Interactive and Dynamic Multi-Objective Software Refactoring Recommendations

Vahid Alizadeh, Marouane Kessentini, Wiem Mkaouer, Mel Ocinneide, Ali Ouni and Yuanfang Cai

Abstract—Successful software products evolve through a process of continual change. However, this process may weaken the design of the software and make it unnecessarily complex, leading to significantly reduced productivity and increased fault-proneness. Refactoring improves the software design while preserving overall functionality and behavior, and is an important technique in managing the growing complexity of software systems. Most of the existing work on software refactoring uses either an entirely manual or a fully automated approach. Manual refactoring is time-consuming, error-prone and unsuitable for large-scale, radical refactoring. On the other hand, fully automated refactoring yields a static list of refactorings which, when applied, leads to a new and often hard to comprehend design. Furthermore, it is difficult to merge these refactorings with other changes performed in parallel by developers. In this paper, we propose a refactoring recommendation approach that dynamically adapts and interactively suggests refactorings to developers and takes their feedback into consideration. Our approach uses NSGA-II to find a set of good refactoring solutions that improve software quality while minimizing the deviation from the initial design. These refactoring solutions are then analyzed to extract interesting common features between them such as the frequently occurring refactorings in the best non-dominated solutions. Based on this analysis, the refactorings are ranked and suggested to the developer in an interactive fashion as a sequence of transformations. The developer can approve, modify or reject each of the recommended refactorings, and this feedback is then used to update the proposed rankings of recommended refactorings. After a number of introduced code changes and interactions with the developer, the interactive NSGA-II algorithm is executed again on the new modified system to repair the set of refactoring solutions based on the new changes and the feedback received from the developer. We evaluated our approach on a set of eight open source systems and two industrial projects provided by an industrial partner. Statistical analysis of our experiments shows that our dynamic interactive refactoring approach performed significantly better than four existing search-based refactoring techniques and one fully-automated refactoring tool not based on heuristic search.

Index Terms—Search-based software engineering, Refactoring, interactive optimization, software quality.

1 Introduction

Successful software products evolve through a process of continual change. However, this process may weaken the design of the software and make it unnecessarily complex, leading to significantly reduced productivity, increased fault-proneness and cost of maintenance, and has even led to projects being canceled. Many studies report that software maintenance activities consume up to 90% of the total cost of a typical software project. It has also been shown that software developers typically spend around 60% of their time in understanding the code they are maintaining [1].

Clearly, software developers need better ways to manage and reduce the growing complexity of software systems and improve their productivity. The standard solution is refactoring, which involves improving the design structure of the software while preserving its functionality [2]. There has been much work done on various techniques and tools for software refactoring [2], [3], [4], [5] and these approaches can be classified into three main categories: *manual*, *semi-automated* and *fully-automated* approaches, as outlined below.

In manual refactoring, the developer refactors with no tool support at all, identifying the parts of the program that require attention and performing all aspects of the code transformation by hand. It may seem surprising that a developer would eschew the use of tools in this way, but Murphy-Hill et al. [6] found in their empirical study of the developers usage of the Eclipse refactoring tooling that in almost 90% of cases the developers performed refactorings manually and did not use any automated refactoring tools. Kim et al. [7] confirmed this observation, finding that the interviewed developers from Microsoft preferred to perform refactoring manually in 86% of cases. In spite of its apparent popularity, manual refactoring is very limited however; several studies have shown that manual refactoring is error-prone, time-consuming, not scalable and not useful for radical refactoring that requires an extensive application of refactorings to correct unhealthy code [8].

By semi-automated refactoring, we refer to the situation where a developer uses the standard refactoring tooling available in IDEs such as Eclipse and Netbeans to apply the refactorings they deem appropriate. Murphy-Hill et al. [6] analyzed data collected from 13,000 Java developers using the Eclipse IDE over a 9-month period, finding that the

- *Vahid Alizadeh, Marouane Kessentini, Wiem Mkaouer are with the University of Michigan, MI,USA.*
  E-mail: {alizadeh,marouane,mmkaouer}@umich.edu
- *Mel Ocinneide is with the University College Dublin, Dublin, Ireland.*
  E-mail: mel.ocinneide@ucd.ie
- *Ali Ouni is with ETS, Montreal, Canada.*
  E-mail: ouniaali@gmail.com
- *Yuanfang Cai is with Drexel University, USA.*
  E-mail: yfcai@cs.drexel.edu

0000–0000/00/$00.00 © 201X IEEE Published by the IEEE Computer Society

trivial Rename refactoring accounted for almost 72% of the refactorings performed, while the combination of Rename, Extract Method/Variable and Move accounted for 89.3% of the total number of refactorings performed.

In fully-automated refactoring, a search-based process is employed to find an entire refactoring sequence that improves the program in accordance with the employed fitness function (involving e.g., code smells, software quality metrics etc.). This approach is appealing in that it is a complete solution and requires little developer effort, but it suffers from several serious drawbacks as well. Firstly, the recommended refactoring sequence may change the program design radically and this is likely to cause the developer to struggle to understand the refactored program [9]. Secondly, it lacks flexibility since the developer has to either accept or reject the entire refactoring solution. Thirdly, it fails to consider the developer perspective, as the developer has no opportunity to provide feedback on the refactoring solution as it is being created. Furthermore, as development must halt while the refactoring process executes, fully-automated refactoring methods are not useful for floss refactoring where the goal is to maintain good design quality while modifying existing functionality. The developers have to accept the entire refactoring solution even though they prefer, in general, step-wise approaches where the process is interactive and they have control of the refactorings being applied [10].

In light of the discussion above, we propose an approach to refactoring recommendation that (1) provides refactoring-centric interaction, (2) enables refactoring and development to proceed in parallel and (3) collects information in a non-intrusive manner that can be used to inform dynamically the refactoring process. We postulate that enabling the developer to interact with the refactoring solution is essential both to creating a better refactoring solution, and to creating a solution that the developer understands and can work with.

We propose that this interaction should be centered on refactorings, which are of direct interest to a developer, rather than code smells or software quality metrics, which have been found not to be strong drivers of the refactoring process in practice [11, 12]. Refactoring and development must be allowed to proceed in parallel, as this is part of test-driven development [13] and the Agile approach to software development in general [14]. Thus the developer can continue to extend the program with new functionality or bug fixes while the refactoring recommendation process executes. Finally, any development carried out is used where possible to improve the refactoring recommendations, e.g., the developer is more likely to value refactorings that affect recently updated code.

Our goal is to present the developer with few refactorings at a time, allowing them to accept / reject/ modify each refactoring as they see it. Thus, developers are not forced to either accept or run the entire refactoring operations or reject them and the developers may not control the number the applied refactorings. In our approach, the developers can apply operations to the extent that they want. Finding a refactoring solution is a naturally multi-objective problem, so there is not one single ”best” solution, rather there is a set of non-dominated solutions, the so-called Pareto front [15].

In this paper, we use the multi-objective evolutionary algorithm NSGA-II [15] to create the Pareto front, using a fitness function that aims to improve software quality metrics while maintaining design coherence and reducing the number of recommended refactorings. The question we face is how to choose one solution from this front to present to the developer? The traditional approach is to seek a ”knee point” on the front, but this ignores the fact that developers have their own refactoring priorities and may prefer a refactoring solution elsewhere on the front. To this end, *we propose, for the first time in search-based software refactoring, the use of innovization (innovation through optimization) [16] to analyze and explore the Pareto front interactively and implicitly with the developer.* Innovization is a technique that seeks interesting commonalities among the solutions of the Pareto front with the aim of developing a deeper understanding of the problem.

Our innovization algorithm starts by finding the most frequently-occurring refactorings among the set of non-dominated refactoring solutions. Based on this analysis, a complete refactoring solution is chosen from the front that best matches the most frequently-occurring refactorings, i.e., one that best represents the entire front in some sense. The recommended refactorings are then ranked and suggested to the developer one by one.

The developer can approve, modify or reject each suggested refactoring. Each such action by the developer is fed back into the search process. For example, if the developer rejects a refactoring, the search process will subsequently avoid this refactoring in creating new solutions. After the software has been changed to some degree, i.e. the developer has changed it by adding new functionality, fixing some bugs or applying some refactorings and/or has provided feedback by rejecting a number of refactorings, NSGA-II will continue to execute in the new modified context to repair the set of good refactoring solutions based on the updated code and the feedback received from the developer. The feedback received from the developers will be also used as a set of new constraints to consider for the next iterations of NSGA-II. The algorithm will avoid, for example, including rejected refactorings by the developer when generating new solutions or repairing existing ones. However, the algorithm is not based on simply discarding all refactoring suggestions rejected by developer since adding new constraints to reduce the search space may make the current recommended refactoring solutions invalid.

We implemented our proposed approach and evaluated it on a set of eight open source systems and two industrial systems provided by our industrial partner, the Ford Motor Company. Statistical analysis of our experiments showed that our proposal performed significantly better than four existing search-based refactoring approaches [17, 18, 19] [20] and an existing refactoring tool not based on heuristic search, JDeodorant [21]. In our qualitative analysis, we found that the software developers who participated in our experiments confirmed the relevance of the suggested refactorings and the flexibility of the tool in modifying and adapting the suggested refactorings.

This paper builds on our previous work [22] extending it in several ways: (1) the interaction mechanism is improved, we define a new ranking function and different algorithm

to repair non-dominated solutions after interactions with developers, (2) ten software applications are studied rather than five, (3) the number of participants in the experiments is doubled from 11 to 22, (4) an entirely new set of experimental results are presented and analyzed in far greater detail, (5) a comparison with a larger set of existing refactoring techniques is included.
It also extends our previous study [9] where we proposed a fully-automated, multi-objective approach to find the best refactoring solutions that improve software quality metrics and reduce the number of recommended refactorings. In [9], we did not consider any developer interaction (fully-automated approach) and did not update/repair refactoring solutions based on new code changes introduced by developers. A recent study [45] extended our previous work [22] to propose an interactive search based approach for refactoring recommendations. The developers have to specify a desired design at the architecture level then the proposed approach try to find the relevant refactorings that can generate a similar design to the expected one. In our work, we do not consider the use of a desired design, thus developers are not required to manually modify the current architecture of the system to get refactoring recommendations.
The primary contributions of this paper can be summarized as follows:

1) The paper introduces a novel interactive way to refactor software systems using innovization and interactive dynamic multi-objective optimization. The proposed technique supports the adaptation of refactoring solutions based on developer feedback while also taking into account other code changes that the developer may have performed in parallel with the refactoring activity.
2) We propose an implicit exploration of the Pareto front of non-dominated solutions based on our novel interactive approach that can help software developers to use multi-objective optimization for software engineering problems, avoiding the necessity for manual exploration of the Pareto front to find the best trade-off between the objectives.
3) The paper reports the results of an empirical study on an implementation of our approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing refactoring techniques based on a benchmark of eight open source systems and two industrial projects. The paper also evaluates the relevance and usefulness of the suggested refactorings for software developers in improving the quality of their systems.

The remainder of this paper is structured as follows. Section 2 presents the relevant background details. Section 3 describes our novel approach to interactive code refactoring while the results obtained from our experiments are presented and discussed in Section 4 and 5. Threats to validity are discussed in Section 7. Section 7 provides an account of related work. Finally, in Section 8, we summarize our conclusions and present some ideas for future work.

2 BACKGROUND

In this section, we describe the required background to understand the proposed approach. First, we give an overview about software refactoring. Then, several definitions related to interactive and dynamic multi-objective optimization are described.

2.1 Software Refactoring

Refactoring is defined as the process of improving the code after it has been written by changing its internal structure without changing its external behavior. The idea is to reorganize variables, classes and methods to facilitate future adaptations and enhance comprehension. This reorganization is used to improve different aspects of the software quality such as maintainability, extensibility, reusability, etc. Some modern Integrated Development Environments (IDEs), such as Eclipse, Netbeans, provide support for applying the most commonly used refactorings, e.g., move method, rename class, etc.
In order to identify which parts of the source code need to be refactored, most of the existing work relies on the notion of bad smells (e.g., Fowler's textbook [23]), also called design defects or anti-patterns. In this paper, we assume that code smells have been already detected, and need to be corrected. Typically, code smells refer to design situations that adversely affect the development of the software. When applying refactorings to fix design defects, software metrics can be used as an overall indication of the quality of the new design. For instance, high intra-class cohesion and low inter-class coupling usually indicate a high-quality system.

2.2 Interactive and Dynamic Evolutionary Multi-Objective Optimization

In this section, we give a brief overview about two important aspects in the Evolutionary Multi-objective Optimization (EMO) [50] paradigm related to the: (1) Interaction with the user and (2) Dynamicity of the problem.
Interacting with the human user means allowing the user to inject his/her preferences into the computational search algorithm and then using these preferences to guide the search process. To express his/her preferences, the user needs some preference modeling tools. The most commonly used ones are [50]:

- *Weights*: Each objective is assigned a weighting coefficient expressing its importance. The larger the weight is, the more important the objective is.
- *Solution ranking*: The user is provided with a sample of solutions (a subset of the current population) and is invited to perform comparisons between pairs of equally-ranked solutions in order to differentiate between solutions that the fitness function regards as equal.
- *Objective ranking*: Pairwise comparisons between pairs of objectives are performed in order to rank the problem's objectives where strong conflict exists between a pair of objectives.
- *Reference point* (also called a goal or an aspiration level vector): The user supplies, for each objective,

the desired level that he/she wishes to achieve. This desired level is called aspiration level.

- *Reservation point* (also called a reservation level vec- tor): The user supplies, for each objective, the ac- cepted level that he/she wishes to reach. This ac- cepted level is called reservation level.
- *Trade-off between objectives*: The user specifies that the gain of one unit in one objective is worth degradation in some others and vice versa.
- *Outranking thresholds*: The user specifies the necessary thresholds to design a fuzzy predicate modeling the truth degree of the predicate solution x is at least as good as solution y.
- *Desirability thresholds*: The user supplies: (1) an absolutely satisfying objective value and (2) a marginally infeasible objective value. These thresholds represent the parameters that define the desirability functions.

Based on these preference modeling tools, we observe that the goal of a preference-based EMO algorithm is to assign different importance levels to the problem's objectives with the aim to guide the search towards the Region of Interest (ROI) that is the portion of the Pareto Front that best matches the user preferences. In fact, usually, the user is not interested with the whole Pareto front and thus he/she is searching only for his/her ROI from which the problem's final solution will be selected. Several preference-based EMO algorithms have been proposed and used to solve real problems such as PI-EMOA [46], iTDEA [47], NOSGA [48], DF-SMS-EMOA [49], just to cite a few. There are several algorithmic challenges that should be overcome such as the preservation of Pareto dominance, the preservation of population diversity, the scalability with the number of objectives, etc.

Until now, the user's preferences are expressed and handled in the objective space. It is important to highlight that one of the original aspects of our work in this paper, as detailed later, is allowing the user (a software developer) to express his/her preferences in the decision space and then handling these preferences to help the user finding the most desired refactoring solution. Moreover, our approach helps the user in eliciting his/her preferences, which is very important for any preference-based EMO algorithm. These preferences are introduced implicitly by moving between the Pareto front of non-dominated solutions after obtaining feedback from the user about just a few parts of the solution in order to better understand his preferences. This implicit exploration of the Pareto front will be detailed in the next section where we describe the formulation of our refactoring problem.

The incorporation of user preferences may require the handling of dynamicity issues related to the introduced changes to the solution or the input (i.e. the software system). Handling dynamicity in EMO means solving dynamic problems where the objective functions and or the constraints may change over time such due to, for example, the dynamic nature of most of software evolution problems including software refactoring. Applying evolutionary algorithms (EAs) to solve Dynamic Multi-Objective Problems (DMOPs) has received great attention from researchers thanks to the adaptive behavior of evolutionary computation methods. A DMOP consists of minimizing or maximizing an objective function vector under some constraints over time. Its general form is the following [50]:

$$\begin{cases} Minf(x,t) = [f_1(x,t), f_2(x,t), \ldots, f_M(x,t)]^T \\ g_j(x,t) \geq 0, & j = 1, \ldots P; \\ h_k(x,t) = 0, & \text{k=1,...,Q;} \\ x_i^L \leq x_i \leq x_i^U, & \text{i=1,...,n;} \end{cases}$$

where M is the number of objective functions, t is the time instant, P is the number of inequality constraints, Q is the number of equality constraints, X_i^L and x_i^U correspond respectively to the lower and upper bounds of the variable x_i .

A solution x_i satisfying the $(P + Q)$ constraints is said to be *feasible*, and the set of all feasible solutions defines the feasible search space denoted by Ω. In this formulation, we consider a minimization MOP since maximization can be easily turned into minimization based on the duality principle by multiplying each objective function by -1 and transforming the constraints based on the duality rules.

The resolution of a MOP yields a set of trade-off solutions, called Pareto optimal solutions or non-dominated solutions, and the image of this set in the objective space is called the Pareto front. Hence, the resolution of a MOP consists in approximating the entire Pareto front. In the following, we provide some background definitions related to multi-objective optimization. It is worth noting that these definitions remain valid in the case of DMOPs.

Definition 1: Pareto optimality

A solution $x^* \in \Omega$ is Pareto optimal if $\forall x \in \Omega$ and $I = \{1, \ldots, M\}$ either $\forall m \in I$ we have $f_m(x) = f_m(x^*)$ or there is at least one $m \in I$ such that $f_m(x) > f_m(x^*)$.

The definition of Pareto optimality states that x^* is Pareto optimal if no feasible vector exists that would improve some objectives without causing a simultaneous worsening in at least one other objective.

Definition 2: Pareto dominance

A solution $u = (u_1, u_2, \ldots, u_n)$ is said to dominate another solution $v = (v_1, v_2, \ldots, v_n)$ (denoted by $\boldsymbol{f}(u) \prec \boldsymbol{f}(v)$) if and only if $\boldsymbol{f}(u)$ is partially less than $\boldsymbol{f}(v)$. In other words, $\forall m \in \{1, \ldots, M\}$ we have $f_m(u) \leq f_m(v)$ and $\exists m \in \{1, \ldots, M\}$ where $f_m(u) < f_m(v)$.

Definition 3: Pareto optimal set

For a given MOP $\boldsymbol{f}(x)$, the Pareto optimal set is $P^* = \{x \in \Omega | \neg \exists x' \in \Omega, \boldsymbol{f}(x') \prec \boldsymbol{f}(x)\}$.

Definition 4: Pareto optimal front

For a given MOP $\boldsymbol{f}(x)$ and its Pareto optimal set P^*, the Pareto front is $PF^* = \{\boldsymbol{f}(x), x \in P^*\}$.

In the next section, we describe an overview of our dynamic interactive refactoring approach then a detailed formulation of our solution.

3 SEARCH-BASED INTERACTIVE REFACTORING RECOMMENDATION

We first detail an overview of our approach and then we provide the details of our problem formulation and the solution approach.

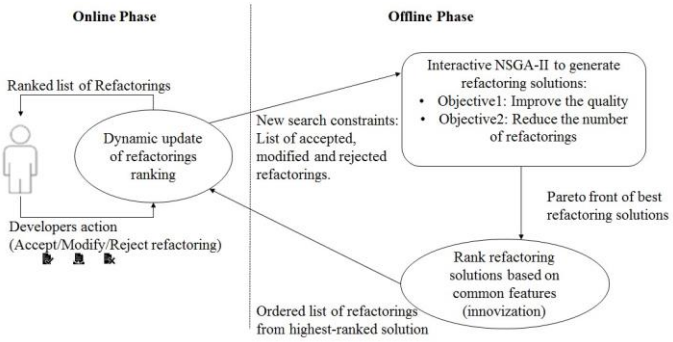


Figure 1. Approach Overview.

3.1 Approach Overview

The goal of our approach is to propose a new dynamic interactive way for software developers to refactor their systems. The general structure of our approach is sketched in Fig. 1.

Our technique comprises two main components. The first component is an *offline phase*, executed in the background, when developers are modifying the source code of the system. During this phase, the multi-objective algorithm, NSGA-II, is executed for a number of iterations to find the non-dominated solutions balancing the two objectives of improving the quality, which corresponds to minimizing the number of code smells, maximizing/preserving the semantic coherence of the design and improving the QMOOD (Quality Model for Object-Oriented Design) quality metrics, and the second objective of minimizing the number of refactorings in the proposed solutions.

The output of this first step of the *offline phase* is a set of Pareto-equivalent refactoring solutions that optimizes the above two objectives. The second step of the offline phase explores this Pareto front in an intelligent manner using innovization to rank recommended refactorings based on the common features between the non-dominated solutions. In our adaptation, we assume true the hypothesis that the most frequently occurring refactorings in the non-dominated solutions are the most important ones. Thus, the output of this second step of the offline phase is a set of ranked solutions based on this frequency score. NSGA-II is able to generate not only one good refactoring solution, but a diverse set of non-dominated solutions. This set of refactoring solutions may include specific patterns that make them better and different than dominated (imperfect) refactoring solutions.

To extract these patterns, we used the heuristic of prioritizing the recommendation of refactorings that are the most redundant ones among the non-dominated solutions. To our intuition, it seems very likely that common patterns in the set of non-dominated solutions are very likely to be good patterns. The opposite situation, where some non-dominated solutions share a pattern that in of poor quality, seems highly unlikely, though it could plausibly occur were the poor quality pattern to be an essential enabling feature for another pattern of high quality. While we are only expressing an intuition here, innovization has proven itself to be of value later in the experiments section.

The second component of our approach is an *online phase* to manage the interaction with the developer. It dynamically updates the ranking of recommended refactorings based on the feedback of the developer. This feedback can be to approve/apply or modify or reject the suggested refactoring one by one as a sequence of transformations. Thus, the goal is to guide, *implicitly*, the exploration of the Pareto front to find good refactoring recommendations. Since the ranking is updated dynamically, our interactive algorithm allows the implicit move between non-dominated solutions of the Pareto front.

After a number of interactions, developers may have modified or rejected a high number of suggested refactorings or have introduced several new code changes (new functionalities, fix bugs, etc.). Whenever the developers stop the refactoring session by closing the suggestions window, the first component of our approach is executed again on the background to update the last set of non-dominated refactoring solutions by continuing the execution of NSGA-II based on the two objectives defined in the first component and also the new constraints summarizing the feedback of the developer. In fact, we consider the rejected refactorings by the developer as constraints to avoid generating solutions containing several already rejected refactorings. This may lead to reducing the search space and thus a fast convergence to better solutions. Of course, the continuation of the execution of NSGA-II takes as input the updated version of the system after the interactions with developers. The whole process continues until the developers decide that there is no necessity to refactor the system any further.

3.2 Adaptation

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

3.2.1 Multi-objective formulation

In our previous work [9], we proposed a fully automated approach, to improve the quality of a system while preserving its domain semantics. It uses multi-objective optimization based on NSGA-II to find the best compromise between code quality improvements and reducing the number of code changes.

In this current work, we introduce the interactive component to our NSGA-II algorithm, which radically changes the process of finding good refactoring solutions in comparison to our earlier work. We will compare later in the experiments the performance of both algorithms. We present in the following the different adaptation steps of our approach. We ignored in this new interactive approach two objectives considered in our previous automated refactoring work. These two objectives are used to estimate, preserve and improve the design coherence (semantics) when fully automatically refactoring software systems. The very initial version of our experiments actually added the interaction, dynamic and innovization components at the top of our previous work. However, we found that the user interactions and the constraints learned and generated from it provided the required guidance to avoid semantics incoherences. Furthermore, the consideration of a large number of objectives make the execution time much longer to converge towards acceptable solutions since an increase in the number of objectives will increase the number of non-dominated

solutions to analyze which is not suitable for interactive optimization algorithms since it will introduce noise in the search. Thus, we considered the textual measures as constraints to satisfy when generating the refactoring solutions rather than an objective to optimize as highlighted later. The users interaction history is sufficient based on our experiments thus we ignored the use of development history in our new interactive approach.
As explained in Algorithm 1, the process starts with a complete execution of a regular NSGA-II algorithm based on the objectives described in the previous section (*offline* phase) then three components are introduced to improve the recommendations: innovization, interactive and dynamic components.

Algorithm 1 Dynamic Interactive NSGA-II at generation t

Input
Sys: system to evaluate, Pt: parent population
Output
P_{t+1}
Begin
/* Test if any user interaction occurred in the previous iteration */
if UserFeedback = TRUE **then**
    /* Rejected refactoring operations as constraints */
    $Ct \leftarrow GetConstraints()$;
    /* Updated source code after applying changes */
    $Sys \leftarrow GetRefactored - System()$;
    $UserFeedback \leftarrow FALSE$;
end if
$S_t \leftarrow \emptyset, i \leftarrow 1$;
$Q_t \leftarrow Variation(P_t)$;
$R_t \leftarrow P_t \cup Q_t$;
$P_t \leftarrow evaluate(P_t, C_t, Sys)$;
$(F_1, F_2, ...) \leftarrow NonDominatedSort(R_t)$;
repeat
    $S_t \leftarrow S_t \cup F_i$;
    $i \leftarrow i + 1$
until $(|S_t| \geq N)$
$F_l \leftarrow F_i$; 1> //Last front to be included
if $|S_t| = N$ **then**
    $P_{t+1} \leftarrow S_t$;
else
    $P_{t+1} \leftarrow \cup_{j=1}^{l-1} F_j$;
    /*Number of points to be chosen from F_l*/
    $K \leftarrow N - |P_{t+1}|$;
    /*Crowding distance of points in Fl */
    $Crowding - Distance - Assignment(F_l)$;
    $Quick - Sort(F_l)$;
    /*Choose K solutions with largest distance*/
    $P_{t+1} \leftarrow P_{t+1} \cup Select(F_l, k)$;
end if
if $t + 1 = Threshold$ **then**
    $UserFeedback \leftarrow TRUE$;
    /* Select and rank the best front */
    $Rank - Solution(F_1)$;
    $Threshold \leftarrow Threshold + t + 1$;
end if
End

The first iterations of the algorithm identify the Pareto front of the non-dominated refactoring solutions based on the fitness functions that will be discussed later. Then, the innovization component (Section 3.3) ranks the different non-dominated solutions based on the most common refactoring patterns between them. The different ranked refactorings are presented to the user based on the interactive component. During this interactive component, the developer may accept or reject or modify the refactoring recommendations (Section 3.3). Finally, the last dynamic component uses the interaction data with the user to reduce the search space of possible refactoring solutions and improve the future suggestions by repairing the Pareto front as detailed later in Section 3.3.

3.2.2 Solution representation

A solution consists of a sequence of n refactoring operations involving one or multiple source code elements of the system to refactor. The vector-based representation is used to define the refactoring sequence. Each vector's dimension has a refactoring operation 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 ensure the feasibility of the operation.
The initial population is generated by randomly assigning a sequence of refactorings to a randomly chosen set of code elements, or actors. 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. An actor can be a package, class, field, method, parameter, statement or variable. Table 1 depicts, for each refactoring, its involved actors and its corresponding parameters.

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 (HV) performance indicator [49] to determine the upper bound after which, the indicator remains invariant. For the lower bound, it is arbitrarily chosen. The experiments section will specify the upper and lower bounds used in this study. Table 2 shows an example of a refactoring solution including three operations applied to a simplified version of a solution applied to JVacation v1.0, a Java open-source trip management and scheduling software.

3.2.3 Solution variation

In each search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards optimal solutions.
For the crossover, we use the one-point crossover operator. It starts by selecting and splitting at random two parent solutions. Then, this operator creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice versa for the second child. This operator must ensure the respect of the length limits by eliminating randomly some refactoring operations. It is important to note that in multi-objective optimization, it is better to create children that are close to

Table 1
List of considered refactorings for our solution representation.

Refactorings	Actors	Roles
Extract class	class	source class, new class
	field	moved fields
	method	moved methods
Extract interface	class	source classes, new interface
	method	moved abstract methods
Inline class	class	source class, target class
Move field	class	source class, target class
	field	moved field
Move method	class	source class, target class
	method	moved method
Push down field	class	super class, subclasses
	field	moved field
Push down method	class	super class, subclasses
	method	moved method
Pull up field	class	subclasses, super class
	field	moved field
Pull up method	class	subclasses, super class
	method	moved method
Move class	package	source package, target package
	class	moved class
Extract method	method	source class, new class
	field	moved fields

their parents in order to have a more efficient search process. For mutation, we use the bit-string mutation operator that picks probabilistically one or more refactoring operations from the solutions and replace or modify or delete them. While the crossover operator does not introduce or modify a refactoring of a solution but just the sequence (a swap between refactoring of different solutions), the mutation operator definitely can add or modify or delete a refactoring when applied to any solution of the population. When a mutation operator is applied, the goal is to slightly change the solution for the purpose to probably improve its fitness functions. We used these three operations for the mutation operator that are randomly selected when a mutation is applied to a solution. Thus, the mutation operator can introduce new refactorings by either adding completely new ones or modifying the controlling parameters of an existing refactoring. For example, move method (m1, A, B) could be replaced by *movemethod*($m1, A, C$) or *movemethod*($m3, A, B$) where $m1$, A and B are the controlling parameters of the refactoring move method. Furthermore, the selection operator allows to regenerate part of the population randomly at every iteration thus new refactoring will be introduced since new solutions are generated during the execution process.

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 *movemethod* 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 [2]. We also apply a repair operator that randomly selects new refactorings to replace those creating conflicts.

3.2.4 Solution evaluation

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

Minimize the number of code changes as an objective: The application of a specific suggested refactoring sequence may require an effort that is comparable to that of re-implementing part of the system from scratch. Taking this observation into account, it is essential to minimize the number of suggested operations in the refactoring solution since the designer may have some preferences regarding the percentage of deviation with the initial program design. In addition, most developers prefer solutions that minimize the number of changes applied to their design. Thus, we formally defined the fitness function as the number of modified actors/code elements (packages, classes, methods, attributes) by the generated refactorings solution.

$$f(x) = \sum_{i-1}^{n} \#code\ elements(R_i, x) \qquad (1)$$

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 a 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 [7] instead of applying scattered changes throughout the whole system. The proposed fitness function is different from that employed in our previous work [9] 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 as an objective: Many studies have utilized structural metrics as a basis for defining quality indicators for a good system design [18, 51]. As an illustrative example, [38] proposed a set of quality measures, using the ISO 9126 specification, called QMOOD. Each of these quality metrics is defined using a combination of low-level metrics as detailed in Tables 3 and 4.

The QMOOD model has been used previously in the area of search-based software refactoring [18], [52] and so we use it to estimate the effect of the suggested refactoring solutions on software quality. QMOOD has the advantage that it defines six high-level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) that can be calculated using 11 lower level design metrics. Its objective function is defined as:

$$Quality = \frac{\sum_{i=1}^{6} QA_i(S)}{6} \qquad (2)$$

Table 2
Example of a solution representation.

Operation	Source/entity	Target entity
Move Method	ctrl.booking.BookingController::handleLodgingViewEvent (java.awt.event.ActionEvent):void	ctrl.booking.LodgingModel
Extract Class	ctrl.booking.SelectionModel:: - flightList + addFlight():void +clearFlight():void	ctrl.booking.FlightList
Move Method	ctrl.booking.BookingController::createBookings():void	ctrl.CoreModel

Table 3
QMOOD metrics description.

Design Metric	**Design Property**	**Description**
Design Size in Classes (*DSC*)	Design Size	Total number of classes in the design.
Number Of Hierarchies (*NOH*)	Hierarchies	Total number of "root" classes in the design (*count(MaxInheritenceTree (class)=0)*)
Average Number of Ancestors (*ANA*)	Abstraction	Average number of classes in the inheritance tree for each class.
Direct Access Metric (*DAM*)	Encapsulation	Ratio of the number of private and protected attributes to the total number of attributes in a class.
Direct Class Coupling (*DCC*)	Coupling	Number of other classes a class relates to, either through a shared attribute or a parameter in a method.
Cohesion Among Methods of class (*CAMC*)	Cohesion	Measure of how related methods are in a class in terms of used parameters. It can also be computed by: *1 − LackOfCohesionOfMethods()*
Measure Of Aggregation (*MOA*)	Composition	Count of number of attributes whose type is user defined class(es).
Measure of Functional Abstraction (*MFA*)	Inheritance	Ratio of the number of inherited methods per the total number of methods within a class.
Number of Polymorphic Methods (*NOP*)	Polymorphism	Any method that can be used by a class and its descendants. Counts of the number of methods in a class excluding private, static and final ones.
Class Interface Size (*CIS*)	Messaging	Number of public methods in class.
Number of Methods (*NOM*)	Complexity	Number of methods declared in a class.

Table 4
Quality attributes and their computation equations.

Quality attributes	**Definition** / **Computation**
Reusability	A design with low coupling and high cohesion is easily reused by other designs.
	$0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$
Flexibility	The degree of allowance of changes in the design.
	$0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism$
Understandability	The degree of understanding and the easiness of learning the design implementation details.
	$0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * DesignSize$
Functionality	Classes with given functions that are publicly stated in interfaces to be used by others.
	$0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendibility	Measurement of design's allowance to incorporate new functional requirements.
	$0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$
Effectiveness	Design efficiency in fulfilling the required functionality.
	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$

Where QA_i is the quality attribute number i being calculated based on the returned structural metrics from the system S.

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. We start from the assumption that the vocabulary of an actor is borrowed from the domain terminology and therefore can be used to determine which part of the domain semantics an actor encodes. Thus, two actors are likely to be semantically similar if they use similar vocabularies.

The vocabulary can be extracted from the names of methods, fields, variables, parameters, types, etc. We calculated the design coherence similarity between actors using an information retrieval-based technique, namely cosine similarity. Each actor is represented as an n-dimensional vector, where each dimension corresponds to a vocabulary term. The cosine of the angle between two vectors is considered as an indicator of similarity. More details can be found in our previous work [53].

3.3 Interactive Recommendation of Refactorings

The first step of the interactive component is executed as described in Algorithm 2, to investigate if there are some common principles among the generated non-dominated refactoring solutions.

Algorithm 2 Rank Refactoring Operation procedure

```
1: Input
2: NS: Non-dominated SolutionSet of the first front
3: Output
4: HM: HashMap of refactorings along with their occurrences.
5: Begin
6: HM ← ∅;
7: /* Compute the number of occurrence of each refactoring operation*/
8: for i = 1 to |NS| do
9:     for each j = 1 to |NS_i| do
10:        /* If a refactoring operation does not exist in the list, add its hash and set its occurrence number to 1*/
11:        if (NS_{i,j} ∉ HM) then
12:            HM ← HM ∪ Hash(NS_{i,j});
13:            HM[Hash(NS_{i,j})] ← 1;
14:        /* If a refactoring operation exists in the list, increment its occurrence number */
15:        else
16:            HM[Hash(NS_{i,j})] ← HM[Hash(NS_{i,j})] + 1;
17:        end if
18:    end for
19: end for
20: End
```

The algorithm checks if the optimal refactoring solutions have some common features such as identical refactoring operations among most or all of the solutions, and a specific common order/sequence in which to apply the refactorings. Such information will be used to rank the suggested refactorings for developers using the following formula:

$$Rank(R_{x,y}) = \frac{\sum_{j=0}^{n} \sum_{i=0}^{size(S_j)} [R_{i,j} = R_{x,y}]}{MAX(\sum_{j=0}^{n} \sum_{i=0}^{size(S_j)} [R_{i,j} = R_{x,y}])} \in [0...1] \quad (3)$$

where $R_{x,y}$ is the refactoring operation number x (index in the solution vector) of solution number y, and n is the number of solutions in the front. S_i is the solution of index i. All the solutions of the Pareto front are ranked based on the score of this measure applied to every solution.

Once the Pareto front solutions are ranked, the second step of the interactive step is executed as described in Algorithm 3. The refactorings of the best solution, in terms of ranking, are recommended to the developer based on their order in the vector. Then, the ranking score of the solutions is updated automatically after every feedback (interaction) with the developer. Our interactive algorithm proposes three levels of interaction as described in Fig. 2 and Algorithm 3.

Algorithm 3 GUF (Get User Feedback) procedure to manage the interactions with the developer (Online Phase)

```
1: Input
2: RNS: Ranked Non-dominated SolutionSet
3: Output
4: HM: HashMap of refactorings along with their occurrences.
5: Begin
6: AppliedRefactorings ← ∅;
7: RejectedRefactorings ← ∅;
8: for i = 1 to |RNS| do
9:     ref[i] ← 0;
10: end for
11: /* Main loop to suggest refactorings one by one to the user*/
12: while |RejectedRefactorings < a do
13:     /* Select index of the the solution with highest rank*/
14:     index ← MaxRank(RNS);
15:     d ← UserDecision(RNS_{index,ref[index]});
16:     /* If the user has applied or modified the operation*/
17:     if (d = True) then
18:         AppliedRefactorings ← AppliedRefactorings ∪ RNS_{index,ref[index]};
19:         /* If the user has rejected the operation*/
20:     else
21:         RejectedRefactorings ← RejectedRefactorings ∪ RNS_{index,ref[index]};
22:     end if
23:     Ref[index] ← ref[index] + 1;
24:     /* Update solutions indexes */
25:     for i = 1 to |RNS| do
26:         UpdateRank(RNS_i; AppliedRefactorings, RejectedRefac
27:     end for
28: end while
29: End
```

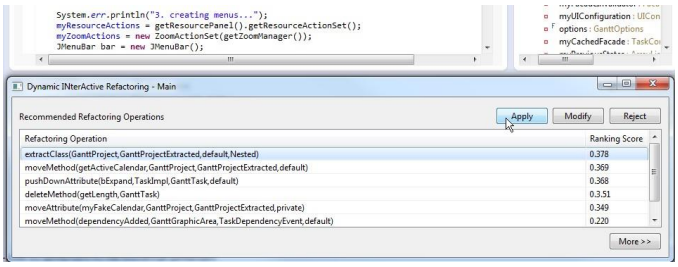


Figure 2. Refactorings recommended by our technique.

The developer can check the ranked list of refactorings and then *apply*, *modify* or *reject* the refactoring. If the developer prefers to modify the refactoring, then our algorithm can help them during the modification process as described in Fig. 3.

In fact, our tool proposes to the developer a set of recommendations to modify the refactoring based on the history of changes applied in the past and the semantic similarity between code elements (classes, methods, etc.). For example, if the developer wants to modify a move method

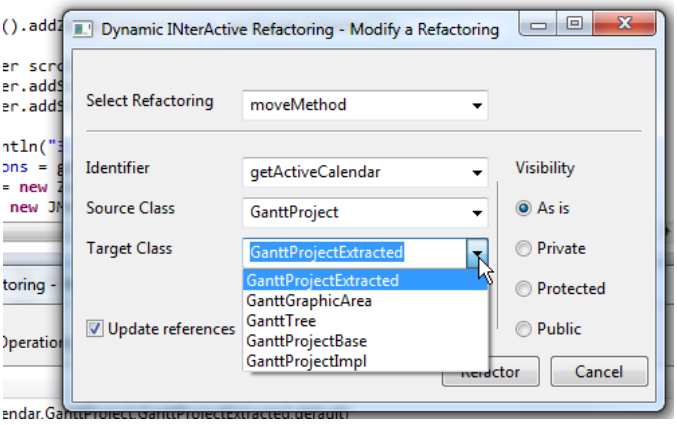


Figure 3. Recommended target classes by our technique for a move method refactoring to modify.

refactoring then, having specified the source method to move, our interactive algorithm automatically suggests a list of possible target classes ranked based on the history of changes and semantic similarity. This is an interesting feature since developers often know which method to move, but find it hard to determine a suitable target class [22]. The same observation is valid for the remaining refactoring types. Another action that the developers can select is to reject/delete a refactoring from the list.

After every action selected by the developer, the ranking is updated based on the feedback using the following formula:

$$Rank(S_i) = \sum_{k=1}^{size(S_i)} Rank(R_{k,i}) \quad (4)$$
$$+(RO \cap AppliedRefactoringsList)$$
$$-(RO \cap RejectedRefactoringsList)$$
$$+0.5 * (RO \cap ModifiedRefactoringsList)$$

Where S_i is the solution to be ranked, the first component consists of the sum of the ranks of its operations as explained previously and the second component will take the value of 1 if the recommended refactoring operation was applied by the developer, or -1 if the refactoring operation was rejected or 0.5 if it was partially modified by the developer. The recommended refactorings will be adjusted based on the updated ranking score.

It is important to note that we calculate the ranking score for each non-dominated solution using the innovization component and then the solution with the highest score is presented refactoring by refactoring to the developer. In fact, refactorings tend to be dependent on one another thus it is important to ensure the coherence of the recommended solution. After a number of modified or rejected refactorings or several new code changes introduced, the generated Pareto front of refactoring solutions by NSGA-II needs to be updated since the system was modified in different locations.

To check the applicability of the refactorings, we continuously check the pre-conditions of individual refactorings on the version after manual edits. Thus, the ranking of the solutions will change after every interaction. If many refactorings are rejected, the NSGA-II algorithm will continue to execute while taking into consideration all the feedback from developers as constraints to satisfy during the search. The rejected refactorings should not be considered as part of the newly generated solutions and the new system after refactoring will be considered in the input of the next iteration of the NSGA-II.

In the non-interactive refactoring systems, the set of refactorings, suggested by the best-chosen solution, needs to be fully executed in order to reach the solution's promised results. Thus, any changes applied to the set of refactorings such as changing or skipping some of them could deteriorate the resulting system's quality. In this context, the goal of this work is to cope with the above-mentioned limitation by granting to the developer's the possibility to customize the set of suggested refactorings either by accepting, modifying or rejecting them. The novelty of this work is the approach's ability to take into account the developer's interaction, in terms of introduced customization to the existing solution, by conducting a local search to locate a new solution in the Pareto Front that is nearest to the newly introduced changes. We believe that our approach may narrow the gap that exists between automated refactoring techniques and human intensive development. It allows the developer to select the refactorings that best matches his/her coding preferences while modifying the source code to update existing features.

3.4 Running Example: Illustration on the JVacation System

3.4.1 Context

To illustrate our interactive algorithm, we consider the refactoring of JVacation *v1.0* [1], a Java open-source trip management and scheduling software. We asked a developer to update an existing feature by adding one more field (*Premium member ID*) in the personal information form that a user has to fill out when booking a flight.

As JVacation architecture is based on the Model/View/Controller model, adding this extra field would trigger small updates on the View by adding a textbox in the personal information input form. Also the controller that handles the booking process needs to be revised. At the model level, an attribute needs to be added to the class that hosts the booking information. Finally, an update on the database level is needed to save the newly modified booking objects.

To simplify the illustration, we have limited the update to these above-mentioned changes knowing that, in order to completely implement this function, several other updates may be needed in other views and controllers in order to show, for example, the newly added field, as part of the information related to the passengers' records for a given flight. We asked the developer to refactor the software system while performing the given task, therefore, the developer has initially launched the plugin that triggered our interactive algorithms. We assisted the developer in only selecting the initial default parameters

1. https://sourceforge.net/projects/jvacation/

Table 5
Quality attributes value on the JVacation system.

Quality Attribute	Original System	Solution 1	Solution 2	Solution 3
Reusability	1.74225	(+0.5) 1.79225	(+0.4) 1.79225	(+0.5) 1.79225
Flexibility	1.82	(+0.001) 1.820	(+0.001) 1.820	(+0.001) 1.820
Understandability	-4.5408	(+0.08) -4.5398	(+0.07) -4.5398	(+0.087) -4.5398
Functionality	1.16314	(+0.5) 1.21314	(+0.6) 1.21314	(+0.5) 1.21314
Extendibility	19.7225	(+0.007) 19.7295	(+0.012) 19.7300	(+0.011) 19.7299
Effectiveness	9.5406	9.5406	9.5406	9.5406
Quality Gain	-	0.198	0.202	0.209
Number of operations	-	11	14	19

for the optimization algorithm (such as the minimum and maximum chromosome lengths).

3.4.2 Illustration of the Innovization Component

After generating the upfront list of best refactoring solutions, three solutions are selected from the Pareto front that were involved in the interactive session to simplify this running example. Each solution has a fitness score composed of the median of quality improvement calculated based on the structural measures of the refactored system for each solution, and the number of operations within each solution. The previous section describes, these fitness values, for each solution, in terms of quality improvement and refactoring effort compared to the original system values before refactoring. These information is shown in Table 5.

One of the classic challenges in multiobjective optimization is the choice of the most suitable solution for the developer. The straightforward solution for this problem would be to manually investigate all solutions, i. e., execute all refactoring operations for each solution and allow the developer to compare between several refactored designs. This task can easily become tedious due to the important number of solutions in the Pareto front.

To facilitate the selection task, decision making support tools can be used to automate the selection of solutions based on the decision maker's preferences. In our context, these preferences can be considered as the packages and classes that the developer is interested in when implementing the requested feature. Thus, another straightforward heuristic would be to automatically shortlist solutions that only refactor entities that are of interest to developers. Unfortunately, this will not necessarily reduce drastically the number of preferred solutions especially if the system is small.

To cope with this issue, another interesting idea would be to calculate the overlap between solutions. Still, choosing the most appropriate solution can be challenging as the developer has to manually break the tie between solutions by comparing between their specific refactorings. This comparison may not be straightforward because specific refactorings between to candidate solutions may both be of an interest to the developer, for example, when comparing between solution 1 and solution 2, both solutions contain a move-method operation that agree on moving a function called *getSaluation()* but disagree on the target class.

Since this function belongs to the booking panel, the participating entities are of an interest to the developer, so no choice can be automatically done based on the developer's preferred entities. Moreover, both target classes (respectively *LabelSpinner* and *LabelEdit*), each proposed by one solution, belong to the same package (*gui.components*) and they are semantically close, so the fitness function values cannot be used to break the tie. In this scenario, only the developer would be qualified to take the decision of either accepting one operation over the other or maybe rejecting both operations. Thus, simply filtering solutions based on the developer's preferred entities may fall short in this kind of scenarios. Furthermore, asking the developer to exhaustively break the tie between shortlisted solutions can become tedious.

In this context, our interactive process differs from simply filtering operations based on a given preference as it learns from the developer's decision making and dynamically break the tie between Pareto-equivalent solutions by upgrading those with the highest number of successful recommendations (applied refactorings) while penalizing those who contain rejected operations. To illustrate this process, Table 6 describes each solution's refactorings along with its rank after the execution of the first step of the interactive algorithm. For the purpose of simplicity, we considered a first fragment of each solution. The solutions are ranked based on Equation 3 to identify the most common refactorings between the non-dominated solutions. This is achieved by counting the number of occurrences of operation within the Pareto front solution set, this number will be averaged by the maximum number of occurrences found.

3.4.3 Illustration of the Interactive and Dynamic Components

In the interaction part, the recommended refactoring wanted to move a function that defines the trip's starting date to a *LabelCombo* class. The developer thought that moving it to *DateEdit* class makes more sense instead because the return value of the moved function is of type Date and *DateEdit* is semantically closer to the method. So the refactroings were partially modified by the developer and the ranking score of the second solution was increased by 0.5 for *Solution 2* but by 1 for *Solution 3* since it has already a move method operation that suggests moving the same method to the chosen class by the developer, i. e., the applied operation exists in that solution.

In the third interaction, the recommended refactoring suggests merging two classes *CoreModel* and *ModelChangeEvent*. The first class gathers, for a given customer, all his/her bookings and sums up the total price, since the price may be later on reduced based on the customer's premium number (field to be added) the developer decided to keep the class intact and thus the operation was rejected and so the score of the top *Solution 2* was decreased by 1. The solution with the highest rank is selected for execution and its related

Table 6
Three simplified refactoring solutions recommended for JVacation v1.0.

Operation	Source entity	Target entity
Solution 1 fitness scores before normalization (0.198, 4)		
Move Method	ctrl.booking.BookingController::handleLodgingViewEvent(java.awt.event.ActionEvent):void	ctrl.booking.LodgingModel
Extract Class	ctrl.booking.SelectionModel:: - flightList + addFlight():void +clearFlight():void	ctrl.booking.FlightList
Move Method	ctrl.booking.BookingController::createBookings():void	ctrl.CoreModel
Move Method	gui.panels.booking.bTravelersPanel::getSalutation():java.lang.String	gui.components.LabelSpinner
Solution 1 Rank		3.960
Solution 2 fitness scores before normalization (0.202, 5)		
Move Method	ctrl.booking.BookingController::handleLodgingViewEvent(java.awt.event.ActionEvent):void	ctrl.booking.lodgingList
Move Method	gui.panels.maintenance.mLodgingsPanel::getStart():java.util.Date	gui.components.LabelCombo
Inline Class	ctrl.ModelChangeEvent	ctrl.CoreModel
Extract Class	ctrl.booking.SelectionModel:: - travelerList + addTraveler():void +clearTraveler():void	ctrl.booking.TravelerList
Move Method	gui.panels.booking.bTravelersPanel::getSalutation():java.lang.String	gui.components.LabelSpinner
Solution 2 Rank		4.064
Solution 3 fitness scores before normalization (0.209, 6)		
Move Method	ctrl.booking.BookingController::handleLodgingViewEvent(java.awt.event.ActionEvent):void	ctrl.booking.lodgingList
Extract Class	ctrl.booking.SelectionModel:: - flightList + addFlight():void +clearFlight():void	ctrl.booking.FlightList
Extract Class	ctrl.booking.SelectionModel:: - travelerList + addTraveler():void +clearTraveler():void	ctrl.booking.TravelerList
Inline Class	ctrl.ModelChangeEvent	ctrl.CoreModel
Move Method	gui.panels.maintenance.mLodgingsPanel::getStart():java.util.Date	gui.components.DateEdit
Move Class	Db.factory.DBObjectFactory	db
Solution 3 Rank		3.471

operations are shown to the user based on their order in the vector. Table 7 summarizes the various interactions between the developer and the suggested refactorings from the three above mentioned solutions when adding the new feature.

The first recommended refactoring of the top ranked solution (Solution 2) suggests moving an event function from the controller class of the booking process, since the developer is required to investigate this class and since this function is not called during the booking procedure, moving it out of the class will reduce the number of investigated functions, so the operation was applied by the developer and accordingly the ranking score was increased by 1 for both *Solutions 2* and *3* since they include this refactoring in their solutions.

Upon the rejection of the third suggested refactoring, the ranking score of *solution 3* has become higher than the one of *solution 2*, this has triggered the fourth recommended operation to be issued from *solution 3* instead. All the refactorings that belong to the intersection between *solution 3* and the lists of applied/rejected refactorings will be skipped during the recommendation process.

For instance, the first and second operation of *solution 3* will be skipped as they have been already applied by the developer, and the third operation will be suggested during the fourth interaction. This operation suggests the extraction of a class from the selection mode of the booking process. Since this refactoring will facilitate the distinction between functions related to the flight from those related to the passengers, the developer has approved the operation. The algorithm will stop recommending new refactorings either on the request of the developer or when the system achieves acceptable quality improvement in terms of reducing the number of design defects and improving quality metrics. These parameters can be specified by the developer or the team manager.

4 VALIDATION

To evaluate the ability of our refactoring framework to generate good refactoring recommendations, we conducted a set of experiments based on eight open source systems and two industrial projects provided by the IT department at the Ford Motor Company. The obtained results are subsequently statistically analyzed with the aim of comparing our proposal with a variety of existing approaches. The relevant data related to our experiments can be found in [61].

In this section, we first present our research questions and validation methodology followed by experimental setup. Then we describe and discuss the obtained results.

4.1 Research Questions

We defined three categories of research questions to measure the correctness, relevance and benefits of our interactive multi-objective refactoring approach comparing to the state of the art based on several practical scenarios. It is important to evaluate, first, the correctness of the recommended refactoring. Since it is not sufficient to make correct refactoring recommendations, we evaluated the benefits of applying the recommended refactorings in terms of fixing code smells and improving quality attributes. Programmers are not interested, in practice, to apply *all* the correct and useful recommended refactorings due to limited resources thus we evaluated both the relevance of our recommendations and our ranking efficiency from programmers perspective based

Table 7
Four different interaction examples with the developer applied on the refactoring solutions recommended for JVacation v1.0.

Operation	R1:MoveMethod(ctrl.booking.BookingController ::handleLodgingViewEvent:void, ctrl.booking.LodgingModel)		
Decision	Applied		
Changes	AppliedRefactoringsList = {R1} , RejectedRefactoringsList = {}		
SolutionSet	Solution1	Solution2 *	Solution3
Initial rank	3.960	4.064	3.471
Interation1	3.960	5.064 (+1)	4.471 (+1)

Operation	R2:MoveMethod(gui.panels.maintenance. mLodgingsPanel::getStart():java.util.Date, gui.components.LabelCombo)		
Decision	Modified to: R2: MoveMethod(gui.panels.maintenance. mLodgingsPanel::getStart():java.util.Date, gui.components.DateEdit)		
Changes	AppliedRefactoringsList = {R1,R2} , RejectedRefactoringsList = {}		
SolutionSet	Solution1	Solution2 *	Solution3
Initial rank	3.960	4.064	3.471
Interation1	3.960	5.064 (+1)	4.471 (+1)
Interation2	3.960	5.564 (+0.5)	5.471 (+1)

Operation	R3: InlineClass(ctrl.ModelChangeEvent, ctrl.CoreModel)		
Decision	Rejected		
Changes	AppliedRefactoringsList = {R1,R2} , RejectedRefactoringsList = {R3}		
SolutionSet	Solution1	Solution2 *	Solution3
Initial rank	3.960	4.064	3.471
Interation1	3.960	5.064 (+1)	4.471 (+1)
Interation2	3.960	5.564 (+0.5)	5.471 (+1)
Interation3	3.960	4.564 (-1)	5.471

Operation	R4: ExtractClass(ctrl.booking.SelectionModel::- flightList +addFlight():void+clearFlight():void, ctrl.booking.FlightList)		
Decision	Applied		
Changes	AppliedRefactoringsList = {R1,R2,R4} , RejectedRefactoringsList = {R3}		
SolutionSet	Solution1	Solution2	Solution3 *
Initial rank	3.960	4.064	3.471
Interation1	3.960	5.064 (+1)	4.471 (+1)
Interation2	3.960	5.564 (+0.5)	5.471 (+1)
Interation3	3.960	4.564 (-1)	5.471
Interation4	4.960 (+1)	4.564	6.471 (+1)

on several real-world scenarios including productivity and post-study questionnaires. We considered various existing refactoring approaches as a baseline for this proposed interactive refactoring technology to define an accurate estimation of possible improvements.

The four research questions are as follows:

RQ1: Correctness, Relevance and Comparison with State of the Art.

- **RQ1-a: Correctness.** To what extent the results of our approach are similar to the ones proposed by developers compared to fully-automated refactoring techniques?
- **RQ1-b: Benefits–antipatterns correction.** To what extent code smells can be fixed using our approach compared to fully-automated refactoring techniques?
- **RQ1-c: Benefits–improving quality.** To what extent can our approach improve the overall quality of software systems compared to fully-automated refactoring techniques?
- **RQ1-d: Relevance to programmers.** To what extent can our approach make meaningful recommendations compared to fully-automated refactoring techniques?

RQ2: Interaction Relevance. To what extent can our approach **efficiently rank** the recommended refactorings?

RQ3: Impact based on Practical Scenarios.

- **RQ3-a:** To what extent our approach can improve the **productivity of programmers when fixing bugs** compared to fully-automated refactoring techniques?
- **RQ3-b:** To what extent our approach can improve the **productivity of programmers when adding new features** compared to fully-automated refactoring techniques?
- **RQ3-c:** How do programmers evaluate the **usefulness of our approach (questionnaire)**?

4.2 Validation Methodology

To answer the research questions described in the previous section, we give, first, an overview about the adopted validation methodology that include the following tasks:

- **Task-1:** Generate data for baseline methods by using other existing state-of-the-art automated refactoring tools and methods offline. (RQ1a-d)
- **Task-2:** Manually refactor a system. (RQ1a)
- **Task-3:** Use our tool (DINAR) to collect final set of recommendations. (RQ1a-d, RQ2)
- **Task-4:** Rate solutions and recommendations of different methods and tools. (RQ1d, RQ2)
- **Task-5:** Code smells detection after refactoring. (RQ1b)
- **Task-6:** Measure quality metrics after refactoring. (RQ1c)
- **Task-7:** Fix bugs on refactored / unrefactored systems. (RQ3a)
- **Task-8:** Implement features on refactored / unrefactored systems.(RQ3b)
- **Task-9:** Post-study questionnaire. (RQ3c)

For each task, we defined and used different evaluation metrics (Precision, Recall, number of fixed antipatterns, the quality gain, manual correctness, number of modified/rejected/accepted recommendations and execution time) which are described in this section. These metrics are calculated and compared for different refactoring techniques which are applied on a variety of software projects under the specific above scenarios. Table 8 shows the summary of the connections between the research questions, metrics and tasks detailed in this section.

Table 8
Summary of the research questions, their goals, defined metrics to answer and analyse them, and the associated tasks to collect data and calculate the metrics.

RQ#	RQ Goal	Sub-RQ	Sub-Goal	Metric(s)	Task(s)#
RQ1	Relevant Solutions	RQ1-a	Similarity	RC, PR	1, 2, 3
		RQ1-b	Fixing code smells	NF	1,3,5
		RQ1-c	Overall quality	G	1,3,6
		RQ1-d	Meaningful recommendation	MC	1,3,4
RQ2	Efficient ranking	-		NAR, NRR, NMR, PR@k, MC@k	3, 4
RQ3	Usefulness	RQ3-a	Productivity / fixing bugs	TP	7
		RQ3-b	Productivity /adding features		8
		RQ3-c	questionnaire		9

In order to have a consistent comparison, we considered the refactoring solutions recommended by our approach after all interactions with the developers (last set of solutions). Therefore, we refer to these sets of refactoring solutions as our approach results afterward. To create a baseline, we asked the participants in our study to analyze and apply manually several refactoring types using Eclipse IDE on several code fragments extracted from different systems where most of them correspond to code smells identified in previous studies as worth removing by refactoring [19, 54]. This golden set is defined based on the following two main criteria: 1. Refactorings that fix a design flaw and did not change the behavior or introduce bugs, 2. Refactorings that improve a set of quality metrics (based on the QMOOD model) and did not change the behavior or introduce bugs. We refer to these refactoring solutions as expected refactorings afterward.

To answer RQ1, it is important to validate the proposed refactoring solutions from both quantitative and qualitative perspectives. For RQ1-a, we calculated precision and recall scores to compare between refactorings recommended by each approach and those expected based on the participants opinion:

$$RC_{recall} = \frac{Approach\ Solution \cap Expected\ Refactorings}{Expected\ Refactorings} \in [0, 1] \quad (5)$$

$$PR_{precision} = \frac{Approach\ Solution \cap Expected\ Refactorings}{Approach\ Solution} \in [0, 1] \quad (6)$$

When calculating the precision and recall, we consider a refactoring as a correct recommendation if all the controlling parameters are the same like the expected ones.

For RQ1-b, we considered another quantitative evaluation which is the percentage of fixed code smells (*NF*) by the refactoring solution. The detection of code smells after applying a refactoring solution is performed using the detection rules of [19]. Formally, *NF* is defined as:

$$NF = \frac{\#fixed\ code\ smells}{\#code\ smells} \in [0, 1] \quad (7)$$

The detection of code smells 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 for RQ1-c another metric, *G*, based on QMOOD that estimates the quality improvement of the system by comparing the quality before and after refactoring independently from the number of fixed design defects. The average of the six QMOOD attributes were used: reusability, flexibility, understandability, Extendibility, Functionality and effectiveness. All of them are formalized using a set of quality metrics. Hence, the gain for each of the considered QMOOD quality attributes and the average total gain in quality after refactoring can be easily estimated as:

$$G = \frac{\sum_{i=1}^{6} G_{q_i}}{6} \quad and \quad G_{q_i} = q_i' - q_i \quad (8)$$

where q_i' and q_i represents the value of the QMOOD quality attribute i after and before refactoring, respectively.

For RQ1-d, we asked the participant in our study, as detailed in Section 4.4, to evaluate, manually, whether the suggested refactorings are feasible and efficient at improving the software quality and achieving their maintainability objectives. We define the metric Manual Correctness (MC) to mean the number of meaningful refactorings divided by the total number of recommended refactorings. Therefore, MC is given by the following equation:

$$MC = \frac{\#\ Meaningful\ Refactorings}{\#\ Recommended\ Refactorings} \quad (9)$$

To avoid the computation of the MC metric being biased by the developer's feedback, we asked the developers to manually evaluate the correctness of the recommended refactorings of our approach on the systems that they did not refactor using our tool. Therefore, The developers did not evaluate the results of their own results of interactive refactoring but the resultant refactorings recommended on other systems where other developers apllied our approach. The main motivation for the manual correctness metric is actually to address the concern that the deviation with the expected refactorings could be just because of the preferences of the developers. The manual correctness metric is evaluated manually on each refactoring one-by-one to check their validity. Thus, we evaluated the results produced by the different tools and we were not limited to the comparison with the expected results. We did the comparison with the expected results to provide an automated way to evaluate the results and avoid the developers being biased by the results of our tool (developers did not know anything about the refactorings suggested by the different tools when

they provided their recommendations).
We used the metrics *MC, RC, PR, NF* and *G* to perform the comparisons and answer respectively RQ1a-d.
We considered some other useful metrics to answer RQ2 that count the percentage of refactorings that were accepted (*NAR*) or rejected (*NRR*) or applied with some modifications (*NMR*). Formally, these metrics are defined as:

$$NAR = \frac{\#Accepted\,Refactorings}{\#\,Recommended\,Refactorings} \in [0, 1] \quad (10)$$

$$NRR = \frac{\#\,Rejected\,Refactorings}{\#\,Recommended\,Refactorings} \in [0, 1] \quad (11)$$

$$NMR = \frac{\#\,Modified\,Refactorings}{\#\,Recommended\,Refactorings} \in [0, 1] \quad (12)$$

To answer RQ2, we also evaluated the relevance of the recommended refactorings in the top *k* where *k =1, 5, 10* and *15* using the following metrics *PR@k* and *MC@k*. We used the same equations defined for RQ1 with the only difference that the considered suggested refactorings are exclusively those located in the top *k* positions of the ranked list of refactorings at multiple instances after the execution of the innovization component.
To answer RQ3, we aimed to assess how the refactoring actually increases the software quality and productivity in that the effort to fixing bugs (R3-a) or adding new features (R3-b) should reduce after performing the refactorings. We asked the software developers participated in this study to add new features and fix a set of bugs. To avoid that the achieved results might be due to the different levels of ability of the developers groups, we adapted a counter-balanced design where each participant performed two tasks, one on the original system and one on the refactored system. The details of these scenarios will be described later as detailed in Section 4.6. To estimate the impact of the suggested refactorings on the productivity of developers, we defined the following metric *TP* to measure the time required to perform the same activities on the system with and without refactoring:

$$TP_i = 1 - \frac{\#minutes\,required\,to\,perform\,task\,i\,on\,the\,system\,after\,refactoring}{\#minutes\,required\,to\,perform\,task\,i\,on\,the\,system\,befor\,refactoring} \quad (13)$$

We have also compared the productivity results of our approach compared to Kessentini et al. [19], Ouni et al. [9] and Harman et al. [17] to test the hypothesis if better quality of the software may increase the productivity of developers. To answer RQ3-b, we used a post-study questionnaire that collects the opinions of developers on our tool as detailed in the next section.

4.3 Studied Software Projects

We used a set of well-known open-source Java projects and two systems from our industrial partner, the Ford Motor Company. We applied our approach to eight open-source Java projects: Xerces-J, JHotDraw, JFreeChart, GanttProject, Apache Ant, Rhino and Log4J and Nutch. Xerces-J is a family of software packages for parsing XML. JFreeChart is a free tool for generating charts. Apache Ant is a build tool and library specifically conceived for Java applications. Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. GanttProject is a cross-platform tool for project scheduling. Log4J is a popular logging package for Java. Nutch is an Apache project for web crawling. JHotDraw is a GUI framework for drawing editors.
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 thousands of business constraints. This system is also used by Ford Motor Company to improve their vehicles sales by selecting the right vehicle configuration to match the expectations of their customers. JDI is highly structured and software developers have developed several versions of it at Ford over the past 10 years. Due to the importance of the application and the high number of updates performed on both systems, it is critical to ensure that they remain of high quality so to reduce the time required by developers to introduce new features in the future.
We selected these 10 systems for our validation because they range from medium to large-sized open-source projects, which have been actively developed over the past 10 years, and their design has not been responsible for a slowdown of their developments. Table 9 provides some descriptive statistics about these 10 programs.

Table 9
Statistics of the studied software projects.

System	Release	#classes	KLOC	#Code smells	#Applicable Refactorings
Xerces-J	v2.7.0	991	240	61	80
JHotDraw	v6.1	585	21	22	36
JFreeChart	v1.0.9	521	170	51	96
GanttProject	v1.10.2	245	41	60	63
Apache Ant	v1.8.2	1191	255	61	74
Rhino	v1.7R1	305	42	79	50
Log4J	v1.2.1	189	31	27	41
Nutch	v1.1	207	39	39	24
JDI-Ford	v5.8	638	247	83	94
MROI-Ford	V6.4	786	264	97	119

4.4 Study Participants

Our study involved 14 participants from the University of Michigan and 8 software developers from the Ford Motor Company. Participants include 6 master students

in Software Engineering, 8 Ph.D. students in Software Engineering and 8 software developers. All the participants are volunteers and familiar with Java development and refactoring. The experience of these participants on Java programming ranged from 2 to 19 years. We carefully selected the participants to make sure that they already applied refactorings during their previous experiences in development.

All the graduate students have already taken at least one position as software engineer in industry for at least two years as software developer and most of them (11 out of 14 students) participated in similar experiments in the past, either as part of a research project or during graduate courses on Software Quality Assurance or Software Evolution offered at the University of Michigan. Furthermore, 6 out the 14 students (the selected master students) are working as full-time or part-time developers in the software industry.

Participants were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software refactoring. In addition, all the participants attended one lecture about software refactoring and passed six tests to evaluate their performance in evaluate and suggest refactoring solutions.

We formed 3 groups. The groups were formed based on the pre-study questionnaire and the test results to ensure that all the groups have almost the same average skill level. We divided the participants into groups according to the studied systems, the techniques to be tested and developers' experience.

Each of the first two groups (A and B) is composed of three masters students and four Ph.D. students. The third group is composed of eight software developers from the Ford Motor company, since they agreed to participate only in the evaluation of their two software systems. It is important to note that the third group formed by the developers from Ford is part of the original developers of the two evaluated systems.

4.5 Techniques Studied

4.5.1 Overview of the used techniques

To answer our research questions from the perspective of evaluating our interactive approach performance against the state-of-the-art refactoring techniques, we compared our approach to four other existing fully-automated search-based refactoring techniques and our multi-objective approach without the interaction component (NSGA-II-Innovization). Studied techniques includes: Kessentini et al. [19], O'Keeffe and O' Cinnide [18], Ouni et al. [9] and Harman et al. [17] that consider the refactoring suggestion task only from the quality improvement perspective.

Autors in [19], formulate software refactoring as a mono-objective search problem where the main goal is to fix design defects and improve quality metrics. Also, [18] proposed a mono-objective formulation to automate the refactoring process by optimizing a set of quality metrics. The authors in [9] and [17] proposed a multi-objective refactoring formulation that generates solutions to fix code smells. Both techniques are non-interactive and fully-automated.

We considered in our experiments another popular design defects detection and correction tool JDeodorant [21] that does not use heuristic search techniques. The current version of JDeodorant is implemented as an Eclipse plug-in that identifies some types of design defects using quality metrics and then proposes a list of refactoring strategies to fix them. 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 such as *Move Method*, *Extract Method* and *Extract Class*.

Our approach differs with the above fully-automated techniques in two factors: *innovization* and *interactive features*. Therefore, it is important to evaluate the impact of every factor on the quality of our results. If the innovization makes the largest contribution, which is another fully automated search-based approach, the results cannot support the hypothesis related to the outperformance of interactive refactoring. Thus, we compared our approach to NSGA-II with the innovization feature using the same multi-objective optimization but without the use of the interactive feature.

All these existing techniques are fully-automated and do not provide any interaction with the developers to update their solutions.

Table 10 summarizes the survey organization including the list of systems and algorithms evaluated by the groups of participants.

Table 10
Survey organization.

Participants groups	Software Projects	Approaches	Tasks
Group A	Xerces-J	Interactive NSGA-II, [18], [9], JDeodorant [21], [19], [17]	-Interactive refactoring -Manual refactoring -Post-study questionnaire -Fixing bugs -Adding features
	JHotDraw		
	JFreeChart		
	GanttProject		
Group B	Apache Ant		
	Rhino		
	Log4J		
	Nutch		
Group C	JDI-Ford	Interactive NSGA-II, [18], [9], JDeodorant [21]	
	MROI-Ford		

4.5.2 Parameters setting

Parameter setting influences significantly the performance of a search algorithm on a particular problem [55]. For this reason, for each algorithm and for each system, we perform a set of experiments using several population sizes: 50, 100, 200, 300 and 500. The stopping criterion was set to 100,000 evaluations for all algorithms in order to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.5 where the probability of gene modification is 0.3; stopping criterion = 100,000 evaluations. In order to have significant results, for each couple (algo-

rithm, system), we use the trial and error method [62] in order to obtain a good parameter configuration. Trial and error is a fundamental method of problem solving. It is characterized by repeated and varied attempts of algorithm configurations.

Regarding the evaluation of fixed code smells, we focus on the following code smell types: Blob, Spaghetti Code (SC), Functional Decomposition (FD), Feature Envy (FE), Data Class (DC), Lazy Class (LC), and Shotgun Surgery (SS). We choose these code smell types in our experiments because they are the most frequent and hard to fix based on several studies [23, 25]. These design flaws are automatically detected using the detection rules of our previous work [19] based on genetic programming. We have generated and manually validated, in [19] and several of our other previous studies, a set of metrics-based rules that can automatically detect the different types of code smells considered in our experiments. Table 6 reports the number of code smells for each system. Only real design flaws that were manually validated in our previous work [19] are considered in this validation.

The upper and lower bounds on the chromosome length used in this study are set to 10 and 350 respectively. Several SBSE problems including software refactoring are characterized by a varying chromosome length. This issue is similar to the problem of bloat control in genetic programming where the goal is to identify the tree size limits. To solve this problem, we performed several trial and error experiments where we assess the average performance of our algorithm using the hypervolume (HV) performance indicator while varying the size limits between 10 and 500 operations.

4.6 Case Studies Summary

Each group of participants received a questionnaire, a manuscript guide to help them to fill the questionnaire, the tools and results to evaluate and the source code of the studied systems as described in the following five scenarios:

In the first scenario, we selected a total of 90 classes from all the systems that include design defects (9 classes to fix per system). Then we asked every participant to manually apply refactorings to improve the quality of the systems by fixing an average of two of these defects. As an outcome of the this scenario, we have a set of expected refactorings and we are able to calculate the differences between the recommended refactorings and the expected ones (manually suggested by the developers).

In the second scenario, we asked the developers to evaluate the suggested solutions of our algorithm. We performed a cross-validation between the ratings of each group to avoid the computation of the *MC* metric being biased by the developer's feedback. Thus, the developers in each group rated results generated by the other developers in the same group.

In the third scenario, we collected a set of 6 bugs per system from the bug reports of the studied release for every project and asked the groups to fix them based on the refactored and non-refactored version. The tasks are completely different and they are applied to different packages/classes of the same version of the systems. Furthermore, the participants did not know if they are working on the system before or after refactoring. We did not follow as well any specific order when asking the developers to work on the tasks. Only 3 out of the 22 participants worked as part of the experiments on the systems before refactoring and then the systems after refactoring. We adapted a counter-balanced design where we asked every developer to fix 2 bugs on the version before refactoring and then 2 other bugs in the version after refactoring. We selected the bugs that require almost the same effort to fix in terms of number of changes, with an average of 15 changes.

In the fourth scenario, we asked the groups to add two simple features to every system before refactoring, and then two other features on the system after refactoring. All the features require almost the same number of changes to be introduced or deleted with an average of 23 code changes per feature. In the third and fourth scenarios, the bugs to fix and features to add are related to the classes that are refactored by the developers when using our tool.
The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study (one faculty member, one postdoc, one Ph.D. student and one Master's student). Participants do not know the particular experiment research questions and the used algorithms.

In the fifth scenario, we asked the participants to use our tool during a period of two hours on the different systems and then we collected their opinions based on a post-study questionnaire. To better understand subjects' opinions with regard to usefulness of our approach in a real setting, the post-study questionnaire was given to each participant after completing the refactoring tasks using our interactive approach and all the techniques considered in our experiments. The questionnaires collected the opinions of the participants about their experience in using our approach compared to manual and fully-automated refactoring tools. We asked participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following statements:

1) The interactive dynamic refactoring recommendations are a desirable feature in integrated development environments (IDEs).
2) The interactive manner of recommending refactorings by our approach is a useful and flexible way to refactor systems compared to fully-automated or manual refactorings.

The remaining questions of the post-study questionnaire were about the benefits and also limitations (possible improvements) of our interactive approach.

4.7 RESULTS AND DISCUSSIONS

Statistical Analysis: Since meta-heuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is based on 30 independent simulation runs for each problem instance. The following statistical tests show that all the comparisons performed between our approach and existing ones are statistically significant based on all the metrics and the systems considered in our experiments.

We used one-way ANOVA statistical test with a 95% confidence level (α = 5%) to find out whether our sample results of different approaches are different significantly. Since one-way ANOVA is an omnibus test, A statistically significant result determines whether three or more group means differ in some undisclosed way in the population.

One-way ANOVA is conducted for the results obtained from each software project to investigate and compare each performance metric (dependent variable) between various studied algorithms (independent variable - groups). We test the null hypothesis (H_0) that population means of each metric are equal for all methods ($\forall$ *Software Projects* : $\mu^{metric}_{M1} = \mu^{metric}_{M2} = ... = \mu^{metric}_{M7}$ *where metric* $\in \{G, NF, MC, PR, RC\}$) against the alternative (H_1) that they are not all equal and at least one method population mean is different.

There are some assumptions for one-way ANOVA test which we assessed before applying the test on the data:

Outliers: There were no outliers in the data, as assessed by inspection of a boxplot for values greater than 1.5 box-lengths from the edge of the box.

Normal Distribution: Some of the dependent variables were not normally distributed for each method, as assessed by Shapiro-Wilk's test. However, the one-way ANOVA is fairly robust to deviation from normality. Since the sample size is more than 15 (there are 30 observations in each group) and the sample sizes are equal for all groups (balanced), non-normality is not an issue and does not affect Type I error.

Homogeneity of variances: The one-way ANOVA assumes that the population variances of the dependent variables are equal for all groups of the independent variable. If the variances are unequal, this can affect the Type I error rate. There was homogeneity of variances, as assessed by Levene's test for equality of variances ($p > 0.05$).

The results of one-way ANOVA tests for all pair of software projects and metrics indicates that The group means were statistically significantly different ($p < .0005$) and, therefore, we can reject the null hypothesis and accept the alternative hypothesis which says there is difference in population means between at least two groups. Table 11 reports the obtained value of F-statistics with the between and within groups degree of freedoms equal to 6 and 203, respectively. In one-way ANOVA, the F-statistic is the ratio of variation between sample means over variation within the samples. The larger value of F-statistics represents the group means are further apart from each other and are significantly different. Also, it shows that the observation within each group are close to the group mean with a low variance within samples. Therefore, a large F-value is required to reject the null hypothesis that the group means are equal. Our obtained F-statistics results are correspond to very small *p*-values.

One-way ANOVA does not report the size of the difference. Therefore, we calculated Eta squared (η^2) which is a measure of the effect size (strength of association) and it estimates the degree of association between the independent factor and dependent variable for the sample. Eta squared is the proportion of the total variance that is attributed to a factor (the "refactoring methods" in this study). Table 12 reports Eta squared values for each pair of software projects and metrics. These values shows to what extent different algorithms are the cause of variability of the metrics. For instance, it says 90% of the total variance of metric *G* for ApacheAnt software project is accounted for by different algorithms effect, not error or other effects.

Table 11
F- value results from one-way ANOVA statistical tests for corresponding software project and metric between different methods.

Software	G	NF	MC	PR	RC
ApacheAnt	335.7	224.8	803.9	379.1	757.1
GanttProject	209.6	593.0	1463.2	379.6	1130.4
JDIFord	135.6	320.3	1036.2	917.3	1032.8
JFreeChart	300.1	776.7	494.7	211.9	663.9
JHotDraw	181.7	408.2	1022.6	158.4	663.8
Log4J	297.8	306.2	477.8	617.9	1044.9
MROIFord	189.5	474.8	1260.2	1228.8	1217.2
Nutch	333.7	361.3	408.1	269.9	658.9
Rhino	121.2	606.2	872.8	598.0	702.2
XercesJ	155.0	214.5	598.0	492.3	633.8

Table 12
Effect Size values (Eta squared (η^2)) for corresponding software project and metric.

Software	G	NF	MC	PR	RC
ApacheAnt	0.908	0.869	0.960	0.918	0.957
GanttProject	0.861	0.946	0.977	0.918	0.971
JDIFord	0.789	0.898	0.966	0.962	0.966
JFreeChart	0.899	0.958	0.936	0.862	0.952
JHotDraw	0.843	0.923	0.968	0.824	0.951
Log4J	0.898	0.900	0.934	0.948	0.969
MROIFord	0.839	0.929	0.972	0.971	0.971
Nutch	0.908	0.914	0.923	0.889	0.951
Rhino	0.782	0.947	0.963	0.946	0.954
XercesJ	0.821	0.864	0.946	0.936	0.949

Tukey post hoc analysis [60] is carried out in order to find out between which group(s) the significant difference is occurred. Basically, it tests all possible group comparisons. However, we only report the results of comparison of our method and others in Table 13. This table represents the point estimate of the difference between each pair of means and is computed from the sample data. Also, it includes the confidence interval showing the difference between population means and is centered on point estimate. If This interval does not include zero, indicates that the difference between the means is statistically significant. The 95% individual confidence level indicates that we can be 95% confident that each interval contains the real difference for that particular comparison. The results shows that all pairwise comparisons between our method and others' for each pair of (software / metric) are statistically significant at the 0.05 level except for G and NF of JFreeChart as their results highlighted in the table of the results. Therefore, the difference between the means of these two metrics,G and NF, for JFreeChart project is 0.

To this end, we used the Vargha-Delaney A measure [57] which is a non-parametric effect size measure. In our context, given the different performance metrics (such as

Table 13
Tukey post hoc analysis results between our method(M1) and others reported by Mean difference and 95% confidence intervals. Label of the methods: **M1** (Our approach)=Interactive+Innovization NSGA-II, **M2**=Innovization NSGA-II, **M3**=Kessentini et al. [19], **M4**=Ouni et al. [9], **M5**=Harman et al. [17], **M6**=O'Keeffe et al. [18], **M7**=Jdeodorant [21]

		Mean difference | 95% Confidence Interval									
Software	**Comparison**	**G**		**NF**		**MC**		**PR**		**RC**	
ApacheAnt	M1-M2	0.10	(0.09,0.12)	0.05	(0.04,0.06)	0.07	(0.06,0.08)	0.09	(0.07,0.10)	0.07	(0.06,0.08)
	M1-M3	0.15	(0.13,0.17)	0.07	(0.06,0.09)	0.12	(0.11,0.13)	0.14	(0.12,0.15)	0.18	(0.17,0.19)
	M1-M4	0.12	(0.10,0.14)	0.05	(0.04,0.07)	0.10	(0.09,0.11)	0.12	(0.10,0.13)	0.13	(0.12,0.14)
	M1-M5	0.21	(0.19,0.23)	0.10	(0.09,0.11)	0.17	(0.16,0.18)	0.13	(0.11,0.14)	0.18	(0.17,0.19)
	M1-M6	0.16	(0.14,0.18)	0.04	(0.03,0.05)	0.14	(0.13,0.15)	0.12	(0.10,0.13)	0.10	(0.09,0.11)
	M1-M7	0.18	(0.17,0.20)	0.15	(0.14,0.17)	0.29	(0.28,0.30)	0.23	(0.21,0.24)	0.28	(0.27,0.29)
GanttProject	M1-M2	0.05	(0.03,0.07)	0.02	(0.01,0.03)	0.11	(0.10,0.12)	0.10	(0.08,0.11)	0.03	(0.02,0.04)
	M1-M3	0.09	(0.07,0.10)	0.06	(0.05,0.07)	0.15	(0.14,0.16)	0.12	(0.10,0.13)	0.08	(0.07,0.09)
	M1-M4	0.07	(0.06,0.09)	-0.04	(-0.05,-0.03)	0.22	(0.21,0.23)	0.12	(0.10,0.13)	0.06	(0.05,0.07)
	M1-M5	0.15	(0.13,0.17)	0.17	(0.16,0.18)	0.30	(0.29,0.31)	0.20	(0.19,0.21)	0.29	(0.28,0.30)
	M1-M6	0.15	(0.13,0.17)	0.14	(0.13,0.15)	0.26	(0.25,0.27)	0.16	(0.14,0.17)	0.10	(0.09,0.11)
	M1-M7	0.12	(0.10,0.14)	0.14	(0.13,0.15)	0.33	(0.32,0.34)	0.18	(0.17,0.19)	0.22	(0.21,0.23)
JDIFord	M1-M2	0.03	(0.01,0.04)	-0.02	(-0.03,-0.01)	0.07	(0.06,0.08)	0.08	(0.07,0.09)	0.06	(0.05,0.07)
	M1-M3	-	-	-	-	-	-	-	-	-	-
	M1-M4	0.03	(0.01,0.04)	-0.03	(-0.04,-0.02)	0.20	(0.19,0.21)	0.13	(0.12,0.14)	0.15	(0.14,0.16)
	M1-M5	-	-	-	-	-	-	-	-	-	-
	M1-M6	0.07	(0.05,0.08)	0.10	(0.09,0.11)	0.20	(0.19,0.21)	0.17	(0.16,0.18)	0.06	(0.05,0.07)
	M1-M7	0.11	(0.09,0.12)	0.08	(0.07,0.09)	0.25	(0.24,0.26)	0.25	(0.24,0.26)	0.27	(0.26,0.28)
JFreeChart	M1-M2	0.09	(0.07,0.11)	0.02	(0.00,0.03)	0.08	(0.07,0.09)	0.07	(0.06,0.08)	0.12	(0.11,0.13)
	M1-M3	0.12	(0.10,0.14)	0.02	(0.01,0.03)	0.14	(0.13,0.15)	0.12	(0.11,0.13)	0.16	(0.15,0.17)
	M1-M4	0.00	(-0.02,0.02)	0.00	(-0.01,0.01)	0.12	(0.11,0.13)	0.10	(0.09,0.11)	0.14	(0.12,0.15)
	M1-M5	0.14	(0.12,0.16)	0.24	(0.22,0.25)	0.14	(0.13,0.16)	0.15	(0.14,0.16)	0.28	(0.26,0.29)
	M1-M6	0.17	(0.15,0.19)	0.09	(0.08,0.10)	0.20	(0.19,0.22)	0.10	(0.09,0.12)	0.16	(0.15,0.17)
	M1-M7	0.13	(0.11,0.15)	0.15	(0.13,0.16)	0.22	(0.21,0.24)	0.12	(0.11,0.13)	0.24	(0.23,0.25)
JHotDraw	M1-M2	0.02	(0.01,0.03)	0.05	(0.04,0.07)	0.08	(0.07,0.09)	0.04	(0.03,0.05)	0.06	(0.04,0.07)
	M1-M3	0.06	(0.05,0.07)	0.04	(0.03,0.05)	0.16	(0.15,0.17)	0.09	(0.08,0.10)	0.10	(0.09,0.12)
	M1-M4	0.03	(0.02,0.04)	-0.02	(-0.03,-0.01)	0.14	(0.13,0.15)	0.07	(0.06,0.08)	0.09	(0.08,0.10)
	M1-M5	0.08	(0.07,0.09)	0.14	(0.13,0.15)	0.30	(0.29,0.31)	0.12	(0.11,0.13)	0.21	(0.20,0.22)
	M1-M6	0.04	(0.03,0.05)	0.14	(0.13,0.15)	0.24	(0.23,0.25)	0.10	(0.09,0.11)	0.17	(0.16,0.18)
	M1-M7	0.11	(0.10,0.12)	0.08	(0.07,0.09)	0.24	(0.23,0.25)	0.10	(0.09,0.12)	0.24	(0.23,0.25)
Log4J	M1-M2	0.08	(0.07,0.10)	0.06	(0.05,0.07)	0.08	(0.07,0.10)	0.03	(0.01,0.04)	0.06	(0.05,0.07)
	M1-M3	0.13	(0.12,0.14)	0.13	(0.12,0.14)	0.12	(0.11,0.13)	0.14	(0.12,0.15)	0.22	(0.21,0.23)
	M1-M4	0.10	(0.09,0.11)	0.06	(0.05,0.07)	0.10	(0.09,0.11)	0.05	(0.03,0.06)	0.08	(0.06,0.09)
	M1-M5	0.14	(0.13,0.15)	0.15	(0.14,0.16)	0.19	(0.18,0.20)	0.19	(0.17,0.20)	0.21	(0.20,0.22)
	M1-M6	0.19	(0.18,0.21)	0.13	(0.12,0.14)	0.16	(0.15,0.17)	0.12	(0.11,0.13)	0.19	(0.18,0.20)
	M1-M7	0.12	(0.10,0.13)	0.15	(0.14,0.16)	0.21	(0.20,0.22)	0.22	(0.21,0.23)	0.31	(0.30,0.32)
MROIFord	M1-M2	0.05	(0.04,0.07)	0.02	(0.01,0.04)	0.08	(0.07,0.09)	0.06	(0.05,0.07)	0.12	(0.11,0.13)
	M1-M3	-	-	-	-	-	-	-	-	-	-
	M1-M4	0.08	(0.07,0.09)	0.03	(0.02,0.04)	0.16	(0.15,0.17)	0.09	(0.08,0.10)	0.16	(0.15,0.17)
	M1-M5	-	-	-	-	-	-	-	-	-	-
	M1-M6	0.12	(0.10,0.13)	0.17	(0.16,0.19)	0.21	(0.20,0.22)	0.13	(0.12,0.14)	0.26	(0.25,0.27)
	M1-M7	0.13	(0.11,0.14)	0.14	(0.13,0.15)	0.29	(0.28,0.30)	0.31	(0.30,0.32)	0.28	(0.27,0.29)
Nutch	M1-M2	0.07	(0.05,0.08)	0.06	(0.04,0.07)	0.07	(0.06,0.08)	0.04	(0.03,0.05)	0.05	(0.04,0.06)
	M1-M3	0.14	(0.12,0.16)	0.11	(0.10,0.12)	0.11	(0.10,0.12)	0.08	(0.07,0.09)	0.14	(0.13,0.15)
	M1-M4	0.10	(0.08,0.12)	0.05	(0.04,0.07)	0.09	(0.08,0.10)	0.08	(0.07,0.09)	0.05	(0.04,0.06)
	M1-M5	0.20	(0.18,0.22)	0.19	(0.18,0.20)	0.18	(0.17,0.19)	0.12	(0.11,0.13)	0.19	(0.18,0.21)
	M1-M6	0.14	(0.12,0.16)	0.15	(0.14,0.16)	0.14	(0.13,0.15)	0.06	(0.05,0.07)	0.17	(0.16,0.18)
	M1-M7	0.17	(0.15,0.19)	0.09	(0.08,0.10)	0.19	(0.18,0.20)	0.16	(0.15,0.17)	0.22	(0.21,0.23)
Rhino	M1-M2	0.06	(0.04,0.08)	0.07	(0.06,0.09)	0.05	(0.03,0.06)	0.04	(0.03,0.05)	0.09	(0.08,0.10)
	M1-M3	0.08	(0.06,0.10)	0.14	(0.13,0.15)	0.09	(0.08,0.10)	0.06	(0.05,0.07)	0.16	(0.15,0.17)
	M1-M4	0.07	(0.05,0.09)	0.12	(0.11,0.13)	0.07	(0.06,0.08)	0.05	(0.04,0.06)	0.13	(0.12,0.15)
	M1-M5	0.13	(0.11,0.15)	0.20	(0.19,0.22)	0.23	(0.21,0.24)	0.22	(0.21,0.23)	0.28	(0.27,0.29)
	M1-M6	0.08	(0.06,0.10)	0.18	(0.17,0.19)	0.14	(0.13,0.15)	0.12	(0.11,0.13)	0.15	(0.14,0.17)
	M1-M7	0.11	(0.09,0.13)	0.24	(0.23,0.26)	0.28	(0.27,0.29)	0.17	(0.16,0.18)	0.23	(0.22,0.24)
XercesJ	M1-M2	0.03	(0.02,0.04)	0.02	(0.01,0.03)	0.06	(0.05,0.07)	0.09	(0.08,0.11)	0.08	(0.07,0.09)
	M1-M3	0.07	(0.06,0.08)	0.02	(0.01,0.04)	0.11	(0.10,0.12)	0.16	(0.15,0.17)	0.13	(0.12,0.14)
	M1-M4	0.04	(0.03,0.05)	-0.02	(-0.03,0.00)	0.08	(0.07,0.09)	0.13	(0.12,0.15)	0.10	(0.09,0.11)
	M1-M5	0.12	(0.11,0.13)	0.12	(0.11,0.13)	0.20	(0.19,0.21)	0.19	(0.18,0.21)	0.22	(0.21,0.23)
	M1-M6	0.08	(0.07,0.09)	0.08	(0.07,0.10)	0.23	(0.21,0.24)	0.16	(0.15,0.17)	0.20	(0.19,0.21)
	M1-M7	0.09	(0.08,0.10)	0.06	(0.05,0.08)	0.17	(0.16,0.18)	0.23	(0.22,0.25)	0.20	(0.19,0.21)

PR, RC, MC, etc.), the A statistic measures the probability that running an algorithm B1 (interactive NSGA-II) yields better performance than running another algorithm B2 (such as [19], [18], [9], etc.). If the two algorithms are equivalent, then A = 0.5. In our experiments, we have found the following results: a) On small and medium scale software projects (GanttProject, Rhino, Log4J and Nutch) our approach is better than all the other algorithms based on all the performance metrics with an A effect size higher than 0.94; and b) On large scale software projects (JDI-Ford, MROI-Ford, Apache Ant, Xerces-J, JHotDraw and JFreeChart), our approach is better than all the other algorithms with an A effect size higher than 0.87.

Results for RQ1a: Fig. 4 summarizes our findings regarding the obtained precision (PR) and recall (RC) results on the 10 systems. We found that a considerable number of proposed refactorings, with an average of more than 82% and 86% respectively in terms of precision and recall, were already applied by the software development team and suggested manually (expected refactorings). The recall scores are higher than precision ones since we found that the refactorings suggested manually by developers are incomplete compared to the solutions provided by our approach. 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 code smells or improving the quality of the code fragments that they frequently modify.

Fig. 4 also confirms the out-performance of our interactive refactoring approach comparing to existing fully-automated techniques and since we confirmed a statistically difference between the means of metrics, we can say that these better results are not obtained by chance. The precision and recall scores were consistent on all the ten systems which confirm that the results are independent from the size of the systems, number of refactorings and number of code smells. The closest results are those obtained by NSGA-II based on innovization (without interaction) and the multi-objective refactoring approach of Ouni et al. This may confirm that the obtained results are more due to the interaction component of our approach. A detailed qualitative discussion will be presented later in RQ1d.

Results for RQ1b: We evaluated also the ability of our approach to fix several types of code smell. Fig. 4 depicts the percentage of fixed code smells (NF). It is higher than 82% on all the ten systems, which is an acceptable score since developers may reject or modify some refactorings that fix some code smells because they do not consider them very important (their goal is not to fix all code smells in the system) or the current version of the code becomes stable. Some systems, such as Rhino and Gantt, have a higher percentage of fixed code smells with an average of more than 88%. This can be explained by the fact that these systems include a higher number of code smells than others.

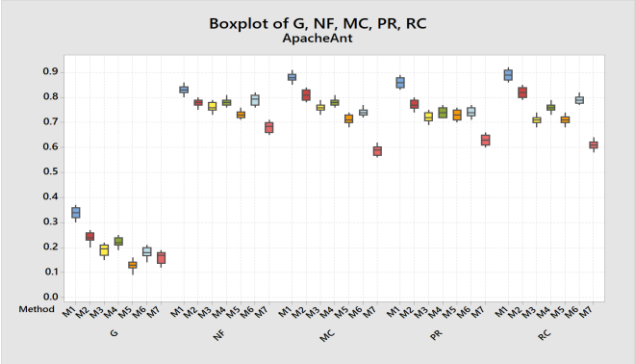
However, the percentage of fixed code smells (NF) is slightly lower than some fully-automated refactoring techniques such as [19] and [9]. This is can be explained by the reason that the main goal of developers during the interaction process is not to fix the maximum number the code smells detected in the system (which was the goal of [19] and [9]) thus they rejected or modified some refactorings suggested by our tool. 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 changes. Therefore, the slight loss in NF is explained by the fact that we are not considering fixing code smells as one of the objectives, and justified by a better improvement in the quality of the refactored system.

Results for RQ1c: Fig. 4 and Table 13 show that the refactorings recommended by the approach and applied by developers improved the quality metrics value (G) of the ten systems. For example, the average quality gain for the two industrial systems was the highest among the ten systems with more than 0.3. The improvements in the quality gain confirm that the recommended refactorings helped to optimize different quality metrics. The functionality attribute has the lowest improvement on the different systems. This may be explained by the fact that refactoring is expected to preserve the behavior of existing functionalities. Our interactive approach clearly also outperforms existing fully-automated techniques. One of the reasons could be related to the fact that the optimization of the quality attributes is considered as part of the fitness functions unlike some of the existing techniques.

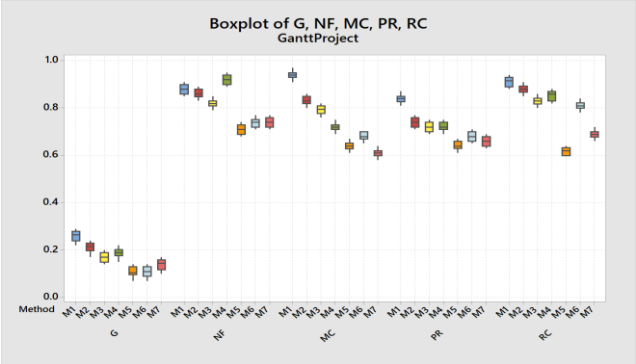
Results for RQ1d: We report the results of our empirical qualitative evaluation (MC) in Fig. 4. As reported in this figure, the majority of the refactoring solutions recommended by our interactive approach were correct and approved by developers. On average, for all of our ten studied projects, 87% of the proposed refactoring operations are considered as semantically feasible, improve the quality and are found to be useful by the software developers of our experiments. The highest MC score is 93% for the Gantt project and the lowest score is 86% for JFreeChart. Thus, it is clear that the results are independent of 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 violating some post-conditions or introducing design incoherence.

Fig. 4 shows that our approach provides significantly higher manual correctness results (MC) than all other approaches having MC scores respectively between 60% and 78%, on average as MC scores on the different systems.

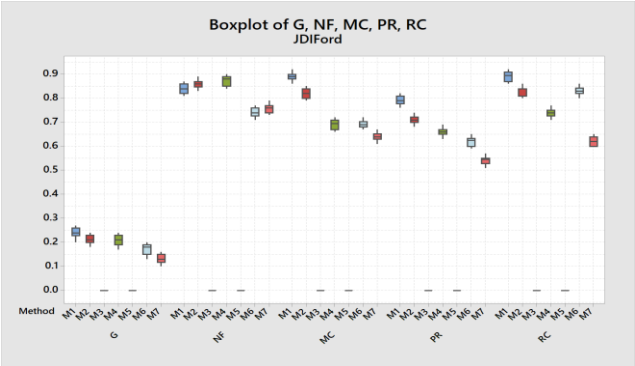
Qualitative Evaluation of RQ1 Results: To provide more qualitative evaluation, we considered some of the feedback that we received from the developers at Ford since they are part of the original developers of these systems. For example, these developers rejected a set of move methods because they believed that these methods should stay in their original class. The original class in this case is responsible for implementing several security constraints (e.g. login information) around database access. The number of security constraints is very high and they were implemented in several methods grouped into one



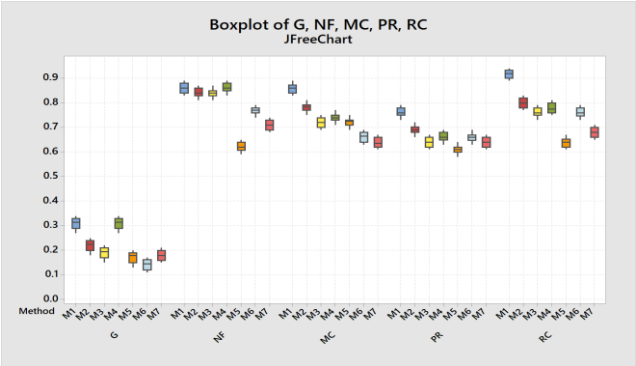
(a) Metrics of Apacheant



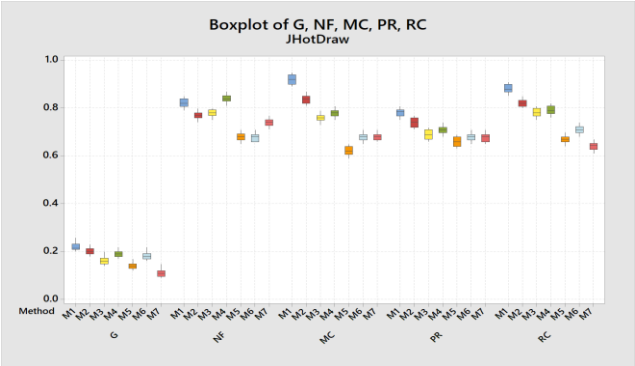
(b) Metrics of GanttProject



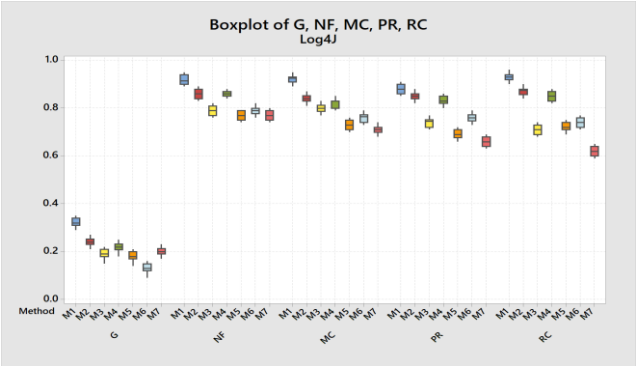
(c) Metrics of JDIFord



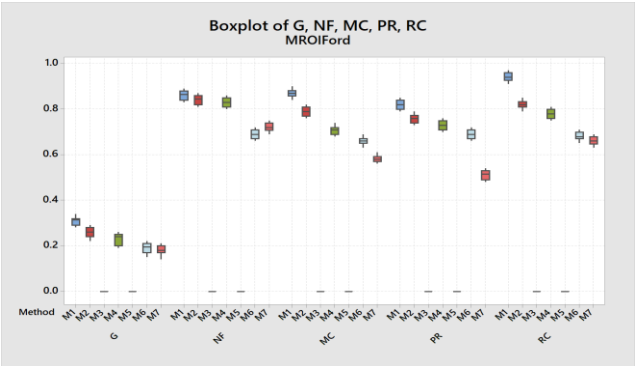
(d) Metrics of JFreeChart



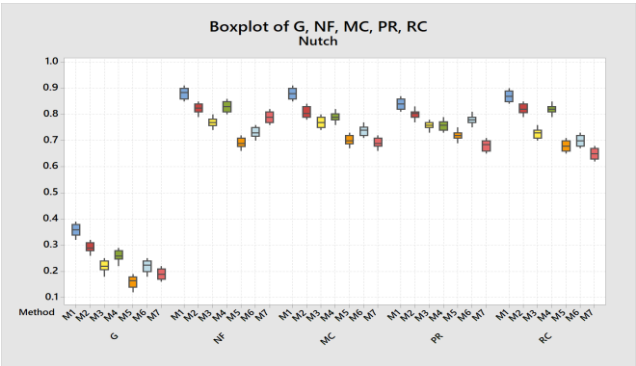
(e) Metrics of JHotDraw



(f) Metrics of Log4J

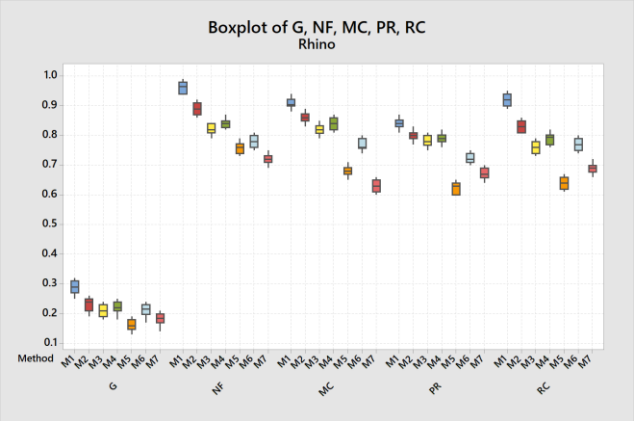


(g) Metrics of MROIFord

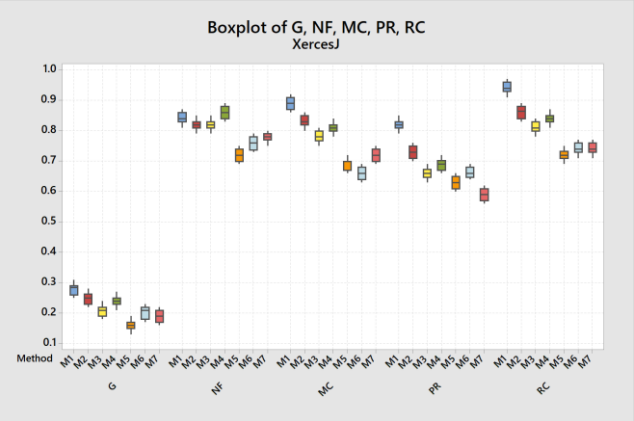


(h) Metrics of Nutch

Figure 4. Boxplots of G, NF, MC, PR, and RC on all the ten systems based on 30 independent runs. (Continue on the next page.) Label of the methods: **M1** (Our approach)=Interactive+Innovization NSGA-II, **M2**=Innovization NSGA-II, **M3**=Kessentini et al. [19], **M4**=Ouni et al. [9], **M5**=Harman et al. [17], **M6**=O'Keeffe et al. [18], **M7**=Jdeodorant [21]



(i) Metrics of Rhino



(j) Metrics of XercesJ

Figure 4. (Continue from the previous page.) Boxplots of G, NF, MC, PR, and RC on all the ten systems based on 30 independent runs. Label of the methods: **M1** (Our approach)=Interactive+Innovization NSGA-II, **M2**=Innovization NSGA-II, **M3**=Kessentini et al. [19], **M4**=Ouni et al. [9], **M5**=Harman et al. [17], **M6**=O'Keeffe et al. [18], **M7**=Jdeodorant [21]

class. Although this set of methods created a blob, the developers assessed that they should stay together because there is a logic behind implementing them in that way, and splitting the behavior may require a redesign of the application.

In another case, the developers elected to extract a class that regroups several methods implementing a parser to extract dealer information. However, this refactoring was not recommended by our approach since the methods were located in a small class that did not contain any code smell or quality violation symptoms. Thus, the refactoring applied by the developers was more based on the features implemented in the methods. This refactoring is hard to recommend even with the considered semantics/textual similarity measures since few comments exist in these methods and furthermore their implementation structures look very different. These observations explain the reasons why some the refactorings recommended by our approach was rejected by the developers and also the differences with those that are manually recommended by the developers.

In general, we found that most of the common patterns in the Pareto front are not individual operations, but a short sequence of refactorings. Thus, we believe that most of these patterns are targeting specific quality issues and hence the applied refactorings are not individual operations but small refactoring patterns. This observation was found to be valid when we manually checked the interactive results of our tool.

A general interesting observation from the experiments is that evolutionary search involves both diversification and convergence, so the question is does innovization emphasize convergence at the cost of sacrificing divergence? We would argue against this, for the following reasons: In the context of our refactoring problem, it is very rare to observe no overlap between non-dominated solutions for several reasons such as the large size of refactoring solutions and the fact that some common quality issues should be fixed (high priority). In fact, at least few quality issues (e.g. code smells) need to be fixed independently from the other objectives. Thus, it is normal to always observe some overlap between the refactoring solutions. Regarding diversification, the ranking of the refactoring solutions is only used after the generation of the Pareto front so this ranking is not part of the fitness function used in the search. The goal is to implicitly explore the front based on the feedback of the developers to identify the region of interest and prioritize the solutions that contain common patterns. We believe that these common patterns distinguish non-dominated solutions from dominated ones. The diversification is not penalized because we do not consider the innovization heuristic as part of the fitness functions but as a post-processing step to prioritize solutions (and not eliminating them).

We compared the results of our approach (M1) and innovization NSGA-II method (M2) in Fig. 4 and Table 13 in order to contrast the impact of interactivity component. The best solution (at the knee point) based on the innovization feature (without interaction) was evaluated based on all studied metrics. The results confirm that our interactive approach outperforms NSGA-II with the only use of innovation (without interaction) in terms of G, NF, MC, PR, and RC. However, the results of NSGA-II with innovization are better than regular multi-objective refactoring approaches (e.g. Ouni et al., etc.) thus it is clear that the positive results of our approach are due to the combination of the two factors: innovization and interactive features.

The superior performance of our interactive approach can be explained by several factors. First, [19], [18] and [17] use only structural indications (quality metrics) to evaluate the refactoring solutions and thus a high number of refactorings lead to a semantically incoherent design. Our approach reduces the number of semantic incoherencies when suggesting refactorings and during the interaction with the developers. Second, the innovization component improved the quality of the suggested refactoring solutions by using an interactive approach as compared to a regular NSGA-II where the developers need to select one solution from the Pareto front that cannot be updated dynamically. Third, JDeodorant proposes some pre-defined patterns to

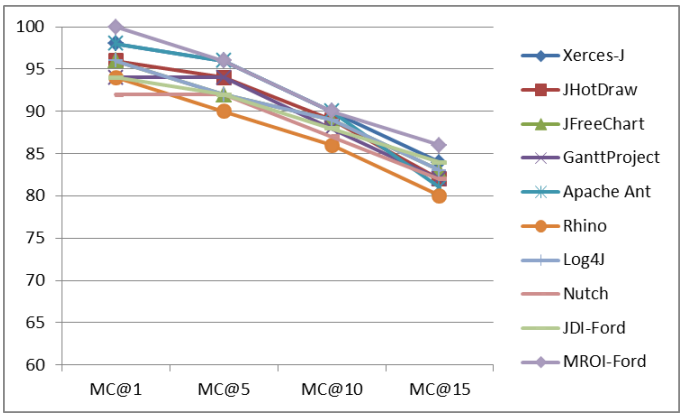


Figure 5. MC@k results on the different systems with k= 1, 5, 10 and 15.

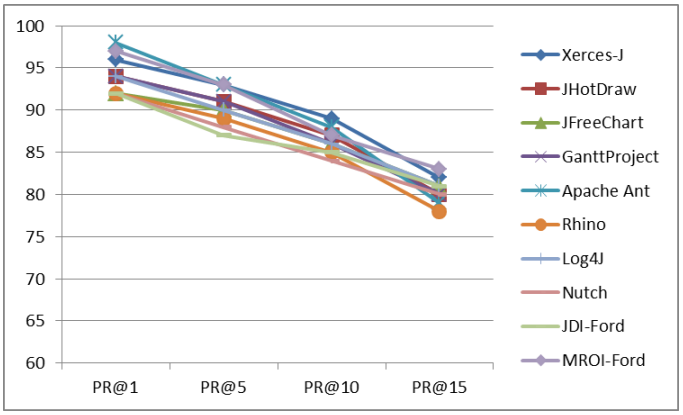


Figure 6. PR@k results on the different systems with k= 1, 5, 10 and 15.

fix some types of code smells that cannot be sometimes generalized.

To summarize and answer RQ1, the experimentation results confirm that our interactive approach helps the participants to refactor their systems efficiently by finding more relevant refactoring solutions and improve the quality of all the ten systems under study. In addition, our interactive approach provides better results, on average, than all of the existing fully-automated refactoring techniques.

Results for RQ2: We evaluated the ability of our approach to help software developers to find quickly good refactorings based on an efficient ranking of the proposed operations. We compared the MC@k and PR@k where k was varied between 1, 5, 10 and 15 as described in Fig. 5 and Fig. 6 where show that the lowest MC@1 is 93% and the highest is 100% on the different ten systems confirming that the highest-ranked refactoring was almost always correct and relevant for the developers.

The MC@15 presents the lowest results, which is to be expected since we evaluated the manual correctness of the top 15 recommended refactorings at several interactions and this increases the probability that it contains few irrelevant refactorings. However, the average MC@15 still could be considered acceptable with an average of more than 81%. The same observations are also valid for the PR@k; however the results are a bit lower than for MC@k. The average PR@k

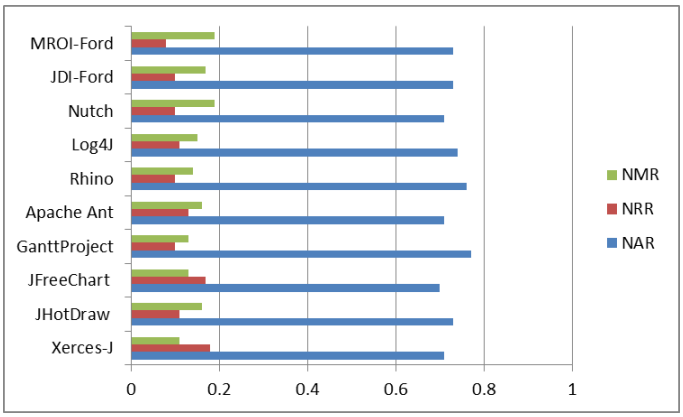


Figure 7. The median NMR, NRR and NAR results in the different systems.

results were respectively 94%, 89%, 84% and 80% for k = 1, 5, 10 and 15. Thus, it is clear that the ranking function used by our interactive approach to explore the Pareto front is efficient.

Considering three other metrics NAR (percentage of accepted refactorings), NMR (percentage of modified refactorings) and NRR (percentage of rejected refactorings), we seek to evaluate the efficiency of our interactive approach to rank the refactorings. We recorded these metrics using a feature that we implemented in our tool to record all the actions performed by the developers during the refactoring sessions. Fig. 7 shows that, on average, more than 71% of the recommended refactorings were applied by the developers. In addition, an average of 17% of the recommended refactorings were modified by the developers, while 12% of the suggested refactorings were rejected by the developers. Thus, it is clear that our recommendation tool successfully suggested a good set of refactorings to apply.

To conclude, our approach efficiently ranked the recommended refactorings and helped software developers to quickly find good refactorings recommendations.

Results for RQ3a: Fig. 8 shows that the time is reduced by 61% and 57% to finalize respectively the two tasks of fixing bugs when programmers worked on the refactored program using our interactive approach. These results outperform the productivity improvements obtained when programmers worked on similar tasks of fixing bugs of the refactored programs by Ouni et al. [9] and Harman et al. [17]. For Ouni et al., the productivity improvements are between 41% and 37% while Harman et al. [17] are between 33% and 31%. The results are correlated with the quality improvements of the evaluated programs, as discussed in the previous sections. Thus, a better quality of the software may increase the productivity of programmers when fixing bugs.

Results for RQ3b: Similar results to RQ3a are obtained for the tasks of adding new features. Fig. 8 shows that the time is reduced by 51% and 48% to finalize respectively the two tasks of adding new features when programmers worked on the refactored program using our interactive approach. These results outperform the productivity improvements obtained when programmers worked on similar tasks of adding features of the refactored programs by Ouni et al.

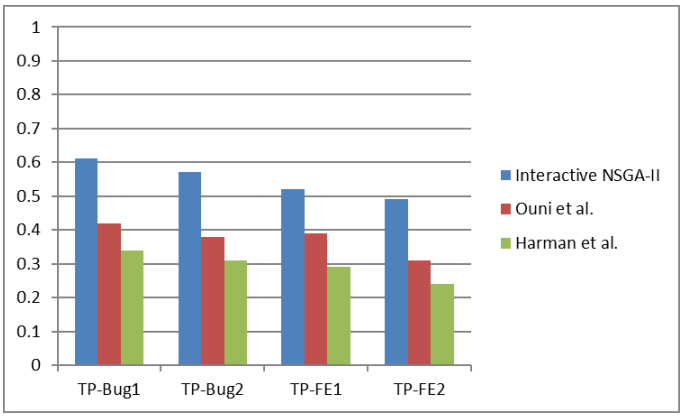


Figure 8. The average productivity difference (TP) results on the different tasks performed by the three groups using our interactive approach, Ouni et al. [9], Harman et al. [17]

[9] and Harman et al. [17]. For Ouni et al., the productivity improvements are between 38% and 31% while Harman et al. [17] are between 29% and 23%. The results are correlated with the quality improvements of the evaluated programs. Thus, a better quality of the software may increase the productivity of programmers when adding new features. Overall, the productivity gain when programmers worked on adding new features is lower than the one observed for fixing bugs. This could be related to the fact that the complexity of adding new features was higher than fixing bugs and the locations where refactorings are introduced.

The metric (TP) to measure the time to perform the different bugs fixing and adding new features task on the systems before and after refactoring included the execution time of the different (interactive and fully-automated) refactoring techniques to generate the new systems after refactoring. While the execution time of our interactive approach is slightly higher than fully-automated approaches with an average of 6 minutes comparing to Ouni et al. and Harman et al. on the different systems used in both scenarios, the overall time that developers spent to perform the new tasks is much lower when working on the new systems after refactoring based on our approach comparing to the state of the art. Thus, the extra manual effort required by our approach is compensated by higher productivity and better accuracy of the results. We believe that the slightly higher execution time by our interactive approach comparing to fully automated search-based refactoring despite the extra-manual effort is explained by the fact that the user feedback can reduce dramatically the search space to converge toward better recommendations. Furthermore, the efficient ranking of refactorings to be inspected by programmers help a lot in reducing the interaction time. Finally, we want to highlight that programmers spend considerable time evaluating long list of refactoring recommendations after the execution of fully-automated approaches which is comparable to the manual interaction effort required during the execution of our interactive approach.

Results for RQ3c: The post-study questionnaire results show the average agreement of the participants was 4.8 and 4.3 based on a Likert scale for the first and second statements (discussed in section 4.6), respectively. This confirms the usefulness of our approach for the software developers considered in our experiments.

We summarize in the following the feedback of the developers. Most of the participants mention that our interactive approach is faster than manual refactoring since they spent a long time with manual refactoring to find the locations where refactorings should be applied. For example, developers spend time when they decide to extract a class to find the methods to move to the newly created class or when they want to move a method then it takes time to find the best target class by manual exploration of the source code. Thus, the developers liked the functionality of our tool that helps them to modify a refactoring and finding quickly the right parameters based on the recommendations.

Our interactive algorithm automatically suggests a list of possible target classes ranked based on the history of changes and semantic similarity. Furthermore, refactorings may affect several locations in the source code, which is a time-consuming task to perform manually, but they can perform it instantly using our tool.

The participants found our tool helpful for both *floss refactoring*, to maintain a good quality design and also for *root canal refactoring* to fix some quality issues such as code smells. The developers justify their conclusions by the following interesting observations about our tool: a) the list of recommended refactorings helps them to choose the desired refactoring very quickly, b) our tool offers them the possibility to modify the source code (to add new functionality) while doing refactoring since the list of recommendations is updated dynamically. So developers can switch between both activities: refactoring and modifying the source code to modify existing functionalities. c) our tool allows developers to access all the functionality of the IDE (e.g., Eclipse). d) the suggested refactorings by our interactive tool can fix code smells (root canal refactoring) or improve some quality metrics (floss canal refactoring) due to the use of the multi-objective approach.

Another important feature that the participants mention is that our interactive approach allows them to take the advantages of using multi-objective optimization for software refactoring without the need to learn anything about optimization and exploring explicitly the Pareto front to select one ideal solution. The implicit exploration of the Pareto front in an interactive fashion represents an important advantage of our tool along with the dynamic update of the recommended list of refactoring using innovization. In fact, the developers found a lot of difficulties using the multi-objective tool of [54] to explore the Pareto front to find a good refactoring solution. In addition, they did not appreciate the long list of refactoring suggested by [54] since they want to take control of modifying and rejecting some refactorings. In addition, the validation of this long list of refactorings is time-consuming. Thus, they appreciate that our tool suggests refactoring one by one and update the list based on the feedback of developers.

The participants also suggested some possible improvements to our interactive approach. Some participants believe that it will be very helpful to extend the tool by adding a new feature to apply automatically some regression testing techniques to generate test cases to test applied refactorings. Another possibly suggested improvement is to use some

visualization techniques to evaluate the impact of applying a refactoring sequence.

5 THREATS TO VALIDITY

Following the methodology proposed by [58], there are four types of threats that can affect the validity of our experiments. We consider each of these in the following paragraphs.

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 creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error, which is commonly used in the SBSE community [59]. However, it would be an interesting perspective to design an adaptive parameter tuning strategy [56] for our approach so that parameters are updated during the execution in order to provide the best possible performance. In addition, our multi-objective formulation treats the different types of refactoring with the same weight in terms of complexity when calculating one of the fitness functions. However, some refactoring types can be more complex than others to apply by developers.

Internal validity is concerned with the causal relationship between the treatment and the outcome. We dealt with internal threats to validity by performing 30 independent simulation runs for each problem instance. This makes it highly unlikely that the observed results were caused by anything other than the applied multi-objective approach. The second internal threat is related to the variation of correctness 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. Regarding the selected participants, we have taken precautions to ensure that our participants represent a diverse set of software developers with experience in refactoring, and also that the groups formed had, in some sense, a similar average skill set in the refactoring area. The results obtained by the developers from Ford and those by the graduate students are consistent. The evaluated open source and industrial systems provided similar conclusions in our experiments. The industrial systems are mainly evaluated by the original developers and the results are still consistent with the open source systems.

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 fully-automated refactoring approaches, 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 in terms of correctness and readability. We considered in our experiments the majority of votes from the developers. We selected the majority of votes as the technique to aggregate the date since it is similar to real-world situations. Almost all of our industrial collaborators in the refactoring area are selecting major refactoring strategies based on discussions between the architects to adopt the best alternative. The architects discuss several possibilities to refactor the current architecture and they will decide the best one based on the majority. We adopted this strategy for our experiments to simulate real-world scenarios. For the selection threat, the participant diversity in terms of experience could affect the results of our study. We addressed the selection threat by giving a lecture and examples of refactorings already evaluated with arguments and justification. For the fatigue threat, we did not limit the time to fill the questionnaire and we also sent the questionnaires to the participants by email and gave them the required time to complete each of the required tasks. We believe that one of the principal strengths of our approach is the interaction component with the developer since many aspects of software quality are subjective and impossible to formalize precisely using quality metrics alone. The interaction with the developer (i.e., developer feedback) can help to improve the refactoring recommendations, by critically augmenting the objective metric values with subjective developer insight. However, a better fitness function may indeed reduce the interaction effort. Thus, the use of the QMOOD model in a fitness function can be considered as a possible threat since the use of quality metrics to solutions' evaluation is subjective.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on eight different widely used open-source systems belonging to different domains and having different sizes, and two industrial projects. However, we cannot assert that our results can be generalized to other applications, and to other practitioners. Future replications of this study are necessary to confirm our findings. Further empirical studies are also required to deeply evaluate the performance of the interactive NSGA-II using the same problem formulation. The first threat is the limited number of participants and evaluated systems, which externally threatens the generalizability of our results. In addition, our study was limited to the use of specific refactoring types and types of code smell. Future replications of this study are necessary to confirm our findings.

6 RELATED WORK

We summarize, in the following, existing studies in the area of software refactoring. We classify them into three categories: manual, automated and interactive refactoring.

6.1 Manual Refactoring

We start, this section, by summarizing existing manual approaches for software refactoring. In Fowler's book [23]

a non-exhaustive list of low-level design problems in source code has been defined. For each type of code smell, a list of possible refactorings is suggested that can be applied by the developers. Du Bois et al. [24] start from the hypothesis that refactoring opportunities correspond to those that improve cohesion and coupling metrics, and use this to perform an optimal distribution of features over classes. They analyze how refactorings manipulate coupling and cohesion metrics, and how to identify refactoring opportunities that improve these metrics. However, this approach is limited to only certain refactoring types and a small number of quality metrics. Murphy-Hill et al. [25, 26] proposed several techniques and empirical studies to support refactoring activities. In [26, 27], the authors proposed new tools to assist software developers in applying refactoring such as selection assistant, box view, and refactoring annotation based on structural information and program analysis techniques.

Recently, Ge and Murphy-Hill [28] have proposed a new refactoring tool called GhostFactor that allows the developer to transform code manually, but checks the correctness of the transformation automatically. BeneFactor [29] and WitchDoctor [30] can detect manual refactorings and then complete them automatically. Tahvildari et al. [31] also propose a framework of object-oriented metrics used to suggest to the software developer refactoring opportunities to improve the quality of an object-oriented legacy system. Dig [32] proposes an interactive refactoring technique to improve the parallelism of software systems. Other contributions are based on rules that can be expressed as assertions (invariants, pre- and post-conditions). All these techniques are more concerned around the correctness of manually applied refactorings rather than interactive recommendations. The use of invariants has been proposed to detect parts of the program that require refactoring [33]. In addition, Opdyke [34] has proposed the definition and use of pre- and post-conditions with invariants to preserve the behavior of the software when applying refactorings. Hence, behavior preservation is based on the verification/satisfaction of a set of pre- and post-condition. All these conditions are expressed as first-order logic constraints expressed over the elements of the program.

To summarize, manual refactoring is a tedious task for developers that involves exploring the software system to find the best refactoring solution that improves the quality of the software and fix design defects.

6.2 Automated Refactoring

To automate refactoring activities, new approaches have been proposed. JDeodorant [35] is an automated refactoring tool implemented as an Eclipse plug-in that identifies certain types of design defect using quality metrics and then proposes a list of refactoring strategies to fix them. Search-based techniques [36] are widely studied to automate software refactoring and consider it as an optimization problem, where the goal is to improve the design quality of a system based mainly on a set of software metrics. The majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng et al. [37] have proposed a single-objective optimization approach using a genetic algorithm to suggest a list of refactorings to improve software quality. The work of O'Keeffe et al. [18] uses various local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. They use the QMOOD metrics suite [38] to evaluate the improvement in quality.

Kessentini et al. [19] have proposed single-objective combinatorial optimization using a genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected in the source code. Kilic et al. [40] explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions. Harman et al. [17] have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. Ouni et al. [41] proposed also a multi-objective refactoring formulation that generates solutions to fix code smells. Cinnide et al. [42] have proposed a multi-objective search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. They have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics.

The majority of existing multi-objective refactoring techniques propose as output a set of non-dominated refactoring solutions (the Pareto front) that find a good trade-off between the considered maintainability objectives. This leaves it to the software developers to select the best solution from a set of possible refactoring solutions, which can be a challenging task as it is not natural for developers to express their preferences in terms of a fitness functions value. Thus, the exploration of the Pareto front is still performed manually, which limits the use of multi-objective search techniques to address software engineering problems. An intelligent exploration of the Pareto front is required to expand the applicability of multi-objective techniques for search-based software engineering problems.

In summary, developers should accept the entire refactoring solution and existing tools do not provide the flexibility to adapt the suggested solution in existing fully-automated refactoring techniques. Furthermore, existing automated refactoring tools execute the whole algorithm again to suggest new refactorings after a number of code changes are introduced by developers, rather than simply trying to update the proposed solutions based on the new code changes. While automation is important, it is essential to understand the points at which human oversight, intervention, and decision-making should impact on automation. Human developers might reject changes made by any automated programming technique. Especially if they feel that they have little control, there will be a natural reluctance to trust and use the automated refactoring tool [6].

6.3 Interactive Refactoring

Interactive techniques have been generally introduced in the literature of Search-Based Software Engineering and especially in the area of software modularization. Hall et al. [43] treated software modularization as a constraint

satisfaction problem. The idea of this work is to provide a baseline distribution of software elements using good design principles (e.g. minimal coupling and maximal cohesion) that will be refined by a set of corrections introduced interactively by the designer.

The approach, called SUMO (Supervised Re-modularization), consists of iteratively feeding domain knowledge into the remodularization process. The process is performed by the designer in terms of constraints that can be introduced to refine the current modularizations. Initially, the system begins with generating a module dependency graph from an input system. This dependency is based on the correlation between software elements (coupling between methods, shared attributes etc.). Possible modularizations are then generated from the graph using multiple simulated authoritative decompositions. Then, using a clustering technique called Bunch, an initial set of clusters is generated that serves as an input to SUMO.

The SUMO algorithm provides a hypothesized modularization to the user, who will agree with some relations, and disagree with others. The user's corrections are then integrated into the modularization process, to generate a more satisfactory modularization. The SUMO algorithm does not necessarily rely on clustering techniques, but it can benefit from their output as a starting point for its refinement process.

Bavota et al. [44] presented the adoption of single objective interactive genetic algorithms in software re-modularization process. The main idea is to incorporate the user in the evaluation of the generated remodularizations. Interactive Genetic Algorithms (IGAs) extend the classic Genetic Algorithms (GAs) by partially or entirely involving the user in the determination of the solutions fitness function. The basic idea of the Interactive GA (IGA) is to periodically add a constraint to the GA such that some specific components shall be put in a given cluster among those created so far. Initially, the IGA evolves similarly to the non-interactive GA.

After a user-defined set of iterations, the individual with the highest fitness value is selected from the population set (in the case of single-objective GA) or from the first front (in the case of multi-objective GA) and presented to the user. After analyzing the current modularization, the user provides feedback in terms of constraints dictating for example, that a specific element needs to be in the same cluster as another one. Although user feedback is important in guaranteeing convergence, it is essential not to overload the user by asking for a decision about all the current relationships between elements, especially for a large system.

Overall, the above existing studies of interactive remodularization are limited to few types of refactoring such as moving classes between packages and splitting packages. Furthermore, the interaction mechanism is based on the manual evaluation of proposed remodularization solutions which could be a time-consuming process. The proposed interactive remdoularization techniques are also based on a mono-objective algorithm and did not consider multiple objectives when evaluating the solutions. A recent study [45] extended our previous work [22] to propose an interactive search based approach for refactoring recommendations. The developers have to specify a desired design at the architecture level then the proposed approach try to find the relevant refactorings that can generate a similar design to the expected one. In our work, we do not consider the use of a desired design, thus developers are not required to manually modify the current architecture of the system to get refactoring recommendations. Furthermore, developers maybe interested to change the architecture mainly when they want to introduce an extensive number of refactorings that radically change the architecture to support new features.

Several possible levels of interaction are not considered by existing refactoring techniques. It is easy for developers to identify large classes or long methods that should be refactored, but they find it is difficult, in general, to locate a target class when applying a move method refactoring [20]. In addition, existing refactoring tools do not update their recommended refactoring solutions based on the software developer's feedback such as accepting, modifying or rejecting certain refactoring operations.

To address the above-mentioned limitations, we proposed in this paper a new way for software developers to refactor their software systems as a sequence of transformations based on different levels of interaction, implicit exploration of non-dominated refactoring solutions and dynamic adaptive ranking of the suggested refactorings.

7 CONCLUSION AND FUTURE WORK

We proposed, in this paper, an interactive recommendation tool for software refactoring that dynamically adapts and suggests refactorings to developers based on their feedback and introduced code changes. Our interactive approach allows developers to benefit from search-based refactoring tools without explicitly involving any knowledge about optimization and multi-objective optimization algorithms. In fact, the exploration of the non-dominated refactoring solutions is implicitly performed based on the interaction with the developers. The feedback received from the developers is used to reduce the search space and converge to better solutions. To evaluate the effectiveness of our tool, we conducted a human study on a set of software developers who evaluated the tool and compared it with the state-of-the-art refactoring techniques. Our evaluation results provide strong evidence that our tool improves the applicability of software refactoring, and proposes a novel way for software developers to refactor their systems interactively.

Future work involves validating our technique with additional refactoring types, programming languages and code smell types in order to conclude about the general applicability of our methodology. Furthermore, we only focused, in this paper, on the recommendation of refactorings. We plan to extend the approach by automating the test and verification of applied refactorings. In addition, we will consider the importance of code smells during the correction step using previous code changes, class complexity, etc. Another future research direction related to our work is to build an interactive software engineering framework that applies a similar approach to other software engineering problems such as the next release problem.

REFERENCES

[1] Brown, W.H., Malveau, R.C., and Mowbray, T.J.: AntiPatterns: refactoring software, architectures, and projects in crisis', 1998

[2] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D.: Refactoring: Improving the design of existing programs'. Proc. Conference Name, Conference Location, 1999 pp.

[3] Opdyke, W., and Johnson, R.: Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications. ACM.' (1990. 1990)

[4] Murphy-Hill, E., and Black, A.: Refactoring Tools: Fitness for Purpose', IEEE software, 2008, 25, (5), pp. 38-44

[5] Negara, S., Chen, N., Vakilian, M., Johnson, R., and Dig, D.: A Comparative Study of Manual and Automated Refactorings. In the European Conference on Object-Oriented Programming. 552-576' (2013. 2013)

[6] Murphy-Hill, E., Parnin, C., and Black, A.: E. R. Murphy-Hill, C. Parnin, and A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering, vol. 38, no. 1, pp. 518.' (2013. 2013)

[7] Kim, M., Zimmermann, T., and Nagappan, N.: A field study of refactoring challenges and benefits'. Proc. Conference Name, Conference Location, 2012 pp. 50

[8] Yang, L., Kamiya, T., Sakamoto, K., Washizaki, H., and Fukazawa, Y.: RefactoringScript: A Script and Its Processor for Composite Refactoring. International Conference on Software Engineering and Knowledge Engineering' (2014. 2014)

[9] Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., and Deb, K.: Multi-criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study', ACM Transactions on Software Engineering and Methodology (TOSEM), 2016

[10] Murphy-Hill, E., and Black, A.: Programmer-Friendly Refactoring Errors', IEEE Transactions on Software Engineering, 2012, 38, (6), pp. 1417-1431

[11] Stroggylos, K., and Spinellis, D.: Refactoring–Does It Improve Software Quality? Proceedings of the 5th International Workshop on Software Quality, IEEE Computer Society.' (2007. 2007)

[12] Counsell, S., Hierons, R.M., Hamza, H., Black, S., and Durrand, M.: Exploring the eradication of code smells: An empirical and theoretical perspective. Advances in Software Engineering.' (2010. 2010)

[13] Beck, K.: Test-driven development: by example. Addison-Wesley Professional' (2003. 2003)

[14] Fowler, M., and Highsmith, J.: The agile manifesto. Software Development. pp. 28-32.' (2001. 2001)

[15] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6(2): 182-197' (2002. 2002)

[16] Deb, K., and Srinivasan, A.: Innovization: innovating design principles through optimization. Proceedings of the 8th annual conference on Genetic and evolutionary computation. Seattle, Washington, USA, ACM: 1629-1636.' (2006. 2006)

[17] Harman, M., and Tratt, L.: Pareto optimal search based refactoring at the design level. Proceedings of the 9th annual conference on Genetic and evolutionary computation. London, England, ACM: 1106-1113.' (2007. 2007)

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

[19] Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., and Ouni, A.: Design Defects Detection and Correction by Example. IEEE 19th International Conference on Program Comprehension (ICPC)' (2011. 2011)

[20] Ouni, A., Kessentini, M., Sahraoui, H., and Hamdi, M.S.: The use of development history in software refactoring using a multi-objective evolutionary algorithm. Proceedings of the 15th annual conference on Genetic and evolutionary computation. Amsterdam, The Netherlands, ACM: 1461-1468.' (2013. 2013)

[21] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A.: JDeodorant: identification and application of extract class refactorings. 33rd International Conference on Software Engineering' (2011. 2011)

[22] Mkaouer, M.W., Kessentini, M., Bechikh, S., Deb, K., and Cinnide, M.: Recommendation system for software refactoring using innovization and interactive dynamic optimization'. Proc. Conference Name, Conference Location, 2014 pp. 331-336

[23] Fowler, M., Beck, Kent, Brant, J., Opdyke, W., and Roberts, D.: Refactoring Improving the design of existing code. Addison Wesley, ISBN 978-0201485677' (1999. 1999)

[24] Du Bois, B., Demeyer, S., and Verelst, J.: Refactoring - improving coupling and cohesion of existing code. 11th Working Conference on Reverse Engineering.' (2004. 2004)

[25] Wang, Hanzhang, Marouane Kessentini, and Ali Ouni. "Bi-level identification of web service defects." International Conference on Service-Oriented Computing. Springer, Cham, 2016.

[26] Mansoor U, Kessentini M, Wimmer M, Deb K. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. Software Quality Journal. 2017 Jun 1;25(2):473-501.

[27] Mkaouer MW, Kessentini M, Bechikh S, Cinnéide MÓ, Deb K. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empirical Software Engineering. 2016 Dec 1;21(6):2503-45.

[28] Ouni, Ali, Raula Gaikovina Kula, Marouane Kessentini, and Katsuro Inoue. "Web service antipatterns detection using genetic programming." In Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1351-1358. ACM, 2015.

[29] Kessentini, M., Wimmer, M., Sahraoui, H. and Boukadoum, M., 2010, June. Generating transformation rules from examples for behavioral models. In Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications (p. 2). ACM.

[30] Kessentini, Marouane, Houari Sahraoui, and Mounir Boukadoum. "Example-based model-transformation testing." Automated Software Engineering 18, no. 2 (2011): 199-224.

[31] Ouni, Ali, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. "Multi-criteria code refactoring using search-based software engineering: An industrial case study." ACM Transactions on Software Engineering and Methodology (TOSEM) 25, no. 3 (2016): 23.

[31] Ouni A, Kessentini M, Sahraoui H, Inoue K, Hamdi MS. Improving multi-objective code-smells correction using development history. Journal of Systems and Software. 2015 Jul 1;105:18-39.

[32] Kessentini M, Bouchoucha A, Sahraoui H, Boukadoum M. Example-based sequence diagrams to colored petri nets transformation using heuristic Search. InEuropean Conference on Modelling Foundations and Applications 2010 Jun 15 (pp. 156-172). Springer, Berlin, Heidelberg.

[33] Kataoka, Y., Ernst, M.D., Griswold, W.G., and Notkin, D.: Automated Support for Program Refactoring Using Invariants. International Conference in Software Maintenance. pp. 736-743.' (2001. 2001)

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

[35] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A.: JDeodorant: identification and application of extract class refactorings'. Proc. Conference Name, Conference Location, 21-28 May 2011 2011 pp. 1037-1039

[36] Harman, M., Mansouri, S.A., and Zhang, Y.: Search-based software engineering: Trends, techniques and applications', ACM Computing Surveys (CSUR), 2012, 45, (1), pp. 11

[37] Seng, O., Stammel, J., and Burkhart, D.: Search-based determination of refactorings for improving the class structure of object-oriented systems. Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM.' (2006. 2006)

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

[39] Fatiregun, D., Harman, M., and Hierons, R.M.: Evolving transformation sequences using genetic algorithms'. Proc. Conference Name, Conference Location, 2004 pp. 65-74

[40] Kilic, H., Koc, E., and Cereci, I.: Search-based parallel refactoring using population-based direct approaches. Search Based Software Engineering, Springer: 271-272.' (2011. 2011)

[41] Ouni, A., Kessentini, M., Sahraoui, H., and Hamdi, M.S.: Search-based refactoring: Towards semantics preservation. 28th IEEE International Conference on Software Maintenance (ICSM)' (2012. 2012)

[42] Cinnide, M., Tratt, L., Harman, M., Counsell, S., and Hemati Moghadam, I.: Experimental assessment of software metrics using automated refactoring. Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, ACM.' (2012. 2012)

[43] Hall, M., Walkinshaw, N., and McMinn, P.: Supervised Software Modularization. 28th IEEE International Conference on Software

Maintenance. pp. 23-30.' (2012. 2012)
[44] Bavota, G., and Carnevale, F.: Putting the developer in a loop:an interactive GA for Software Re-modularization. Search based soft- ware engineering, Lecture notes in computer science volume 7515. pp 75-89' (2012. 2012)
[45] Lin, Y., Peng, X., Cai, Y., Dig, D., Zheng, D., and Zhao, W.: In- teractive and Guided Architectural Refactoring with Search-Based

Recommendation', International Symposium on the Foundations of Software Engineering (FSE 2016), Accepted, 2016
[46] Deb, K., Sinha, A., Korhonen, P.J., and Wallenius, J.: An interactive evolutionary multiobjective optimization method based on progressively approximated value functions. IEEE Transactions on Evolutionary Computation. 14(5): 723-739.' (2010. 2010)
[47] Karahan, I., and Kksalan, M.: A territory defining multiobjective evolutionary algorithms and preference incorporation. IEEE Transactions on Evolutionary Computation. 14(4): 636-664.' (2010. 2010)
[48] Fernandez, E., Lopez, E., Bernal, S., Coello Coello, C.A., and Navarro, J.: Evolutionary multiobjective optimization using an outranking-based dominance generalization. Computers & Operations Research 37(2): 390-395.' (2010. 2010)
[49] Wagner, T., and Trautmann, H.: Integration of preferences in hypervolume-based multiobjective evolutionary algorithms by means of desirability functions. IEEE Transactions on Evolutionary Computation. 14(5): 688-701.' (2010. 2010)
[50] Farina, M., Deb, K., and Amato, P.: Dynamic multiobjective optimization problems: test cases, approximations, and applications. IEEE Transactions on Evolutionary Computation 8(5): 425-442.' (2004. 2004)
[51] Mkaouer, W., Kessentini, M., Bechikh, S., Cinnide, M., and Deb, K.: On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empirical Software Engineering: 1-43.' (2015. 2015)
[52] Jensen, A.C., and Cheng, B.H.C.: On the use of genetic programming for automated refactoring and the introduction of design patterns. Proceedings of the 12th annual conference on Genetic and evolutionary computation. Portland, Oregon, USA, ACM: 13' (2010. 2010)
[53] Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., and Ouni, A.: Many-objective software remodularization using NSGA-III. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(3): 17.' (2015. 2015)
[54] Ouni, A., Kessentini, M., Sahraoui, H., and Boukadoum, M.: Maintainability defects detection and correction: a multi-objective approach. Automated Software Engineering 20(1): 47-79.' (2012. 2012)
[55] Arcuri, A., and Fraser, G.: Parameter tuning or default values? An empirical investigation in search-based software engineering. Empirical Software Engineering 18(3): 594-623.' (2013. 2013)
[56] Arcuri, A., and Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. 33rd International Conference on Software Engineering (ICSE).' (2011. 2011)
[57] 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 25(2): 101-132.' (2000. 2000)
[58] Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., and Wessln, A.: Experimentation in software engineering, Springer Science & Business Media.' (2012. 2012)
[59] Harman, M., Mansouri, S.A., and Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys 45(1): 1-61.' (2012. 2012)
[60] Keppel, G., and T. D. Wickens. "Simultaneous comparisons and the control of type I errors." Design and analysis: A researchers handbook. 4th ed. Upper Saddle River (NJ): Pearson Prentice Hall. p (2004): 111-130.
[61] http://kessentini.net/tse18
[62] Jackson, Robert R., Chris M. Carter, and Michael S. Tarsitano. "Trial-and-error solving of a confinement problem by a jumping spider, Portia fimbriata." Behaviour 138.10 (2001): 1215-1234.