Personalized Multi-Objective Approach for Refactoring Recommendations

Troh Josselin Dea, Marouane Kessentini and Ali Ouni
Computer and Information Science Department
University of Michigan, USA
firstname@umich.edu

Abstract— Refactoring is an extremely important solution to reduce and manage the growing complexity of software systems. However, maintaining a high-level code quality can be expensive since time and monetary pressures force developers to neglect to improve the quality of their source code. Thus, 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 work, we describe a search based approach to recommend refactorings based on the analysis of the history of changes to maximize the recommended refactorings for 1) recently modified classes, 2) classes containing incomplete refactorings detected in previous releases, and 3) buggy classes identified based on bug reports. The obtained results on 2 industrial projects shows significant improvements of the relevance of recommended refactorings, as evaluated by the original developers of the systems, and much lower execution time comparing to existing search-based refactoring techniques.

Keywords— Search-based software engineering, multi-objective simulated annealing, refactoring, history of changes.

I. 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 functionnalities is termed the “software refactoring problem” [1]. Classically, program refactoring has been studied in the context of technical debt which is a metaphor introduced by Cunningham to point out “not quite right code which we postpone making it right” [2]. An effective strategy in managing technical debt has been to identify and fix quality issues using refactoring. A large portion of existing refactoring tools suggests refactorings that can be used to improve the overall quality of systems without being personalized to the specific needs of programmers. As a result, the number of refactorings to apply is huge and developers spent a long time to find relevant refactorings.

When a high number of refactorings are recommended, manual refactoring becomes error-prone and time-consuming. Murphy-Hill et al. [3] 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. In addition, developers find fully automated refactoring risky because it can introduce bugs or undesired changes [4]. Furthermore, the automated extensive applications of refactorings may radically change the initial design.

To increase the relevance of recommended refactorings, we used the history of changes, in our previous work [5], to identify similar patterns between existing recommendations and previous ones. We also considered several semantic constraints to improve the correctness of recommended refactorings. However, the proposed multi-objective approach still generates a high number of refactorings since it is trying to fix most of the possible quality issues of the system without considering the current needs of the developers. Morales et al. [6] extended our previous work [5] by proposing a search-based approach to consider developer’s task to find the best sequence of refactorings that affects only entities modified by the developers collected using the Eclipse plug-in Mylyn. The results show that the number of antipatterns is reduced by 50% with a low number of refactorings. However, the validation was limited to only the consideration of quality improvements based on QMOOD [7], number of antipatterns and the execution time. It is not clear

how much the relevance of the recommended refactorings is improved based the system developers' perspective after manually inspecting the recommendations. In addition, they did not consider the time dimension when mining the collected developer's tasks. In fact, developers are more interested, in general, to refactor *recently* modified entities. Furthermore, the identification of incomplete refactorings in previous releases was not considered in [6][41][42][43] to recommend new refactorings.

In this paper, we propose a personalized search-based approach for refactoring recommendations. We start from the following observations that:

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][15][44][45][46] identified correlation between bugs and refactoring opportunities, and

3) recent introduced refactorings is an indication of quality issues that should be fixed by the auto-completion of these applied refactorings.

To this end, we propose to reduce the search space of possible refactoring solutions by considering the above three observations then executing a local search algorithm, based on simulated annealing [9], to find the best sequence of refactorings maximizing the quality improvements and minimizing the number of refactored classes based on their ranking after the space reduction phase. In fact, the space reduction phase ranks the potential classes to refactor, identified using a set of antipatterns detection rules [10], based on a measure formalizing the three above observations. Then, the multi-objective simulated annealing algorithm [9] is executed to find the best sequence of refactorings optimizing these two objectives.

We implemented our proposed approach and evaluated it on a set of two industrial systems provided by our industrial partner, the Ford Motor Company. 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 [11] [12] and an existing refactoring tool not based on heuristic search, JDeodorant [13] in terms of 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 fixed quality issues in their daily development activities.

The primary contributions of this paper can be summarized as follows:

- The paper introduces a new way to refactor software systems by reducing the search space to identify the most relevant refactoring recommendations. The proposed technique supports the adaptation of refactoring solutions based on the recent code changes that the developer performed, recent bugs and identified incomplete refactorings.
- The paper presents an evaluation of the proposed personalized multi-objective approach based on two industrial systems. The obtained results confirm the outperformance of the proposed technique comparing to existing search-based refactoring approaches [11] [12] and an existing refactoring tool not based on heuristic search, JDeodorant [13] in terms of the relevance and correctness of recommended refactorings.

The remainder of this paper is structured as follows. Section 2 provides an account of the related work. Section 3 describes our personalized 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.

II. Related Work

Refactoring is the process of improving the code quality of an existing system while preserving its external behavior [14][41][42][43][44][45][46][47][48][49][50][51][52][53][54][55]. The refactoring procedure 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 [14]. These code smells correspond to design practices that have a negative impact on the maintainability, understandability and performance of the software [15]. The study of determining refactoring opportunities has been initially correlated mainly with the improvement of the software architecture. These proposals vary from manual to semi- and fully-automated.

The manual investigation of refactorings can be seen in the early work of Fowler [14] in whish, a broad description of smell symptoms has been given along with a set of suggested operations to apply as remedy for each code smell. Similarly, Wake [16] coupled the detection of code smells to a Refactoring Workbook that details their removal. Later on, studies has emphasized on deploying several metrics to better characterize code smells and thus facilitate their detection, in this context, Mäntylä [17] refined Fowler's definitions of structural anomalies at the source code level in terms of metrics in order to automate their detection and proposed their refactoring based on developers' opinions.

Another interesting and recent study on refactoring has been proposed by Piveta el al. [18] who conducted a study on when refactoring operations are eventually needed when detecting bad smells in the context of aspect-oriented software. Also, Counsell et al. [19] defined smells' refactoring in terms of fan in and fan out-degrees on a dependency graph. The emphasis of using structural metrics instead has given more maturity in better understanding smells taxonomy [20] and so, it has orientated studies towards automating their detection and then correction.

In this context, Meananeatra [21] 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, number of applied changes and number of refactored code fragments. Another tool is proposed, called JDeodorant [13], 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. [10] 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 the elimination of design defects, but also driven by the optimization of the software physical design through increasing software quality attributes. For example, Du Bois et al. [23] 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. [24] 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 [25] 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. Mkaouer et al. [26] extended the range of used metrics to 15 by considering the code refactoring as a many-objective optimization problem. They extended their work to also preserve the domain semantics along with optimizing the software architectural quality through the optimization of QMOOD quality attributes [27].

One major observation extracted from existing studies, is the extensive use of structural code measures in to either defining design defects or describing quality attributes, and so the refactoring process was essentially piloted by enhancing the software architecture without respect to other important factors such as prioritizing buggy source code, suspect of future investigation and conformance with the history of code changes previously done by developers in software previous releases.

To cope with this limitation, recent studies were no longer only limited to structural metrics, but also work on the auto-completion of refactoring activities based on knowledge extracted from target languages, as in [28], Spinellis et al. developed a refactoring browser that restructures classes by tagging their identifiers

along with their location and then clustering them into equivalence classes with respect to C language's namespace and scope extents. Murphy-Hill et al. suggested various studies [29] [30] [31] to assess engineers in their refactoring sessions. In [29] they provided a tool that provides visual support and structural analysis to software engineers in order to help them in better locating refactoring opportunities. Then as well, the authors proposed BeneFactor [30] that can be called during a refactoring session to safely complete a refactoring change for the Java language. They also proposed GhostFactor that helps developers in automatically checking the correctness of any executed code refactoring [31].

In [32], the authors considered software remodularization as an interactive multi-objective optimization problem. In their approach, each remodularization solution is a sequence of clustering operations that present a potential decomposition of entities into modules/packages. The set of best solutions, extracted from the Pareto front, are suggested to developers for evaluation. Then the developer's feedback is used in the next phase of the remodularization process to prune the search space through penalizing solutions that do not satisfy the feedback. Furthermore, the history of code changes, gathered from developers, was mined in [33] to identify code fragments infected with bad smells.

III. Personalized Search-Based Refactoring

In this section, we start first by describing the general process of the proposed algorithm then we discuss the formulation and adaptation of our search algorithm.

A. Approach Overview

The main objective of the proposed technique is to search for the best set of refactorings that may fix most of the design violations for programmers and based on their recent update/changes of the system. Figure 1 describes an overview of the proposed framework.

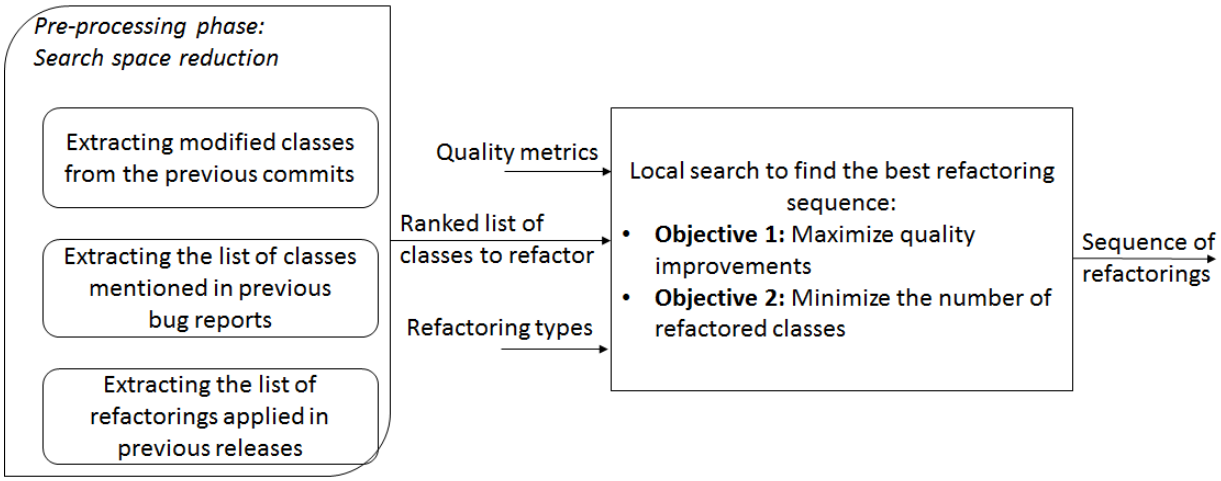

Fig. 1. Approach overview

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 from previous releases or those mentioned in previous bug reports or the classes recently refactored. The classes mentioned in recent commits, are maybe important to refactor since they have a higher 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.

Developers, for example, may introduce bugs because the complexity of the system and its poor design and extendibility. Furthermore, the classes that are refactored recently by the developers but still contains quality issues can be automatically refactored since already programmers expressed an interest in fixing them but did not finish the work due to the time constraints before a new commit or release. The list of applied refactorings in previous releases are detected using our previous work [34]. The outcome of this first phased is a list of ranked classes that could be refactored. The ranking function is based on the three different measures of recently modified classes, recent classes mentioned in bug reports and recent incomplete refactoring activities. The formalization of these measures will be described in the next section.

The outcome of the first phase is used to reduce the search space to find the best refactoring sequence to recommend for developers. Instead of exploring a large search space of fixing most of the identified quality issues on the system, a multi-objective search algorithm is used to focus mainly on refactoring the ranked classes of the first phase. 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 improving the quality, which corresponds to improving the QMOOD (Quality Model for Object-Oriented Design) quality metrics [7] and minimizing the number of refactored classes.

A multi-objective simulated annealing algorithm 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 [27].

The next section will discuss the formalization of our approach and the adaptation of the multi-objective simulated annealing algorithm to our problem.

B. *Problem Formulation and Solution Approach*

We describe in the following subsections the details of the two main components of our framework.

1) *Pre-processing: search space reduction*

The main goal of the pre-processing phase is to rank potential classes for refactoring based on their relevance to the developers. Formally, the ranking function is defined as follows:

$$Rank(c) = \frac{commitf(c) + bugreportsf(c) + refactoringf(c)}{3},$$

where c is the class to rank and *commitf(c)*, *bugreportsf(c)* and *refactoringf(c)* are respectively the functions to estimate the relevance of the class based on previous recent commits, bug reports and incomplete applied refactorings.

The first function *commitf(c)* checks if a class was recently changed. In fact, a class that was modified recently has a higher 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 that 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. Furthermore, we prioritized the classes fixed in recent bug reports. Formally, this function, normalized between [0,1] is defined as:

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

The third function *refactoring (c)* counts the number of antipatterns in the class *c*, refactored in the past, divided by the maximum number of antipatterns per class in the system *S*. The classes that are refactored in the past by developers but still contain antipatterns have a high probability of being relevant refactoring opportunities for developers. The classes that were not refactored in the past by developers will take the value of 0. We used our previous work of refactorings detection [34] to identify the list of classes refactored in previous releases. The third function is formalized and normalized as follows:

$$refactoringf\,(c) = \frac{\#\,antipatterns(c)}{Max(\#\,antipattenrsPerClass(S))}$$

The outcome of this phase is a list of ranked classes based on their relevance for programmers as refactoring opportunities. This outcome is used as input to the multi-objective simulated annealing algorithm that will be described in the next section.

2) Simulated Annealing Adaptation

Due to the reduce search space after the pre-processing phase of filtering refactoring opportunities based on recent changes, we used a local search algorithm based on multi-objective simulated annealing (MOSA) . In the next sections, we described the three main steps of adaptation of MOSA [42] to our problem.

a) 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, Extract method. More details about these refactorings can be found in [43].

The size of a solution, i.e. the vector's length is randomly chosen between upper and lower bound values. The determination of these two bounds is similar to the problem of bloat control in genetic programming where the goal is to identify the tree size limits. Since the number of required refactorings depends mainly on the size of the target system, we performed, for each target project, several trial and error experiments using

the HyperVolume (HP) performance indicator [35] to determine the upper bound after which, the indicator remains invariant. For the lower bound, it is arbitrarily chosen. The size of the solution will depend on the list of classes identified by the pre-processing step to reduce the search space. The experiments section will specify the upper and lower bounds used in this study.

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)

b) Fitness functions

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

Minimize the number of refactored classes: Due to the limited time frame dedicated to refactoring software systems by programmers, it is important to find the most critical and relevant refactorings to apply rather than trying to fix every detected refactoring opportunity. To this end, we formulated the fitness function as the number of modified actors/code elements (packages, classes, methods, attributes) by the generated refactorings solution.

$$f_1(x) = \sum_{i-1}^{n} \#code\_elements(R_i, x)$$

where x is the solution to evaluate, n is the number of refactorings in the solution x and *#code_elements* is a function that counts the number of modified code elements in a refactoring. Any solution with refactorings being performed on the same code elements will have better (lower) fitness value for this objective. Such

definition of the objective is in favor of code locality since it encourages refactoring the same code fragment, as developers prefer to refactor the specific elements with which they are most familiar instead of applying scattered changes throughout the whole system.

The proposed fitness function is different from that employed in our previous work [27] where only the number of applied refactorings are counted. In fact, each refactoring type may have a different impact on the system in terms of number of code changes it engenders, something that can be identified using our new formulation.

Maximize software quality and refactorings relevance: The second fitness function is defined as follows:

$$f_2(x) = \frac{\frac{\sum_{i-1}^{n} Rank(class(R_i, x))}{\sum_{j=1}^{r} Rank(c_j)} + \frac{\sum_{i=1}^{6} QA_i(S)}{6}}{2}$$

The first component of the function maximizes the ranking score of the selected classes to be refactored based on the pre-processing phase. The rank function was already described in the previous section. The second component of the function is based on the use of QMOOD [7] where QA_i is the quality attribute number *i* being calculated based on the returned structural metrics from the system *S*. Several work have used structural measures to evaluate the quality of software projects [36]. One of the widely used models is proposed by Bansiya et al. [7] , called QMOOD, which is based on six different quality attributes such as reusability, felexibility, understandability, functionality, extendibility and effectiveness. These quality attributes are defined using a set of strucutal quality metrics. More details about the QMOOD model can be found in [43].

Since it may not be sufficient to consider structural metrics, we used the design coherence measures of our previous work to ensure that every refactoring solution preserves the semantics of the design. More details can be found in [27].

c) 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. For example, to apply the refactoring operation *move method* a number of necessary pre-conditions should be satisfied such as the method and the source and target classes should exists. A post-condition example is to check that the method exists and was moved to the target class and did not exist anymore in the source class. More details about the adapted pre- and post-conditions for refactorings can be found in [37]. We also apply a repair operator to randomly select new refactorings to replace those creating conflicts.

IV. EVALUATION

This section describes the experimental results of our approach to find relevant and correct set of refactorings.

We performed several experiments on two industrial projects provided by the IT department at the Ford Motor Company. We compared the results of our approach on these systems with several exisiting refactoring techniques based on a set of 30 different runs.

A. Research Questions and Evaluation Metrics

To evaluate and compare the performance and relevance of the recommended refactoring by our personalized 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:

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

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

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 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-atuomate 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. [41] 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.

All these existing techniques are fully-automated and do not consider the different heuristics used in our approach for the relevance of recommended refactorings. We did not consider the recent work of Morales et al. [6] in our experiments since the tool is not available. 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.

B. *Studied Projects*

In order to get feedback from the original developers of a system, we considered in our experiments two large industrial projects provided by our industrial partner, the Ford Motor Company.

The first project is a marketing return on investment tool, called MROI, used by the marketing department of Ford 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 Ford Motor Company to create the best schedule of orders from the dealers based on many business constraints. This system is also used by the Ford Motor 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 Ford 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.

TABLE II. THE EVLUATED INDUSTRIAL PROJECTS

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

C. Scenarios

Our study involved 19 software developers from the Ford Motor 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 about 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 similatiry 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.

D. *Experimental Setting and Statistical Test Methods*

The parameters tuning is one of the important steps when comparing and evaluating different computational search algorithms [38]. To this end we used different population sizes of the used algorithms to evaluate their performance ranging from 100, 200, 300 and 500 indviduals 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 [38] 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% ($\alpha < 5\%$).

To evaluate the difference in magnitude, we used the Vargha-Delaney A measure [39] 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 than 0.88.

We used in our experiments, eight different types of code smells: 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 [12] [33].

The upper and lower bounds on the chromosome length used in this study are set to 10 and 250 respectively. We performed several trial and error experiments where we assess the average performance of our algorithm using the HV (hypervolume) performance indicator while varying the size limits between 10 and 500 operations. 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.

E. 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-Ford project and the *RR* score is 87% for the second system MROI-Ford. Thus, it is clear the obtained results are

not dependent to 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.

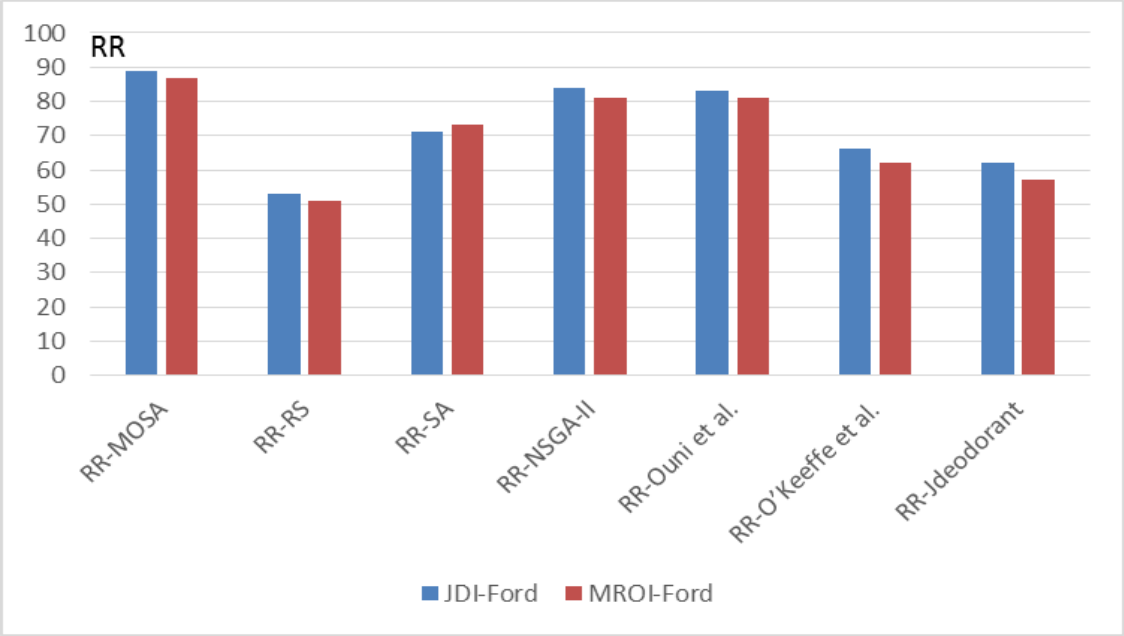

Fig. 2. Median refactoring relevance (RR) value for 30 executions on all the two systems based on the different studied refactoring approaches with a 95% confidence level (α<5%).

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 more complete list of refactorings comparing to the manually recommended operations by the programmers due to the time-consuming process of code refactoring. In addition, 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.

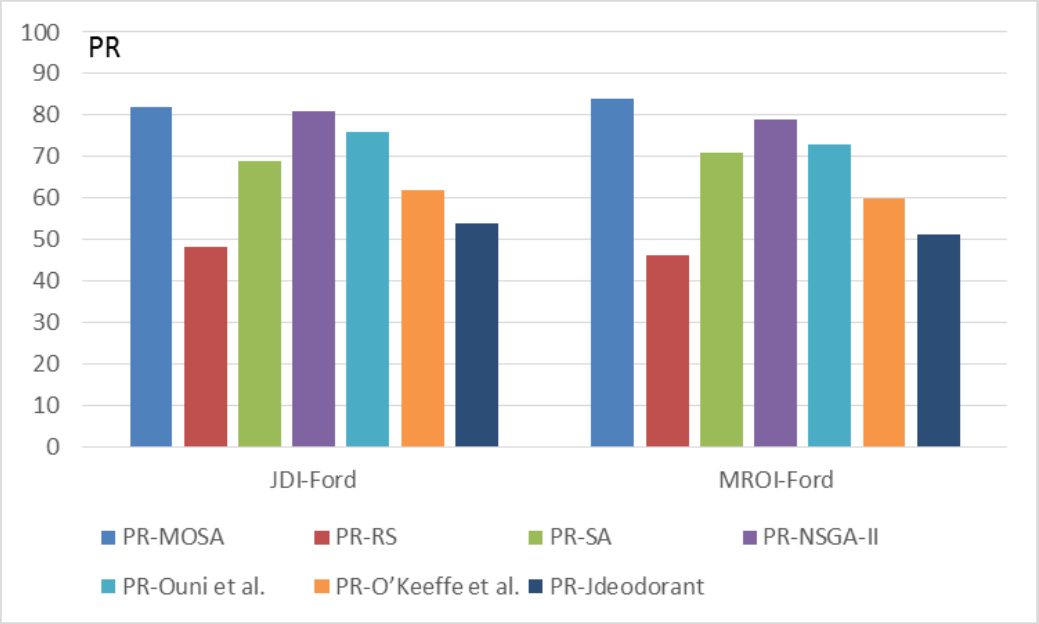

Fig. 3. Median precision (PR) value for 30 executions on all the two systems based on the different studied refactoring approaches with a 95% confidence level (α<5%).

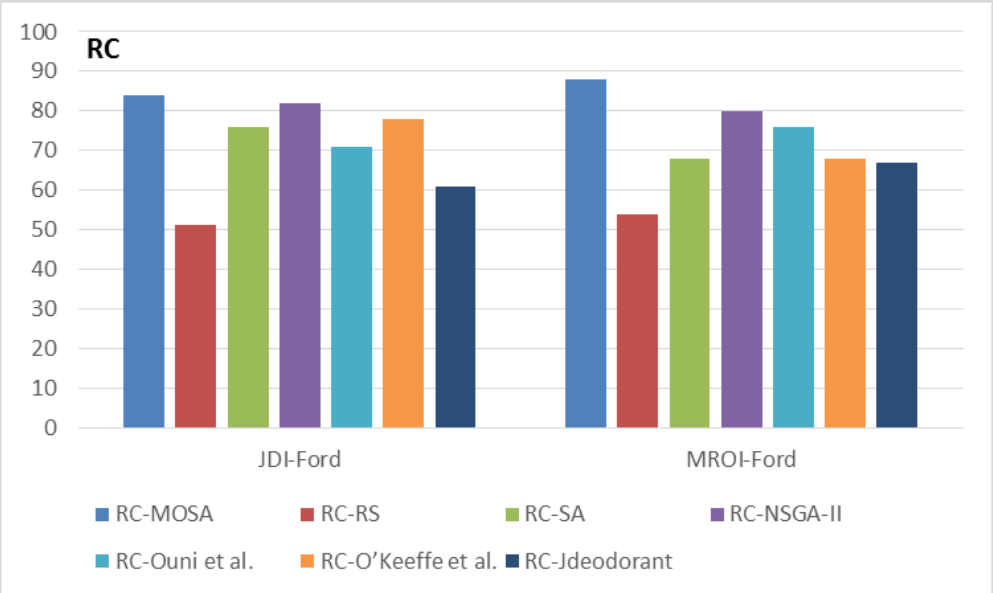

Fig. 4. Median recall (RC) value for 30 executions on all the two systems based on the different studied refactoring approaches with a 95% confidence level (α<5%).

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.

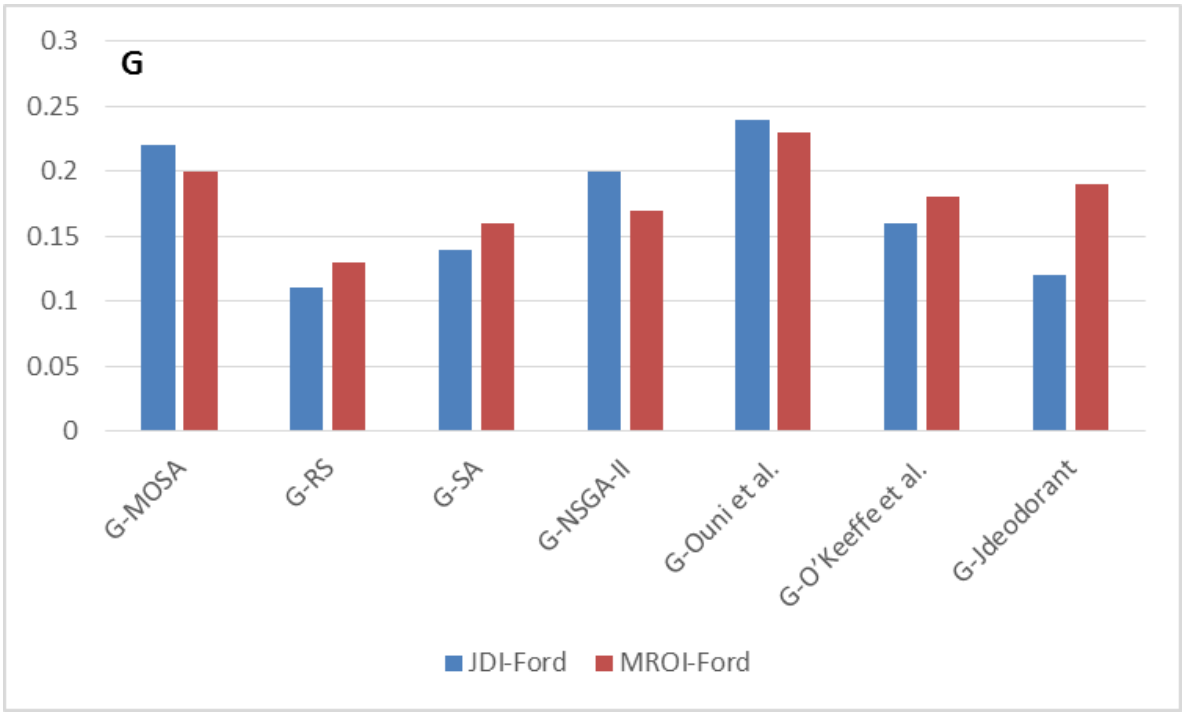

Fig. 5. Median quality gain (G) value for 30 executions on all the two systems based on the different studied refactoring approaches with a 95% confidence level (α<5%).

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 existing techniques as showed in Figure 6. This is 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 existing studies). In addition, 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 converge much faster to acceptable refactoring solutions comparing to most of existing studies. For example, the work of Ouni et al. required at least 20 minutes to converge to good quality of solutions however our approach was able to recommend good refactoring opportunities within two minutes. One the reasons 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).

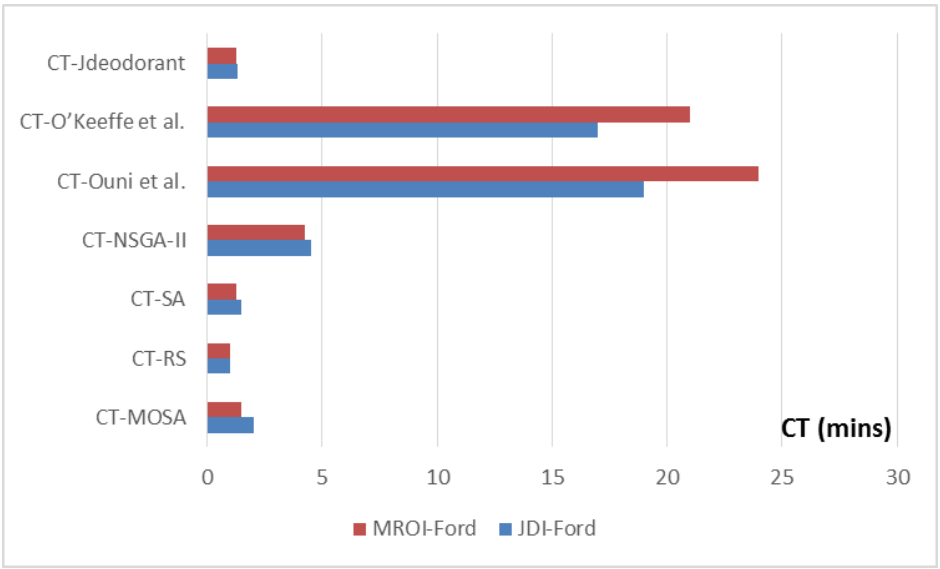

Fig. 6. Median execution time (CT) for 30 executions on all the two systems based on the different studied refactoring approaches with a 95% confidence level (α<5%).

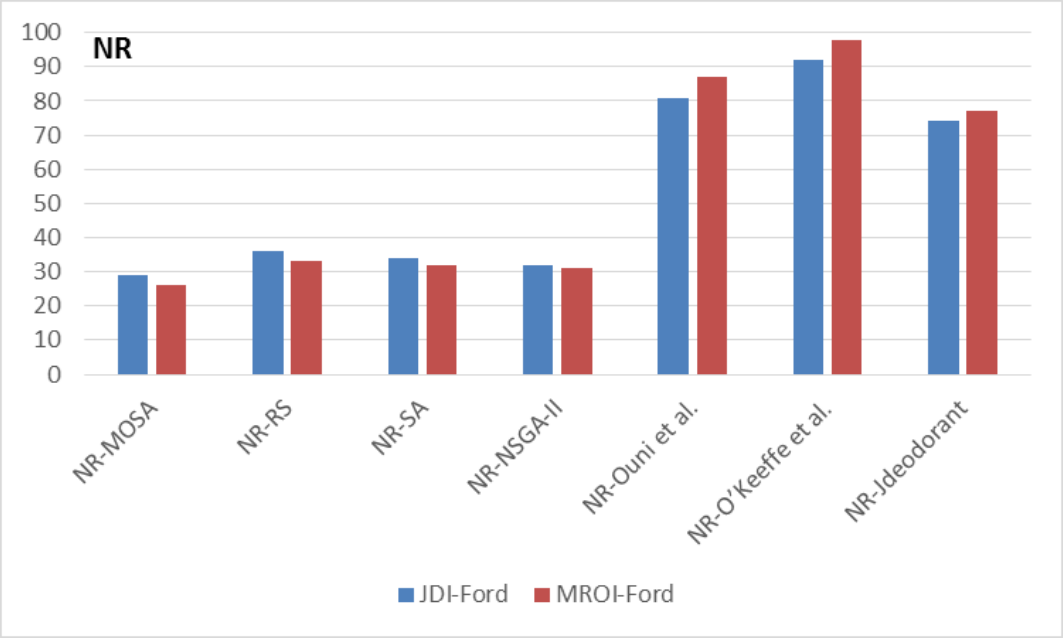

Fig. 7. Median number of refactorings (NR) for 30 executions on all the two systems based on the different studied refactoring approaches with a 95% confidence level (α<5%).

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.

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.

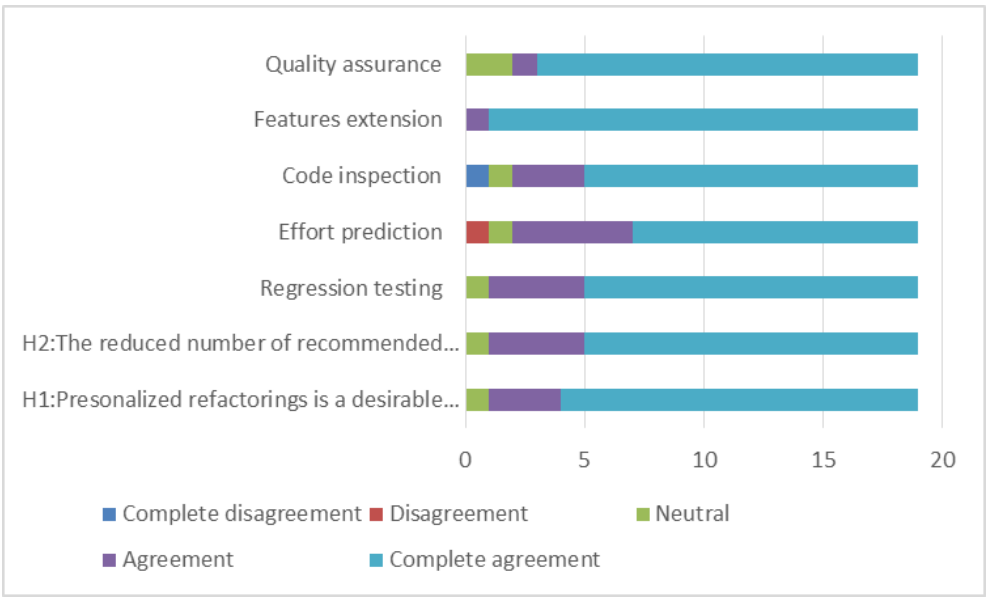

Fig. 8. Post-study questionnaire results

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.

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.

V. 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, bug reports and incomplete refactorings.

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 [11] [12] and an existing refactoring tool not based on heuristic search, JDeodorant [13] when evaluating the relevance and correctness of recommended refactorings by programmers.

Future work involves validating our technique with additional refactoring types, programming languages and potential users to evaluate the performance of our methodology. Furthermore, we plan to extend the approach by automating the test and verification of applied refactorings.

ACKNOWLEDGMENT

The authors acknowledge and thank the subjects who participated in the experiments for their valuable time, comments and suggestions for the improvement of this work. This work is funded by the Ford Motor Company.

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] Cunningham, W.: 'The WyCash portfolio management system', ACM SIGPLAN OOPS Messenger, 1993, 4, (2), pp. 29-30

[3] Murphy-Hill, E., Parnin, C., and Black, A.P.: 'How we refactor, and how we know it', IEEE Transactions on Software Engineering, 2012, 38, (1), pp. 5-18

[4] Soares, G., Gheyi, R., and Massoni, T.: 'Automated behavioral testing of refactoring engines', IEEE Transactions on Software Engineering, 2013, 39, (2), pp. 147-162

[5] Ouni, A., Kessentini, M., Sahraoui, H., and Hamdi, M.S.: 'The use of development history in software refactoring using a multi-objective evolutionary algorithm'. Proc. Proceedings of the 15th annual conference on Genetic and evolutionary computation, Amsterdam, The Netherlands2013 pp. Pages

[6] Morales, R., Soh, Z., Khomh, F., Antoniol, G., and Chicano, F.: 'On the use of developers' context for automatic refactoring of software anti-patterns', Journal of Systems and Software, 2016

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

[8] Kim, M., Zimmermann, T., and Nagappan, N.: 'A field study of refactoring challenges and benefits', in Editor (Ed.)^(Eds.): 'Book A field study of refactoring challenges and benefits' (ACM, 2012, edn.), pp. 50

[9] 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

[10] Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., and Ouni, A.: 'Design Defects Detection and Correction by Example', ICPC2011, pp. 81-90

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

[12] 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

[13] 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

[14] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D.: 'Refactoring: Improving the design of existing programs', in Editor (Ed.)^(Eds.): 'Book Refactoring: Improving the design of existing programs' (Addison-Wesley Reading, 1999, edn.), pp.

[15] Yamashita, A.: 'Assessing the capability of code smells to support software maintainability assessments: Empirical inquiry and methodological approach', 2012

[16] Wake, W.C.: 'Refactoring workbook' (Addison-Wesley Professional, 2004. 2004)

[17] Mantyla, M., Vanhanen, J., and Lassenius, C.: 'A taxonomy and an initial empirical study of bad smells in code', in Editor (Ed.)^(Eds.): 'Book A taxonomy and an initial empirical study of bad smells in code' (IEEE, 2003, edn.), pp. 381-384

[18] 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-

[19] Counsell, S., Hierons, R.M., Hamza, H., Black, S., and Durrand, M.: 'Is a strategy for code smell assessment long overdue?', in Editor (Ed.)^(Eds.): 'Book Is a strategy for code smell assessment long overdue?' (ACM, 2010, edn.), pp. 32-38

[20] 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.

[21] 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

[22] Tahvildari, L.a., and Kontogiannis, K.: ‘A metric-based approach to enhance design quality through meta-pattern transformations’, in Editor (Ed.)^(Eds.): ‘Book A metric-based approach to enhance design quality through meta-pattern transformations’ (2003, edn.), pp. 183-192

[23] 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’ (2004, edn.), pp. 144-151

[24] 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

[25] 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

[26] Mkaouer, M.W., Kessentini, M., Bechikh, S., Deb, K., and Ó Cinnéide, M.: ‘High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III’, GECCO2014, (ACM, 2014, edn.), pp. 1263-1270

[27] Mkaouer, M.W., Kessentini, M., Bechikh, S., Cinnéide, M.Ó., and Deb, K.: ‘On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach’, Empirical Software Engineering, 2015, pp. 1-43

[28] Spinellis, D.: ‘CScout: A refactoring browser for C’, Science of Computer Programming, 2010, 75, (4), pp. 216-231

[29] Murphy-Hill, E.: ‘Improving usability of refactoring tools’, in Editor (Ed.)^(Eds.): ‘Book Improving usability of refactoring tools’ (ACM, 2006, edn.), pp. 746-747

[30] Ge, X., and Murphy-Hill, E.: ‘BeneFactor: a flexible refactoring tool for eclipse’. Proc. Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, Portland, Oregon, USA2011 pp. Pages

[31] Ge, X., and Murphy-Hill, E.: ‘Manual refactoring changes with automated refactoring validation’. Proc. Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India2014 pp. Pages

[32] Bavota, G., Carnevale, F., De Lucia, A., Di Penta, M., and Oliveto, R.: ‘Putting the developer in-the-loop: an interactive GA for software re-modularization’: ‘Search Based Software Engineering’ (Springer, 2012), pp. 75-89

[33] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., and Poshyvanyk, D.: 'Detecting bad smells in source code using change history information', in Editor (Ed.)^(Eds.): 'Book Detecting bad smells in source code using change history information' (IEEE, 2013, edn.), pp. 268-278

[34] ben Fadhel, A., Kessentini, M., Langer, P., and Wimmer, M.: 'Search-based detection of high-level model changes', ICSME2012 (IEEE, 2012, edn.), pp. 212-221

[35] Zitzler, E., Brockhoff, D., and Thiele, L.: 'The hypervolume indicator revisited: On the design of Pareto-compliant indicators via weighted integration', in Editor (Ed.)^(Eds.): 'Book The hypervolume indicator revisited: On the design of Pareto-compliant indicators via weighted integration' (Springer, 2007, edn.), pp. 862-876

[36] Chidamber, S.R., and Kemerer, C.F.: 'A metrics suite for object oriented design', Software Engineering, IEEE Transactions on, 1994, 20, (6), pp. 476-493

[37] Opdyke, W.F.: 'Refactoring object-oriented frameworks', University of Illinois at Urbana-Champaign, 1992

[38] Arcuri, A., and Fraser, G.: 'Parameter tuning or default values? An empirical investigation in search-based software engineering', Empirical Software Engineering, 2013, 18, (3), pp. 594-623

[39] Vargha, A., and Delaney, H.D.: 'A critique and improvement of the CL common language effect size statistics of McGraw and Wong', Journal of Educational and Behavioral Statistics, 2000, 25, (2), pp. 101-132

[40] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., and Wesslén, A.: 'Experimentation in software engineering' (Springer Science & Business Media, 2012. 2012)

[41] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, Kalynmoy Deb, 2015. Multi-criteria Code Refactoring using Search-Based Software Engineering: An Industrial Case Study. ACM Trans. Softw. Eng. Methodol. 9, 4, Article 39

[42] Jeffery Shelburg, Marouane Kessentini, Daniel R. Tauritz: Regression Testing for Model Transformations: A Multi-objective Approach. SSBSE 2013: 209-223

[43] Mkaouer, M.W., Kessentini, M., Bechikh, S. et al. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empir Software Eng (2015). doi:10.1007/s10664-015-9414-4

[44] Sahin, Dilan, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. "Code-Smell Detection as a Bilevel Problem", ACM Transactions on Software Engineering and Methodology, 2014.

[45] Mkaouer, Mohamed Wiem, Marouane Kessentini, Mel Ó Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. "A robust multi-objective approach to balance severity and importance of refactoring opportunities", Empirical Software Engineering, 2016.

[46] Mansoor, Usman, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm", Software Quality Journal, 2015.

[47] Mkaouer, M.W., Kessentini, M., Bechikh, S., Deb, K., and Ó Cinnéide, M.: 'Recommendation system for software refactoring using innovization and interactive dynamic optimization'. ASE 2014 pp. 331-336

[48] Kalboussi S, Bechikh S, Kessentini M, Said LB. Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents. In International Symposium on Search Based Software Engineering 2013 Aug 24 (pp. 245-250). Springer, Berlin, Heidelberg.

[49] Ouni, Ali, Marouane Kessentini, and Houari Sahraoui. "Search-based refactoring using recorded code changes." In Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, pp. 221-230. IEEE, 2013.

[50] Bechikh, Slim, Marouane Kessentini, Lamjed Ben Said, and Khaled Ghédira. "Chapter four-preference incorporation in evolutionary multiobjective optimization: A survey of the state-of-the-art." Advances in Computers 98 (2015): 141-207.

[51] Boussaa, Mohamed, Wael Kessentini, Marouane Kessentini, Slim Bechikh, and Soukeina Ben Chikha. "Competitive coevolutionary code-smells detection." In International Symposium on Search Based Software Engineering, pp. 50-65. Springer, Berlin, Heidelberg, 2013.

[52] Kessentini, Marouane, Houari Sahraoui, Mounir Boukadoum, and Manuel Wimmer. "Search-based design defects detection by example." In International Conference on Fundamental Approaches to Software Engineering, pp. 401-415. Springer, Berlin, Heidelberg, 2011.

[53] Kessentini, Marouane, Arbi Bouchoucha, Houari Sahraoui, and Mounir Boukadoum. "Example-based sequence diagrams to colored petri nets transformation using heuristic Search." Modelling Foundations and Applications (2010): 156-172.

[54] Kessentini, Marouane, Philip Langer, and Manuel Wimmer. "Searching models, modeling search: On the synergies of SBSE and MDE." In Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering, pp. 51-54. IEEE Press, 2013.

[55] Kessentini, Marouane, Manuel Wimmer, Houari Sahraoui, and Mounir Boukadoum. "Generating transformation rules from examples for behavioral models." In Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications, p. 2. ACM, 2010.