also limitations (possible improvements) of our approach. They found that the personalized refactoring technique is much more efficient than the traditional manual and fully-automated techniques. The programmers considered the use of most of existing manual refactoring techniques as a time-consuming process, and it is more relevant to apply refactorings related to their recent development activities. Most of the participants mention that our personalized approach to refactor the code is much faster than analyzing the long list of recommended refactorings by current techniques. The programmers also highlighted that our personalized approach recommended relevant refactorings to continue improving the quality of some code fragments that they started refactoring them in the past.

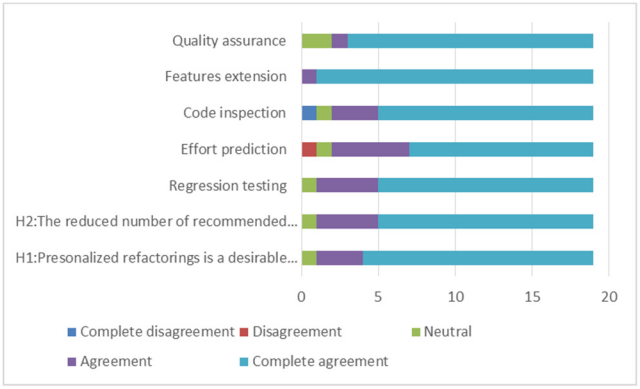

Figure 8. Post-study questionnaire results

The participants also suggested some possible improvements to our personalized refactoring approach. Several participants found that it will be very interesting and helpful to integrate to the tool a new functionality to visualize the design before and after

refactoring. The developers also proposed to explore the area of impact changes analysis as a complementary step of our technique after applying the recommended refactorings.

Threats To Validity. Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We addressed conclusion threats to validity by performing 30 independent simulation runs for each problem instance and statistically analyzing the obtained results using the Wilcoxon rank sum test with a 95% confidence level (α = 5%). However, the parameter tuning of the different optimization algorithms used in our experiments, such as MOSA and NSGA-II, creates another internal threat that we need to evaluate in our future work.

Internal validity is concerned with the causal relationship between the treatment and the outcome. A possible internal threat is related to the variation of relevance and speed between the different groups when using our approach and other tools such as JDeodorant. In fact, our approach may not be the only reason for the superior performance because the participants have different programming skills and familiarity with refactoring tools. To counteract this, we assigned the developers to different groups according to their programming experience so as to reduce the gap between the different groups and we also adapted a counter-balanced design.

Construct validity is concerned with the relationship between theory and what is observed. To evaluate the results of our approach, we selected solutions at the knee point when we compared our approach with existing techniques, but the developers may select a different solution based on their preferences to give different weights to the objectives when selecting the best refactoring solution. The different developers involved in our experiments may have divergent opinions about the recommended refactorings. We considered in our experiments the majority of votes from the developers. External validity refers to the generalizability of our findings. In this study, we performed our experiments on only two industrial systems belonging to different domains and having different sizes to get the feedback from the original developers of these systems.

5 CONCLUSION AND FUTURE WORK

In this work, we described a personalized search based technique for software refactoring to recommend refactorings for programmers based on the history of changes of the system. Our personalized approach helps programmers to take the advantage of search-based refactoring tools with a reasonable execution time or a short list of refactorings to recommend. In fact, the pre-processing phase reduced the search space to explore based on analyzing previous commits and bug reports.

The paper describes an evaluation of the proposed personalized multi-objective approach based on two industrial systems. The obtained results show the outperformance of the proposed technique comparing to existing search-based refactoring approaches and an existing refactoring tool not based on heuristic search, JDeodorant when evaluating the relevance and correctness of recommended refactorings by programmers. Future work may involve the validation of our technique with additional refactoring types.

REFERENCES

[1] Brown, W.H., Malveau, R.C., McCormick, H.W., and Mowbray, T.J.: 'AntiPatterns: refactoring software, architectures, and projects in crisis' (John Wiley & Sons, Inc., 1998. 1998)

[2] Murphy-Hill, E., Parnin, C., and Black, A.P.: 'How we refactor, and how we know it', TSE, 2012, 38, (1), pp. 5-18

[3] Bansiya, J., and Davis, C.G.: 'A hierarchical model for object-oriented design quality assessment', TSE, 2002, 28, (1), pp. 4-17

[4] Ulungu, E., Teghem, J., Fortemps, P., and Tuyttens, D.: 'MOSA method: a tool for solving multiobjective combinatorial optimization problems', Journal of multicriteria decision analysis, 1999, 8, (4), pp. 221

[5] Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., and Ouni, A.: 'Design Defects Detection and Correction by Example', in Editor (Ed.)^(Eds.): 'Book Design Defects Detection and Correction by Example' (2011, edn.), pp. 81-90

[6] O'Keeffe, M., and Ó Cinnéide, M.: 'Search-based refactoring for software maintenance', Journal of Systems and Software, 2008, 81, (4), pp. 502-516

[7] Ouni, A., Kessentini, M., Sahraoui, H., and Boukadoum, M.: 'Maintainability defects detection and correction: a multi-objective approach', Automated Software Engineering, 2012, 20, (1), pp. 47-79

[8] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A.: 'JDeodorant: identification and application of extract class refactorings', in Editor (Ed.)^(Eds.): 'Book JDeodorant: identification and application of extract class refactorings' (2011, edn.), pp. 1037-1039

[9] Piveta, E.K., Hecht, M., Moreira, A., Pimenta, M.S., Araújo, J., Guerreiro, P., and Price, R.T.: 'Avoiding Bad Smells in Aspect-Oriented Software', in Editor (Ed.)^(Eds.): 'Book Avoiding Bad Smells in Aspect-Oriented Software' (Citeseer, 2007, edn.), pp. 81-

[10] Marinescu, C., Marinescu, R., Mihancea, P.F., and Wettel, R.: 'iPlasma: An integrated platform for quality assessment of object-oriented design', in Editor (Ed.)^(Eds.): 'Book iPlasma: An integrated platform for quality assessment of object-oriented design' (Citeseer, 2005, edn.), pp.

[11] Ali Ouni, Marouane Kessentini, Houari A. Sahraoui, Katsuro Inoue, Kalyanmoy Deb: Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study. ACM Trans. Softw. Eng. Methodol. 25(3): 23:1-23:53 (2016)

[12] Meananeatra, P.: 'Identifying refactoring sequences for improving software maintainability', in Editor (Ed.)^(Eds.): 'Book Identifying refactoring sequences for improving software maintainability' (ACM, 2012, edn.), pp. 406-409

[13] Bader Alkhazi, Terry Ruas, Marouane Kessentini, Manuel Wimmer, William I. Grosky: Automated refactoring of ATL model transformations: a search-based approach. MoDELS 2016: 295-304

[14] Bois, B.D., Demeyer, S., and Verelst, J.: 'Refactoring - improving coupling and cohesion of existing code', in Editor (Ed.)^(Eds.): 'Book Refactoring - improving coupling and cohesion of existing code' pp. 144-151

[15] Seng, O., Stammel, J., and Burkhart, D.: 'Search-based determination of refactorings for improving the class structure of object-oriented systems', in Editor (Ed.)^(Eds.): 'Book Search-based determination of refactorings for improving the class structure of object-oriented systems' (ACM, 2006, edn.), pp. 1909-1916

[16] Ali Ouni, Marouane Kessentini, Houari A. Sahraoui, Mounir Boukadoum: Maintainability defects detection and correction: a multi-objective approach. Autom. Softw. Eng. 20(1): 47-79 (2013)

[17] Harman, M., and Tratt, L.: 'Pareto optimal search based refactoring at the design level'. Proc. Proceedings of the 9th annual conference on Genetic and evolutionary computation, London, England2007 pp. Pages

[18] Ben Fadhel, A., Kessentini, M., Langer, P., and Wimmer, M.: 'Search-based detection of high-level model changes', in Editor (Ed.)^(Eds.): 'Book Search-based detection of high-level model changes' (IEEE, 2012, edn.), pp. 212-221

[19] Adnane Ghannem, Ghizlane El-Boussaidi, Marouane Kessentini: On the use of design defect examples to detect model refactoring opportunities. Software Quality Journal 24(4): 947-965 (2016)

[20] Hanzhang Wang, Marouane Kessentini, Ali Ouni: Bi-level Identification of Web Service Defects. ICSOC 2016: 352-368