Interactive Multi-Objective Refactoring via Decision and Objective Space Exploration

Soumaya Rebai, Vahid Alizadeh, Marouane Kessentini, Houcem Fehri and Rick Kazman

Abstract—Due to the conflicting nature of quality measures, there are always multiple refactoring options to fix quality issues. Thus, interaction with developers is critical to inject their preferences. While several interactive techniques have been proposed, developers still need to examine large numbers of possible refactorings, which makes the interaction time-consuming. Furthermore, existing interactive tools are limited to the ”objective space” to show developers the impacts of refactorings on quality attributes. However, the “decision space” is also important since developers may want to focus on specific code locations. In this paper, we propose an interactive approach that enables developers to pinpoint their preference simultaneously in the objective (quality metrics) and decision (code location) spaces. Developers may be interested in looking at refactoring strategies that can improve a specific quality attribute, such as extendibility (objective space), but such strategies may be related to different code locations (decision space). A plethora of solutions is generated at first using multi-objective search that tries to find the possible trade-offs between quality objectives. Then, an unsupervised learning algorithm clusters the trade-off solutions based on their quality metrics, and another clustering algorithm is applied within each cluster of the objective space to identify solutions related to different code locations. The objective and decision spaces can now be explored more efficiently by the developer, who can give feedback on a smaller number of solutions. This feedback is then used to generate constraints for the optimization process, to focus on the developer's regions of interest in both the decision and objective spaces. A manual validation of selected refactoring solutions by developers confirms that our approach outperforms state of the art refactoring techniques.

Index Terms—Search based software engineering, refactoring, multi-objective search, clustering

1 Introduction

With the ever-growing size and complexity of software projects, there is a high demand for efficient refactoring [1] tools to improve software quality, reduce technical debt, and increase developer productivity. However, refactoring software systems can be complex, expensive, and risky [2], [3], [4]. A recent study [5] shows that developers are spending considerable time struggling with existing code (e.g., understanding, restructuring, etc.) rather than creating new code, and this may have a harmful impact on developer creativity.

Various tools for code refactoring have been proposed during the past two decades ranging from manual support [6], [7], [8] to fully automated techniques [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]. While these tools are successful in generating correct code refactorings, developers are still reluctant to adopt these refactorings. This reluctance is due to the tools' poor consideration of context and developer preferences when finding refactorings [11], [19], [20], [21]. In fact, the preferences of developers ranging from quality improvements to code locations, are still not well supported by existing tools and a large number of refactorings are recommended, in general, to fix the majority of the quality issues in the system.

In our recent survey, supported by an NSF I-Corps project, with 127 experienced developers in software maintenance at 38 medium and large companies (Google, eBay, IBM, Amazon, etc.) [22], [23], 84% of face-to-face interviewees confirmed that most of the existing automated refactoring tools detect and recommend hundreds of code-level issues (e.g., antipatterns and low quality metrics/attributes) and refactorings. However, these tools do not specify where to start or how they relate to a developer's context (e.g., the recently changed files) and preferences in terms of quality targets. This observation is consistent with another recent study [24]. Furthermore, refactoring is a human activity that cannot be fully automated and requires a developer's insight to accept, modify, or reject recommendations because developers understand their problem domain and may have a clear target design in mind. Several studies reveal that automated refactoring does not always lead to the desired architecture even when quality issues are properly detected, due to the subjective nature of software design choices [15], [17], [18], [20], [25], [26], [27]. However, manual refactoring is often error-prone and time-consuming [28], [29].

Several studies have been proposed recently to have developers interactively evaluate refactoring recommendations [22], [23], [24], [30], [31]. The developers provide feedback about the refactored code and may introduce manual changes to some of the recommendations. However, this interactive process can be expensive since developers must evaluate a large number of possible refactorings and eliminate irrelevant ones. Both interactive and automated refactoring approaches have to deal with the challenge of considering many quality attributes for the generation of refactoring solutions. One of the most commonly used quality attributes are the ones of the QMOOD model including

- *Soumaya Rebai,Vahid Alizadeh,Marouane Kessentini and Houcem Fehri are with the Department of Computer and Information Science, University of Michigan-Dearborn.*
  E-mail: firstname@umich.edu
- *Rick Kazman is a Professor at the University of Hawaii and a Principal Researcher at the Software Engineering Institute of Carnegie Mellon University.*

Manuscript received on November 2019

reusablitiy, extensibility, effectiveness, etc [32]. QMOOD was empirically validated by many studies, based on hundreds of open source and industry projects, to ensure that they are associated with the qualities they are supposed to measure and that they are also conflicting [25], [33], [34].

Refactoring studies have either aggregated these quality metrics to evaluate possible code changes or treated them separately to find trade-offs [15], [18], [24], [25], [26], [27], [31], [35]. However, it is challenging to define weights upfront for the quality objectives since developers are often unable to express them. Furthermore, the number of possible trade-offs between quality objectives is large, which makes developers reluctant to look at many refactoring solutions—a time-consuming and confusing process. The closest work to this study of Alizadeh et al. [22], [23] shows that even the clustering of non-dominated refactoring solutions based on quality metrics will still generate a considerable number of refactorings to explore. Developers, in practice, combine the use of quality metrics and code locations/files to target when deciding which refactoring to apply. However, existing refactoring tools are not enabling the interactive exploration of both quality metrics and code locations during the refactoring process. The search is beyond just filtering the refactorings but how can the algorithm find better recommendations after understanding the preferences of the users and giving them a good understanding on how the refactorings are distributed if they are interested in improving specific quality objectives.

In this paper, we propose an interactive approach that combines multi-objective search, interactive optimization, and unsupervised learning to reduce developer effort in exploring both objective spaces (quality attributes) and decision spaces (files). As a first step, a multi-objective search algorithm, based on NSGA-II [36], is executed to find a compromise between the multiple conflicting quality objectives and generates a set of non-dominated refactoring solutions. Then, an unsupervised clustering algorithm clusters the different trade-off solutions based on their quality metrics. Finally, another clustering algorithm is applied within each cluster of the objective space based on the code locations where the refactorings are recommended to help developers explore the impact of quality attributes while choosing the code fragments to refactor. The input for the second clustering is generated from the first clustering step, hence both algorithms are hierarchical. In other words, the developer can interact with our tool by exploring both the decision and objective spaces to identify relevant refactorings based on their preferences quickly. Thus, the developers can focus on their regions of interest in both the objective and decision spaces. The developers are, in general, first concerned about improving specific quality attributes then they will look for the refactorings that best target the files related to their current interests and ownership [20], [37]. Therefore, we followed this pattern in our approach by clustering first the objective space then we showed the developers the distribution of the refactorings into different decision space clusters for their preferred objective space cluster.

Our approach takes advantage of multi-objective search, clustering, and interactive computational intelligence. Multi-objective algorithms are powerful in terms of diversifying solutions and finding trade-offs between many objectives but generate many solutions. The clustering and interactive algorithms are useful in terms of extracting developers' knowledge and preferences. Existing interactive search-based software refactoring techniques are mainly limited to objective space exploration without considering the decision space.

To evaluate our approach, we selected active developers to manually evaluate the effectiveness of our tool on 6 open source projects and one industrial system. Our results show that the participants found their desired refactorings faster and more accurately than the current state of the art of refactoring tools. This confirms our hypothesis that the second level of clustering (decision space) can help developers to quickly find relevant refactorings based on their preferences in terms of both quality objectives to improve and the location of these changes. A video demo of our interactive refactoring tool can be found at [38].

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

1) To the best of our knowledge, the paper introduces one of the first search-based software engineering techniques that enables the interactive exploration of the objective and decision spaces while existing work focus only on either the objective space or the decision space and they often lack user interaction in the decision space. Our approach is not about a simple filtering of the refactorings based on the locations/files or a clustering of the Pareto front based on the locations. We enabled programmers to interactively navigate between both objective and decision spaces to understand how the refactorings are distributed if they are interested to improve specific quality objectives. Then, our approach can generate even more relevant suggestions after extracting that knowledge from the exploration of the Pareto front.
2) Our contribution is beyond the adoption of an existing metaheuristic technique to refactoring. The proposed approach includes a novel algorithm to enable the exploration of both decision and objective spaces by combining two level of clustering algorithms with multi-objective search.
3) We implemented and validated our framework on a variety of open source and industrial projects. The results support the hypothesis that the combination of both the objective and decision spaces significantly improved the refactoring recommendations.

The remainder of this paper is structured as follows. Section 2 presents the relevant background details. Section 3 describes our approach, while the results obtained from our experiments are presented and discussed in Section 4. Threats to validity are discussed in Section 5. Section 6 provides an account of related work. Finally, in Section 7, we summarize our conclusions and present some ideas for future work.

2 Interactive Refactoring Challenges

Refactoring is a human activity that is hard to automate due to its subjective nature and the high dependency on context.

While successful tools for refactoring have been created, several challenges are still to be addressed to expand the adoption of refactoring tools in practice. To investigate the challenges associated with current refactoring tools, we conducted a survey, as part of an NSF I-Corps project, with 127 professional developers at 38 medium and large companies including eBay, Amazon, Google, IBM, and others [22], [23]. All these developers had a minimum of 11 years of experience in software maintenance tasks and especially refactoring. 112 face-to-face meetings were conducted based on semi-structured interviews to understand the challenges that developers are facing with existing refactoring tools.

From these interviews and our extensive industry collaboration, we learned that architects usually have a desired design in mind as a refactoring target, and developers need to conduct a series of low-level refactorings to achieve this target. Without guiding developers, such refactoring tasks can be demanding: it took one software company several weeks to refactor the architecture of a medium-size project (40K LOC) [23], [39]. Several books [1], [2], [40] on refactoring legacy code and workshops on technical debt [41] present the substantial costs and risks of large-scale refactorings. For example, Tokuda and Batory [42] proposed different case studies with over 800 applied refactorings, estimated to take more than 2 weeks.

There are two major strategies for refactoring in practice: (a) root-canal refactoring and (b) incremental refactoring. The root-canal refactoring is when project owners decide to heavily refactor their system, since some major issues were observed such as the inability to add new features without introducing clones. While root-canal refactoring is less frequent than incremental refactoring, it is still very important in practice. It is currently a major challenge in the software industry, especially with legacy systems such as the ones that we observed at Ford, eBay and so on.

The majority of the interviewees emphasized that root-canal refactoring to restructure the whole system is rare and they are mainly interested in refactoring files that they own rather than files owned by their peers. Refactoring is a complex problem and there are many reasons for why developers may adopt recommended refactorings, among them ownership and code metrics. Note that ownership does not mean a lot in the context of root-canal refactoring (unlike incremental refactoring) since developers may refactor code even though they do not own it. Most existing refactoring tools do not offer a capability of integrating developers' preferences, in terms of which files they may want to refactor, and purely rely on potential quality improvements. Fully automated refactoring usually do not lead to the desired architecture, and a designer's feedback should be considered. Moreover, prior work [43] shows that even some semi-automated tools are underutilized by developers. Over 77% of our interviewees reported that the refactorings they perform do not match the capabilities of low-level transformations supported by existing tools, and 86% of developers confirmed that they need better design guidance during refactoring: *"We need better solutions of refactoring tasks that can reduce the current time-consuming manual work. Automated tools provide refactoring solutions that are hard and costly to repair because they did not consider our design needs."*

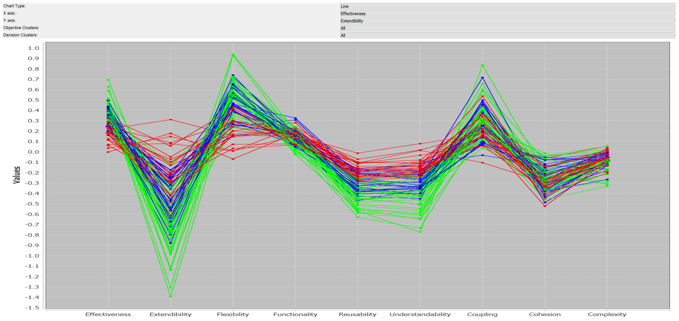


Chart Type: Line
X axis: Effectiveness
Y axis: Extendibility
Objective Clusters: All
Decision Clusters: All

Fig. 1. The output of a multi-objective refactoring tool [22] finding trade-offs between QMOOD quality attributes on ganttproject v1.10.2 with clustering only in the objective space.

Based on our previous experience on licensing refactoring research prototypes to industry, developers always have difficulties and concerns about expressing their preferences up-front as an input to guide refactoring suggestions. They prefer to get insights from some generated refactoring solutions then decide which quality attributes they want to improve. For instance, the number of code smells that are detected for systems is in the hundreds and we have seen reluctance about up-front selection of code smells for refactoring since it is hard for developers to understand the benefits of fixing these smells. Even worse, developer's preferences are not limited to just the quality metrics and their improvements but also where these refactorings will be applied. Our goal is to reduce the need for these up-front developer preferences since they are hard to define in practice by integrating the user's feedback within the different components of multi-objective algorithm for its next run, as described in section 3.5. If the developers are clear about their preferences up-front then they can adjust the fitness functions to target them. Many existing refactoring tools fail to consider the developer perspective, and the developer has no opportunity to provide feedback on the refactoring solution being recommended. 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. Step-wise approaches, unlike the fully automated ones, involve the developers in the loop so they can accept and reject refactoring solutions and express their preferences, thus they have more flexibility in choosing the final set of refactoring to be applied to the system. Determining which quality attribute should be improved, and how, is never a purely technical problem in practice. Instead, high-level refactoring decisions have to take into account the trade-offs between code quality, available resources, project schedule, time-to-market, and management support.

Based on our survey, it is challenging to aggregate quality objectives into one evaluation function to find good refactoring solutions since developers are not able, in general, to express their preferences upfront. While recent ad-

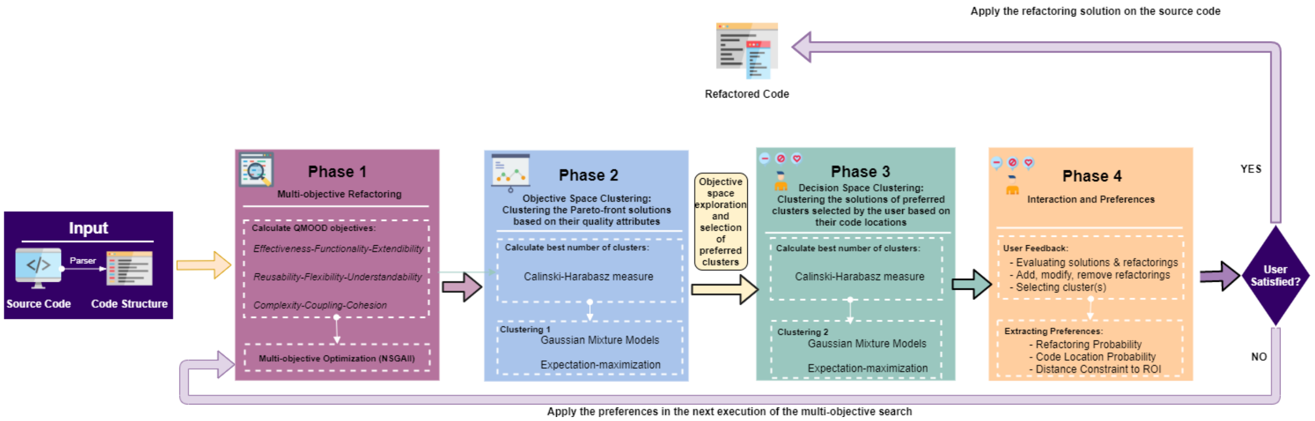

Apply the refactoring solution on the source code
Refactored Code
Input
Parser
Source Code
Code Structure
Phase 1
Multi-objective Refactoring
Calculate QMOOD objectives:
Effectiveness-Functionality-Extendibility
Reusability-Flexibility-Understandability
Complexity-Coupling-Cohesion
Multi-objective Optimization (NSGAII)
Phase 2
Objective Space Clustering: Clustering the Pareto-front solutions based on their quality attributes
Calculate best number of clusters:
Calinski-Harabasz measure
Clustering 1
Gaussian Mixture Models
Expectation-maximization
Objective space exploration and selection of preferred clusters
Phase 3
Decision Space Clustering: Clustering the solutions of preferred clusters selected by the user based on their code locations
Calculate best number of clusters:
Calinski-Harabasz measure
Clustering 2
Gaussian Mixture Models
Expectation-maximization
Phase 4
Interaction and Preferences
User Feedback:
- Evaluating solutions & refactorings
- Add, modify, remove refactorings
- Selecting cluster(s)
Extracting Preferences:
- Refactoring Probability
- Code Location Probability
- Distance Constraint to ROI
YES
User Satisfied?
NO
Apply the preferences in the next execution of the multi-objective search

Fig. 2. Overview of our proposed approach: DOIMR

vances on refactoring proposed tools support multiple preferences of developers based on multi-objective search, these tools still require the user to navigate through many solutions. Figure 1 shows an example of a Pareto front of non-dominated refactoring solutions improving the QMOOD [26] quality attributes of a Gantt Project generated using an existing tool [22]. QMOOD is a widely accepted software quality model, based on our collaborations with industry and existing studies [22], [23], [25], [30], [44], [45], [46]. While developers were interested in giving feedback for some refactoring solutions, they still find the interaction process time-consuming. Even when refactoring solutions are clustered based on the quality objectives, as shown in Figure 1, the number of solutions to be checked by developers can be substantial. Thus, they want to know how different the solutions are within the same objective space. It may be possible to find more than one refactoring solution that offers the same level of quality improvements but by refactoring different code locations/files. In fact, the objective space clustering is important for developers to understand which refactorings could help them to achieve their goals of improving specific quality attributes. However, each cluster will still include a considerable number of solutions since each solution contains a good number of refactorings. Thus, the objective space clustering is necessary and the decision space clustering is complementary to the first phase. Existing refactoring techniques do not, however, enable developer interaction based on both the decision space and objective space; that is the main challenge of this paper. For instance, the objective space exploration can help developers focusing on their targeted design quality improvements then the decision space can help them to focus on files they are owning or related to their current tasks or interests.

3 Approach Description

Figure 2 describes our proposed approach which is composed of four major steps. In the first step, a multi-objective search algorithm is executed to find a set of non-dominated solutions between different conflicting quality objectives of QMOOD [47]. Then, the second step clusters these solutions based on these quality attributes. We call this procedure ”objective space clustering”. The third step takes, as input, every cluster identified from the user’s choice in the objective space and execute another unsupervised learning algorithm to cluster the solutions based on their code locations. Hence, we call this ”decision space clustering”. Finally, developers can interactively choose among the clustered solutions to find a compromise that suits their preferences in both the decision and objective spaces. For instance, developers may select a cluster (from the objective space clustering) that corresponds to their quality improvement preferences. Then, the second clustering will show them how the solutions in the preferred objective space cluster are different in the decision space. For example, the user can easily avoid looking at many solutions that are similar in the decision space (modifying almost the same code locations) based on the second clustering. Note that our algorithm is hierarchical, thus the input of the second clustering algorithm (decision space) is the set of clusters generated by the first clustering algorithm (objective space) that are selected as preferred ones by the user. The multi-objective search algorithm runs for a number of iterations to finally generate refactoring solutions to the user. If the developer is not satisfied with the solutions that are recommended from these iterations, s/he can explore the clustering results and express their preferences and needs; then another run of the multi-objective algorithm will take place for a number of iterations taking into consideration the developer’s preferences (more details are presented in the next sections). This process is iterative until the user is satisfied with a final set of refactoring solutions that is aligned with his preferences.

The next sections will explain in further detail the steps of our methodology.

3.1 Phase 1: Multi-Objective Refactoring

The search for a refactoring solution requires the exploration of a large search space to find trade-offs between 6 different quality objectives. The multi-objective optimization problem can be formulated mathematically in this manner:

$$\begin{aligned} &Minimize && F(x) = (f_1(x), f_2(x), ..., f_M(x)), \\ &Subject\ to && x \in S, \\ & && S = \{x \in R^m : h(x) = 0, g(x) \geq 0\}; \end{aligned}$$

where S is the set of inequality and equality constraints, g and h are real valued functions defined on S, x is an N vector of decision variables, and the functions f_i are *objective* or *fitness* functions. In multi-objective optimization, the quality of an optimal solution is determined by dominance. The set of feasible solutions that are not dominated with respect to each other is called *Pareto-optimal* or *Non-dominated* set.

In the following subsections, we briefly summarize the adaptation of multi-objective search to the software refactoring problem.

3.1.1 Solution Representation

We encode a refactoring solution as an ordered vector of multiple refactoring operations. Each operation is defined by an action (e.g., move method, extract class, etc.) and its specific controlling parameters (e.g., source and target classes, attributes, methods, etc.) as described in Table 3. We considered a set of the most important and widely used refactorings in our experiments: Extract Class/SubClass/SuperClass/Method, Move Method/Field, PullUp Field/Method, PushDown Field/Method, Encapsulate Field and Increase/Decrease Field/Method Security. We selected these refactoring operations because they have the most impact on QMOOD quality attributes [48]. During the process of population initialization or a mutation operation of the algorithm, the refactoring operation and its parameters are formed randomly. Due to the random nature of this process, it is crucial to evaluate the feasibility of a solution meaning to preserve the software behavior without breaking it. This evaluation is based on a set of specific pre- and post-conditions for each refactoring operation as described in [49]. Figure 3 shows an example of a concrete refactoring solution proposed by our approach for GanttProject v1.10.2, including several refactorings applied to different code locations.

3.1.2 Fitness Functions

We used the Quality Model for Object-Oriented Design (QMOOD) [32] as a means of estimating the effect of a refactoring operation on the quality of the software. This model is developed based on the international standard for software product quality measurement and is widely used in the industry. QMOOD is a comprehensive way to assess software quality and includes four levels. Using the first two levels—*Object-oriented Design Properties* and *Design Quality Attributes*—as fitness functions, we formulated the problem as discovering refactorings to improve the design quality of a software system. The fitness functions we calculate are Understandability, Functionality, Reusability, Effectiveness Flexibility, Extendibility, Complexity, Cohesion, and Coupling. We measured the relative change of these quality attributes after applying a refactoring solution as follows:

$$FitnessFunction_i = \frac{Q_i^{after} - Q_i^{before}}{Q_i^{before}} \tag{1}$$

where Q_i^{before} and Q_i^{after} are the value of the quality metric i before and after applying a refactoring solution, respectively.

Table 1 and 2 describe the QMOOD metrics and their computation formulas used in our optimization approach.

TABLE 1
QMOOD metrics and their computation formulas.

QMOOD Metrics	Definition / Computation
Reusability	$-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$
Flexibility	$0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism$
Understandability	$-0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * DesignSize$
Functionality	$0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendibility	$0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$
Effectiveness	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$

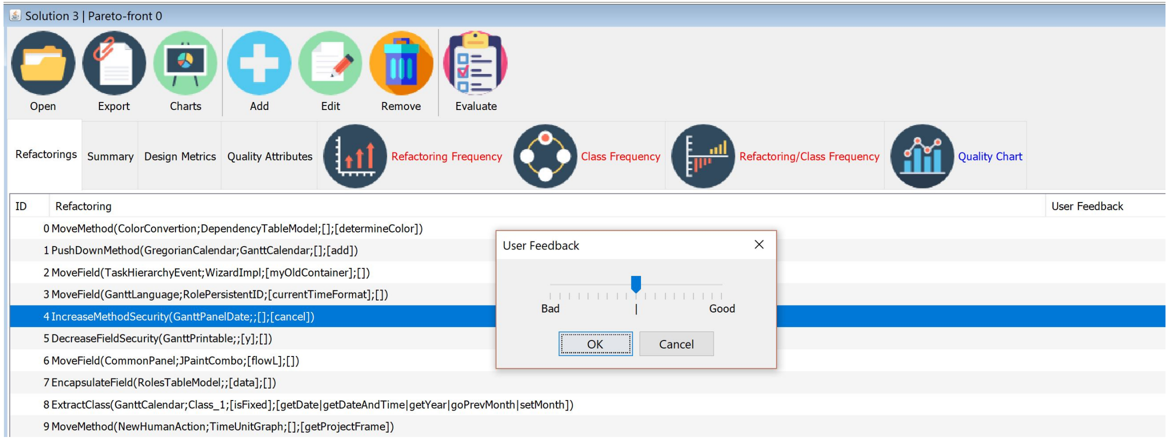
TABLE 2
Design metrics description.

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.

3.2 Phase 2: Objective Space Clustering

One of the most challenging and tedious tasks for a user during any multi-objective optimization process is decision making. Since many Pareto-optimal solutions are offered, it is up to the user to select among them, which requires exploration and evaluation of the Pareto-front solutions.

The goal of this step is to cluster and categorize solutions based on their similarity in the objective space. These clusters of solutions help give the user an overview of the options. Therefore, this technique gives the users more explicit initial exploration steps where they can initiate the interaction by evaluating each cluster center or representative member. Based on our previous refactoring collaborations with industry, developers are always highlighting the time-consuming and confusing process to deal with the large population of Pareto-front solutions: ”where should



ID	Refactoring	User Feedback
0	MoveMethod(ColorConvertion;DependencyTableModel;[];[determineColor])	
1	PushDownMethod(GregorianCalendar;GanttCalendar;[];[add])	
2	MoveField(TaskHierarchyEvent;WizardImpl;[myOldContainer];[])	
3	MoveField(GanttLanguage;RolePersistentID;[currentTimeFormat];[])	
4	IncreaseMethodSecurity(GanttPanelDate;;[];[cancel])	
5	DecreaseFieldSecurity(GanttPrintable;;[y];[])	
6	MoveField(CommonPanel;JPaintCombo;[flowL];[])	
7	EncapsulateField(RolesTableModel;;[data];[])	
8	ExtractClass(GanttCalendar;Class_1;[isFixed];[getDate|getDateAndTime|getYear|goPrevMonth|setMonth])	
9	MoveMethod(NewHumanAction;TimeUnitGraph;[];[getProjectFrame])	

Fig. 3. Example of a refactoring solution proposed by our tool for GanttProject v1.10.2.

I start to find my preferred solution?". This observation is valid for many Search-based software engineering (SBSE) applications using multi-objective search [23].

Clustering is an unsupervised learning method to discover meaningful underlying structures and patterns among a set of unlabelled data. It puts the data into groups where the similarity of the data points within each group is maximized while minimizing the similarity between groups.

Determining the optimal number of clusters is a fundamental issue in clustering techniques. One method to overcome this issue is to optimize a criterion where we try to minimize or maximize a measure for the different number of clusters formed on the data set. For this purpose, we used the Calinski Harabasz (CH) Index, which is an internal clustering validation measure based on two criteria: compactness and separation [50]. We selected the CH index due to the small size of the number of solutions to cluster (our data), and it is known to provide quick clustering solutions with acceptable quality for similar problems. CH assesses the clustering outcomes based on the average sum of squares between individual clusters and within clusters. Therefore, we execute the clustering algorithm on the Pareto-front solutions with various numbers of components as input. The CH score is calculated for each execution, and the result with the highest CH score is recognized as the optimal clustering.

After determining the best number of clusters, we employ a probabilistic model-based clustering algorithm called ”Gaussian Mixture Model” (GMM). GMM is a soft-clustering method using a combination of Gaussian distributions with different parameters fitted on the data and more details about this algorithm can be found in [51]. The parameters are the number of distributions, Mean, Co-variance, and Mixing coefficient. The optimal values for these parameters are estimated using the Expectation-Maximization (EM) algorithm [52]. EM trains the variables through a two-step iterative process.

After the convergence of EM, the membership degree of each solution to a fitted Gaussian or cluster is kept for the preference extraction step. Furthermore, to find a representative member of each cluster, we measure the corresponding density for each solution and select the solution with the highest density.

To calculate the probability distribution function of different Gaussian components, we compute the Mahalanobis distance between data points and its estimated mean vector for all clusters. We allow to choose full covariance matrices in order to model each cluster as an ellipsoid with arbitrary orientation and stretch. In practice, using full covariance matrices improves the performance of the GMM.

3.3 Phase 3: Decision Space Clustering

Our approach gives developers the ability to pinpoint their preferences in a different space than the optimization space related to the location of refactorings. In the exploration of the decision space, user preferences are defined for the set of controlling parameters (mainly code elements to be refactored) that each refactoring has (see Table 3). After selecting a preferred objective space cluster, the developer may want to see "the distribution of the solutions within that region of interest". In other words, the clustering in the decision space will show developers the refactoring solutions that improve the quality at the same level (within the same objective space cluster) but targeting different parts of the systems. To do this, we group the solutions by their similarity in the decision space and present them to the developer as depicted in Figure 4 where only two clusters were found in the decision space. In each of these two clusters, the solutions composing it are introducing refactorings into similar locations with comparable impact on the different quality attributes. These solutions in the decision space are clustered based on the refactoring locations and their frequency. In fact, Figure 4 shows the projection on the objective space of the solutions clustered based on the criteria of the decision space (each color is one decision space cluster); a user can click on the preferred solution to see the criteria of the decision space including the code locations. The developer can combine both kinds of information together (impact of the solution on quality and the code locations) to decide which solution to explore further.

To get an optimal grouping of solutions in the decision space of where refactorings are applied, we use a procedure similar to the one used in the objective space with additional pre-processing steps to project the solutions on the decision space. We define a projection operator based on the

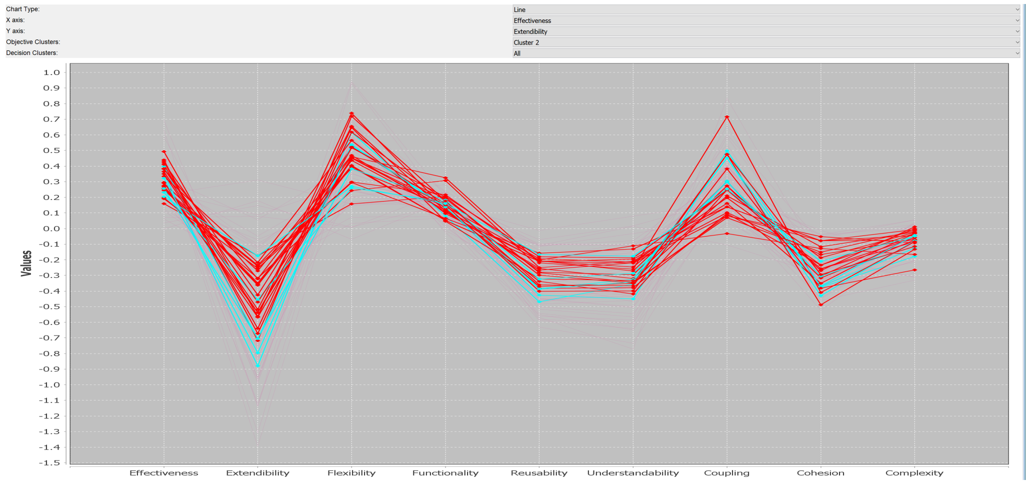


Chart Type: Line
X axis: Effectiveness
Y axis: Extendibility
Objective Clusters: Cluster 2
Decision Clusters: All

Fig. 4. Clustering based on code locations (decision space) of the refactoring solutions of one region of interest in the objective space of GanttProject v1.10.2.

TABLE 3
Refactoring operations with their controlling parameters.

Refactorings	Controlling parameters
Move Method	(sourceClass, targetClass, method)
Move Field	(sourceClass,targetClass,field)
Pull Up Field	(sourceClass,targetClass,field)
Pull Up Method	(sourceClass,targetClass,method)
Push Down Field	(sourceClass,targetClass,field)
Push Down Method	(sourceClass,targetClass,method)
Inline Class	(sourceClass,targetClass)
Inline Method	(sourceClass,sourceMethod,targetClass,targetMethod)
Extract Method	(sourceClass,sourceMethod,targetClass,extractedMethod)
Extract Class	(sourceClass,newClass)

frequency of changes to the classes by the refactorings and their locations (refactored files). Since refactoring operations affect classes differently, where some make changes only at the same class level while others have a source class and a target class, we only count source classes in our work to have a consistent representation for all vectors and to create a new representation for the refactoring vector in the decision space. In this new domain space, the solutions are represented as vectors of integers where the refactored classes are the dimensions of the space, and the values are the number of refactoring operations for that class. The projection operator is used for the entire Pareto-front and enables having two different representations of the same solution set. Note that the number of refactored classes depends on the size of the refactoring solutions. Since we considered the same minimum and maximum size thresholds of refactoring solutions for all executions of the algorithm, the time to generate the clusters is similar even for larger projects since the size is not based on all code elements of the project but just those in the refactoring solutions. A larger set of modified code elements may generate more clusters to explore, which can make the interaction more time-consuming. Additionally, the decision space clustering heavily depends on how many code elements are refactored within each solution. If the majority of the solutions in the Pareto front are refactoring almost the same code elements (for instance, one class) then mainly one big cluster will be generated in the decision space. It is true that a large refactoring solution may have a higher probability to modify larger code elements than a smaller one but it is more accurate to estimate the number of possible clusters in the decision space based on the code elements that are refactored by the solutions in the Pareto front.

The main contribution of our work is enabling the exploration of a diverse set of refactoring solutions within the same objectives space. This amounts to having multiple solutions that are neighbors in the objective space but completely different in the decision space. To do this, we go through all the clusters determined in the previous step and then use the GMM clustering algorithm with the same steps described above to group similar solutions in the decision space. Thus, developers can improve the code toward their preferred objectives while only refactoring the parts of the code that interest them.

Figure 5 shows an example of our approach (DOIMR) where after generating the Pareto-front for the effectiveness and extendibility objectives, the developer can select a cluster in the objective space for further exploration. Then, a developer can explore the clusters and observe that within this cluster, there are three different clusters in the decision space. The region of interest can be highlighted, and the developer can select solutions that correspond to their interest to create further preferences that can be integrated in the optimization process to converge to the desired optimum. For better visualization of the clustered solutions, our tool offers a feature for two-dimensional views.

3.4 Phase 4: Developer Feedback and Preference Extraction

The results of the Bi-Space clustering algorithm are presented to developers in the form of an interactive chart where they can visualize the cluster of their choice in the objective and decision spaces. This presentation helps them get a complete picture of the diversity of the refactoring solutions and the various compromises they may offer. Our goal is to minimize the effort spent by developers to interact with the system and select a final set of refactorings.

Looking at the solutions, developers can evaluate every solution based on their preferences. The granularity offered

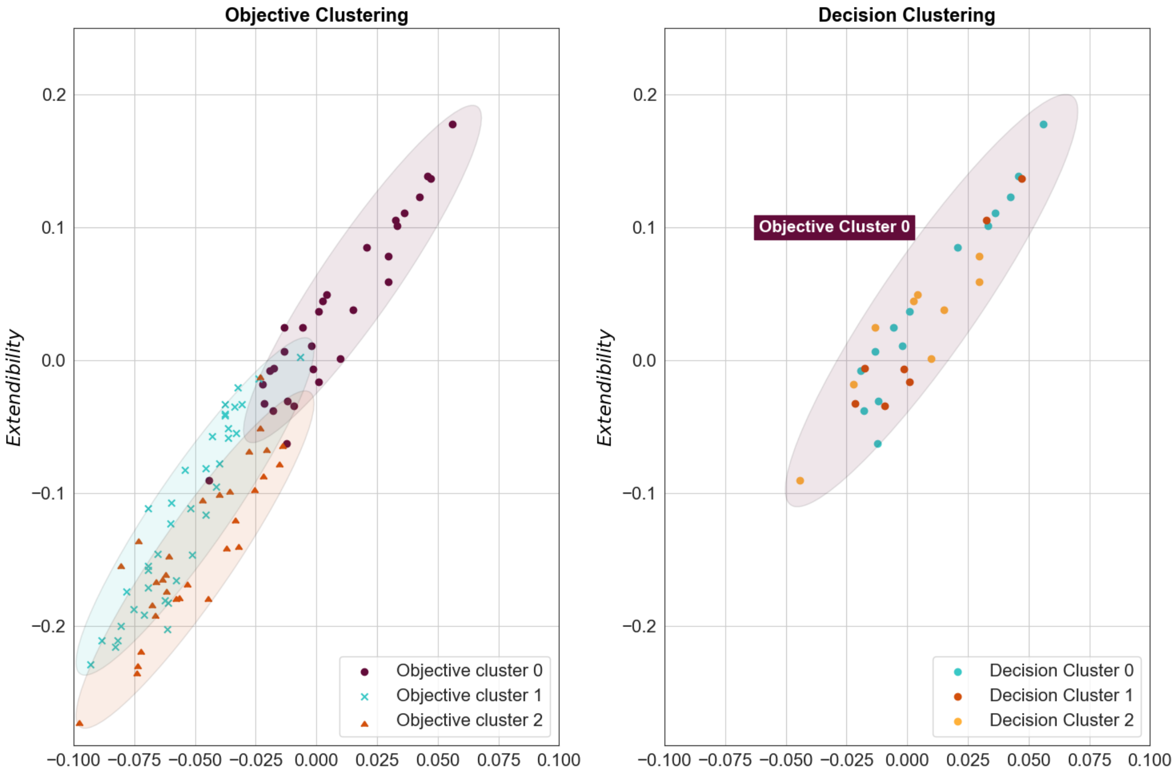


Fig. 5. Illustration of the clustered solutions in the objective space and the decision space

by our representation enables developers to make evaluations at the cluster level (selecting one or more clusters in the objective space), solution level (selecting solutions within a chosen cluster) and refactoring level (choosing to accept, reject, or modifying some refactorings within the chosen solution as shown in Figure 3.). The score obtained reflects developer preferences and serves to determine their region of interest.

At the solution level, the developer is capable of inspecting every refactoring and modifying it. Refactoring operations can be added, deleted, modified, or re-ordered. The information collected afterward is used to calculate a score at the solution level by averaging the scores for every refactoring, and at the cluster levels by averaging the scores of the solutions. The user can reorder the refactorings during the interaction process to fix those that become invalid, due to the violation of pre-conditions, after removing or modifying other refactorings in the sequence. As described in the solution representation section, these conditions are checked when generating new solutions including the application of change operators. It is possible that the order need to be changed again by the user during the new interactions phase with new solutions since the purpose of reordering is not mainly related to the quality improvements or locations but more to keep the refactoring sequence valid if removing/modifying some refactorings require to change the order again.

We calculate the score of a solution and a cluster after the developer interacts with the solutions and provides his feedback in terms of rejecting, accepting, deleting and reordering the solutions. Thus, the scores are extracted from the developer during every interaction independently. If the new population contains some exact same solutions from the previous interaction then the solution already has the score calculated from the previous interaction.

In this way, we can characterize the developer's region of interest as the cluster with the highest score. Information about the preferred classes, refactorings, and quality metrics is extracted and used to create preferences that can be considered in the optimization process. Therefore, the search becomes guided in both the decision and the objective spaces, and we can converge on a developer's preferred solution faster.

For this purpose, we compute the weighted probability of refactoring operations (RWP) and target classes of the source code (CWP) as follow:

$$RWP_p = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|r_p \in s_i|)}{\sum_{r_m \in Ref} \sum_{s_i \in c_j} \gamma_{ij} \times (|r_m \in s_i|)} \tag{2}$$

$$CWP_q = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|cl_q \in s_i|)}{\sum_{cl_m \in Cls} \sum_{s_i \in c_j} \gamma_{ij} \times (|cl_m \in s_i|)} \tag{3}$$

where j is the index of selected cluster, s_i is the solution vector, γ_{ij} is the membership weight of solution i to the cluster j, r is refactoring action, Ref is the set of all refactoring operations, and Cls is the set of all classes in the source code. For every interaction, we compute the probabilities RWP and CWP again without considering previous values because the preferences are already considered when generating the new solutions and we are interested in knowing the developer's feedback about the new solutions thus the new clusters.

3.5 Integrating Developer's Feedback

If the user decides to continue the search process, then the generation and selection of the solutions in the next iteration of the multi-objective search is based on (1) the probability formulas of both refactorings and their locations extracted from the preferred decision and objective clusters which are used in the selection step and change operators; and (2) the initial population of the next iteration of the search algorithm which is seeded from the solutions of the preferred cluster. These are the two key factors to integrate user's preferences. More details about the different components of multi-objective optimization are described in the following:

- *Preference-based initial population:* The solutions from preferred clusters will make up the initial population of next iteration as a means of customized search starting point. In this way, we initiate the search from the region of interest rather than randomly. New solutions need to be generated to fill and achieve the pre-defined population size. Instead of random creation of the refactoring operations (refactoring action and target class) based on a unify probability distribution, we utilize RWP and CWP as a probability distribution. In other words, we copy the solutions from the preferred cluster of the previous round and we randomly create new solutions using the probability distributions to reach the expected population size.
- *Preference-based mutation:* We use a bit flip mutation with mutation probability fixed to 0.4. For every solution that is selected to be mutated, instead of randomly selecting refactoring operations and controlling parameters from equally probabilities distribution, we considered preferred refactoring operations which have higher RWP and CWP. The refactorings with higher RWP are the first candidates to be considered for replacing selected refactorings by the mutation operator and the locations with higher CWP are selected for the controlling parameters to be changed for the selected refactorings.
- *Preference-based selection:* the selection operator tends to filter the population and assign higher chance to the more valuable ones based on their fitness values. In order to consider the user preferences in this process, we adjusted this operator to include closeness to the reference solution as an added measure of being a valuable individual of the population. That means the chance of selection is related to both fitness values and distance to the region of interest as:

$$Chance(s_i) \propto \frac{1}{dist(s_i, CR_j)}, Fitness(s_i) \qquad (4)$$

  where $dist()$ indicates Euclidean distance and CR_j is the representative solution of cluster j. The representative solution is the centroid of the preferred cluster. All the of the six used fitness functions are aggregated in $Fitness(s_i)$ by calculating the average. The selection operator is computed on the final region of interest of the developer which includes the results of both decision and objective space clusterings. Since the two clustering algorithms are hierarchical, the cluster j is the user's preferred decision space cluster.

The above-mentioned customized operators aid to keep the stochastic nature of the optimization process and at the same time take the user preferred refactoring and target code locations (classes) into account.

Our proposed approach will help the developer to understand the diversity of the refactoring solutions when visualizing the clusters thus it will help the user to locate her/his region of interest in both the objective and decision spaces. The goal of the interactions and clustering is to gradually reduce the number of refactoring solutions to be explored by the users based on their preferences. If the developer is still interested to apply more refactorings after selecting the final solution, the tool can be re-executed on the new system after refactoring to find other potential solutions.

4 Evaluation

4.1 Research Questions

We defined three main research questions to measure the correctness, relevance, and benefits of our decision and objective space interactive clustering-based refactoring (DOIMR) tool comparing to existing approaches that are based on interactive clustering-based refactoring only in the objectives space (Alizadeh et al.) [23], interactive multi-objective search (Mkaouer et al.) [22], [30], fully automated multi-objective search (Ouni et al.) [53] and fully automated deterministic tool not based on heuristic search (JDeodorant) [54]. A tool demo of our tool and supplementary appendix materials (questionnaire, setup of the experiments, statistical analyses, and detailed results) can be found in our study's website [1].

The research questions are as follows:

- **RQ1:** Does our approach make more relevant recommendations for developers, as compared to existing refactoring techniques?
- **RQ2:** Does our approach significantly reduce the number of relevant refactoring recommendations and the user interaction effort, as compared to existing interactive refactoring approaches?
- **RQ3: Qualitative Analysis.** To what extent are the user preferences, interaction and identified region of interest similar?

4.2 Experimental Setup

We considered a total of seven systems, summarized in Table 4, to address the above research questions. We selected these seven systems because they are of reasonable size, have been actively developed over the past 10 years, and have been extensively analyzed by the other tools considered in this work. UTest[2] is a project of our industrial partner used for identifying, reporting, and fixing bugs. We selected that system for our experiments since five developers of that system agreed to participate in the experiments, and they

[1] A demo and supplementary appendix materials can be found at the following link: https://sites.google.com/view/tse2020decision

[2] SEMA Inc.

TABLE 4
Statistics of the studied systems.

System	Release	#Classes	KLOC
ArgoUML	v0.3	1358	114
JHotDraw	v7.5.1	585	25
GanttProject	v1.10.2	241	48
UTest	v7.9	357	74
Apache Ant	v1.8.2	1191	112
Azureus	v2.3.0.6	1449	117
JFreeChart	v1.0.9	521	170

are very knowledgeable about refactoring—they are part of the maintenance team. Table 4 provides information about the size of the subject systems (in terms of number of classes and KLOC).

To answer RQ1, we asked a group of 35 participants to manually evaluate the relevance of the refactoring solutions that they selected using four other tools. The first tool of Alizadeh et al. is an approach based on only objective clustering of the Pareto front [23], using the interactive multi-objective search. The second tool is an interactive multi-objective refactoring approach proposed by Mkaouer et al. *et al.* [22], [30], but the interactions were limited to the refactorings (accept/reject) and there is no clustering of the Pareto front or learning mechanisms from the interaction data. Thus, the comparison with these tools will help us to evaluate our main contribution that is built on the top of existing multi-objective refactoring algorithms: the combined use of decision and objective space exploration for interactive refactoring. We have also compared our DOIMR approach to two fully-automated refactoring tools: Ouni *et al.* [53] and JDeodorant [54]. Ouni *et al.* [53] proposed a multi-objective refactoring formulation based on NSGA-II that generates a solution to maximize the design coherence and refactoring reuse from previous releases. JDeodorant [54] is an Eclipse plugin to detect bad smells and apply refactorings. As JDeodorant supports a lower number of refactoring types with respect to the ones considered by our tool, we restrict our comparison with it to those refactorings. We used these two tools to evaluate the relative benefits of our interactive features in helping developers identifying relevant refactorings.

We preferred not to use measures such as anti-patterns or internal quality indicators as proxies for estimating the relevance of refactorings since the developers' manual evaluation already includes a review of the impact of suggested changes on the quality. Furthermore, not all the refactorings that improve quality attributes are relevant to the developers, which is one of the main motivations of this work. The only rigorous way to evaluate the relevance of our tool is the manual evaluation of the results by active developers. This manual evaluation score, MC, consists of the number of relevant refactorings identified by the developers over the total number of refactorings in the selected solution. Due to the subjective nature of refactoring and the large size of considered systems, it is almost impossible to estimate the recall. There is no unique solution to refactor a code/design; thus, it is challenging to construct a gold-standard for large-systems, which makes calculating the recall very challenging.

Participants were first asked to fill out a pre-study questionnaire containing six questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software refactoring. The list of questions of all the questionnaires and the obtained results can be found in the online appendix. Although the vast majority of participants were already familiar with refactoring as part of their jobs and graduate studies, all the participants attended a two-hour lecture on refactoring by the organizers of the experiments. The details of the selected participants can be found in Table 5, including their programming experience in years, familiarity with refactoring, etc. These participants were recruited based on our networks and previous collaborations with 4 industrial partners. They all had a minimum of 6 years experience post-graduation and were working as active programmers with strong backgrounds in refactoring, Java, and software quality metrics.

Each participant was asked to assess the meaningfulness of the refactorings recommended after using the five tools on distinct 5 systems (one tool per system), to avoid a training threat to validity. In this case, none of the participants get more familiar with a specific system or a tool during the validation. We have also randomized the order of evaluated tools between the participants to ensure a fair comparison. The participants not only evaluated the suggested refactorings but were asked to configure, run, and interact with the tools on the different systems. The only exceptions were related to the five participants from the industrial partner, where they agreed to evaluate only their industrial software. We assigned tasks to the participants according to the studied systems, the techniques to be tested and developers' experience. Each of the five tools has been evaluated 5 times on each of the seven systems. Thus, the total number of manual evaluations is 175 among all the 7 projects and 5 tools. Our aim is to find a trade-off between the statistical power and reducing the training and fatigue threats. Thus, we asked each participant to evaluate 5 distinct tools on 5 different projects to avoid that their performances will be impacted by the training effect of the system or/and refactoring tool.

To answer RQ2, we measured the time (*T*) that developers spent to identify the best refactoring strategies based on their preferences and the number of refactorings (*NR*). Furthermore, we evaluated the number of interactions (*NI*) required on the Pareto front for all interactive refactoring approaches. This evaluation will help to understand if we efficiently reduced the interaction effort. For this research question, we decided to limit the comparison to only the interactive multi-objective work of Mkaouer et al. [22], [30] and Alizadeh et al. [23] since they are the only ones offering interaction with the users, and it will help us understand the real impact of the decision space exploration (not supported by existing studies) on the refactoring recommendations and interaction effort. However, for the execution time, we compared our tool with non-interactive approaches as well.

TO answer RQ3, our experiments involved the 35 participants where each of the 7 projects is evaluated using the 5 tools however only two of these tools can generate regions of interests (clusters). Thus, we evaluated if the participants selected the same regions of interests on the 7 projects using the two clustering-based interactive tools. We considered

TABLE 5
Selected Participants.

System	#Subjects	Prog. Exp. (Years)[Avg-Min-Max]	Avg. Refactoring Exp.
ArgoUML	5	[7.5 - 6 - 8.5]	Very High
JHotDraw	5	[8 - 6.5 - 9]	Very High
Azureus	5	[9.5 - 7.5 - 11.5]	High
GanttProject	5	[7 - 6 - 8.5]	High
UTest	5	[15.5 - 13 - 19.5]	Very High
Apache Ant	5	[9 - 6 - 12.5]	Very High
JFreeChart	5	[7 - 6 - 9.5]	Very High

two regions of interests are similar/overlapping if the coordinates of their centroid is almost same by calculating the euclidean distance. We have also evaluated the frequency of common refactorings among the selected final solutions by the users to identify any common patterns.

4.3 Parameter Setting

It is well known that many parameters compose computational search and machine learning algorithms. Parameter setting is one of the longest standing grand challenges of the field. We have used one of the most efficient and popular approaches for parameter setting of evolutionary algorithms, which is Design of Experiments (DoE). Each parameter has been uniformly discretized in some intervals. Values from each interval have been tested for our application. Finally, we pick the best values for all parameters. Hence, a reasonable set of parameter's values have been experimented. This process is done for each of the studied algorithms while the interactive module is disabled.

The stopping criterion was set to 100,000 evaluations for all optimization and search algorithms to ensure fairness of comparison (without counting the number of interactions since it is part of the users' decision to reach the best solution based on their preferences).

The parameters of the multi-objective algorithm are as follows: Single point crossover probability = 0.7; Bit flip mutation probability = 0.4, where the probability of gene modification is 0.5 and stopping criterion was set to 100,000 evaluations. We also set the initial population size to 100 and utilized Binary selection operator. The minimum and maximum length of solution vectors are limited to 10 and 30, respectively.

Furthermore, we used the maximum number of iterations = 1000 and convergence threshold = 0.0001 for the GMM clustering phase. We calculated these parameters using the same DoE approach in a way to make sure that log likelihood function is converged for all studied systems. For instance we picked the minimum number of iterations that guarantees the convergence of clustering algorithm for all systems.

4.4 Results

Results for RQ1. Figure 6 summarizes the manual validation results of our DOIMR approach compared to the state of the art, as evaluated by the participants. It is clear from the results that interactive approaches generated much more relevant refactorings, as compared with the automated tools of Ouni et al. and JDeodorant. Among the interactive approaches, DOIMR outperformed the other interactive approaches of Mkaouer et al. and Alizadeh et al. which supports the idea that information that the developer used from the decision space, such the code locations where refactorings were applied and the refactorings frequency, was helpful. On average, for all of our seven studied projects, 91% of the proposed refactoring operations were considered to be useful by the subjects. The remaining approaches have an average of 83%, 71%, 67%, and 56% respectively for Alizadeh et al. (interactive with objective space clustering), Mkaouer et al. (interactive multi-objective approach), Ouni et al. (fully automated multi-objective approach) and JDeodorant (deterministic non-search-based approach). The highest MC score is 100% for the Azureus and Gantt projects, and the lowest score is 91% for the industrial system UTest. This lowest score can be explained by the fact that the participants are very knowledgeable about the evaluated system. The participants were not guided on how to interact with the systems, and they mainly looked at the source code to understand the impact of recommended refactorings.

We found that automated refactorings generate a lot of false positives. Both the Ouni et al. and JDeodorant tools recommended a large number of refactorings compared to the interactive tools, and many of them are not interesting for the context of the developers, and so the developers reject these refactorings, even though they may be correct. For instance, the developers of the industrial partner rejected several recommendations from these automated tools simply because they were related to stable code or code fragments outside of their interests. The majority of them will not change code out of their ownership as well. Furthermore, they were not interested to blindly change anything in the code just to improve quality attributes. Compared to the remaining interactive approaches, we found that some of the refactoring solutions of DOIMR will never be proposed by Mkaouer et al. or Alizadeh et al. since they are selected because of their extensive refactoring on specific code fragments that developers may found essential to improve their quality based on the features included in these classes. In fact, one of the main challenges of multi-objective search is the noise introduced by sacrificing some objectives and trying to diversify the solutions. Thus, the decision space exploration can help the developers know the most diverse refactoring solutions among one preferred cluster in the objective space. Thus, developers did not waste time on evaluating refactoring solutions that are similar but related to entirely different code files.

To better investigate the comparison of our approach to the closest work of Alizadeh et al. based only on the objectives space exploration, we qualitatively evaluated the role of the decision space exploration to increase the relevance of refactoring recommendations. Based on the participants feedback during the post study interviews, 26 of the interviewees highlighted that the final step of the decision space exploration helped them to understand differences between the refactoring solutions targeting their goals such as improving specific quality attributes. It was not practical for them to check all the solutions of the preferred objective space cluster. Thus, the decision space highlighted the solutions that are truly different (modifying different code locations) but still achieving the same levels of quality

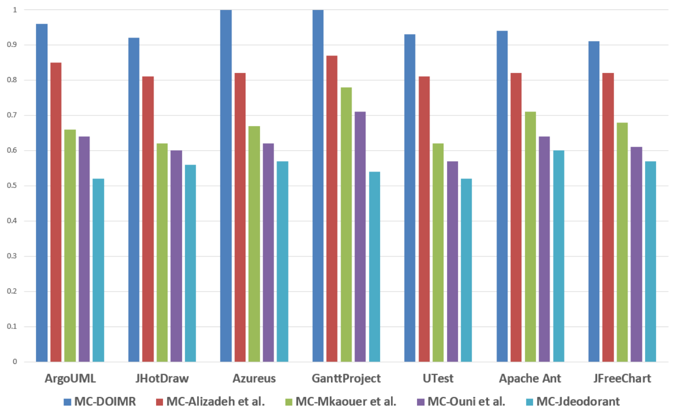


Fig. 6. Median manual evaluations, MC, on the 7 systems.

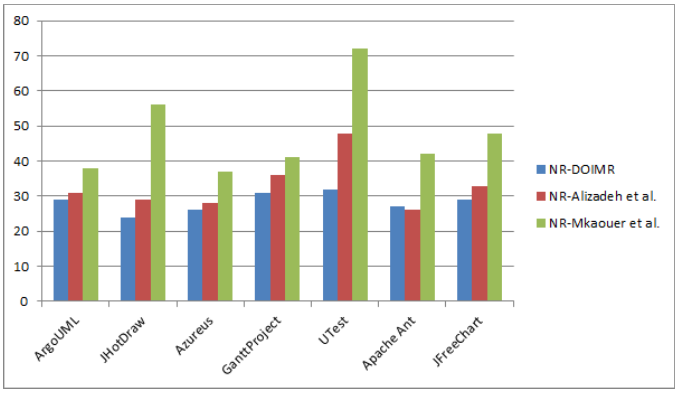


Fig. 7. The median number of recommended refactorings, NR, of the selected solution on the 7 systems.

improvements. For instance, some developers preferred solutions that modified a minimum number of code locations but still reached the same level of quality improvements. Others preferred solutions that modified the files that they owned. Still other developers found the refactorings addressing diverse code locations, including long refactorings sequences, are best since they want to make major changes independently of the cost. And other developers selected solutions that can be associated with recent pull-requests or those under review. Thus, the main advantage of the decision space clustering is to help the users understand, with low effort, which refactoring strategy may help them achieve their goal based on their context. For most cases, it was sufficient to look at the center of the clusters to understand the differences between solutions that can target the same objectives.

To conclude, our DOIMR approach outperformed the four other refactoring approaches in terms of recommending relevant refactoring solutions for developers (RQ1).

Results for RQ2. Figures 7, 8, and 9 give an overview of the number of refactorings for the selected solution, number of required interactions, and the time, in minutes, using our tool, the interactive clustering approach of Alizadeh et al., and the interactive multi-objective approach of Mkaouer et al. However, for the execution time, we compared our tool with non-interactive approaches as well. Based on the results of Figure 7, it is transparent that our approach significantly reduced the number of recommended refactorings compared to the other interactive approaches while increasing the manual correctness as described in RQ1. The highest number of refactorings was observed on the industrial system with 32 refactorings using DOIMR, 48 using Alizadeh et al. and 72 refactorings using Mkaouer et al. This result may be explained by the size and the quality of this system along with the fact that it was evaluated by some of the original developers of UTest. The lower number of recommended refactorings using DOIMR, compared to the other interactive approaches, is related to the elimination of the noise in multi-objective search not only in terms of objectives but the relevant code locations to be refactored (decision space). It is normal to see fewer refactorings when the search space is reduced to a smaller number of files, which was the case of DOIMR.

Figure 8 shows that DOIMR required far fewer developer interactions than the other interactive approaches. For instance, only 13 interactions were required to modify, reject and select refactorings on Azureus using our approach, while 23 and 38 interactions respectively were needed for Alizadeh et al. and Mkaouer et al. The reduction of the number of interactions is mainly due to the smaller number of solutions to explore, after the selection of a preferred cluster in both the objective and decision spaces.

The participants also spent less time to find the most relevant refactorings on the various systems compared to the other interactive and non-interactive approaches, as described in Figure 9. The execution time of our approach includes the execution of the multi-objective search, both clusterings, and the different phases of interaction until the developer is satisfied with a specific solution. The execution time of Alizadeh et al. included all the steps of multi-objective search, the objective space clustering, and the interactions while Mkaouer et al. included the multi-objective search and the user interactions. Thus, it is natural that the main differences in the execution time can be observed in the interaction effort. The average time of our approach is reduced by over 40 minutes (70%) compared to Mkaouer et al. for the case of JHotDraw. The reduction of the execution time is mainly explained by the rapid exploration of fewer solutions after looking mainly to the most diverse (different) solutions in the decision space of the preferred cluster in the objective space. In fact, our DOIMR tool has more components (clustering at both objective and decision spaces) than Alizadeh et al. and Mkaouer et al. but the clustering at both spaces significantly reduced the most time-consuming step (user interactions) since the clusterings, and multi-objective search algorithms are quick and executed in few minutes (between 2 and 4 minutes). The execution time is mainly affected by the developer's interaction effort. The developer's interaction effort is not only affected by the number of recommended refactorings, but it is also affected by the solutions that they need to explore and check manually. The decision space clustering of the preferred cluster from the objective space dramatically reduced the number of solutions to check which resulted in fewer interactions. For instance, a user can easily avoid checking many solutions within the same decision space cluster (modifying similar elements) that have similar impacts on the objectives. We note that the execution times included the interaction with

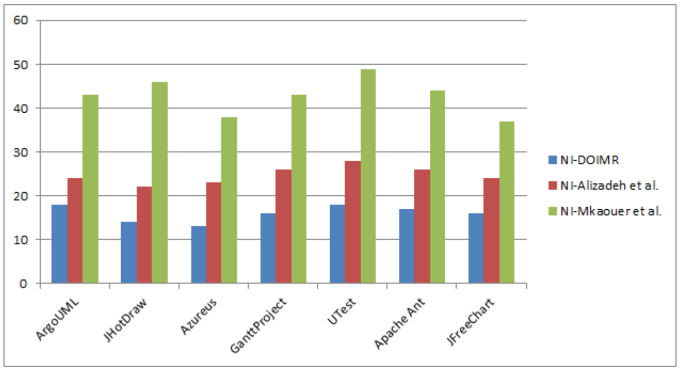


Fig. 8. The median number of required interactions (accept/reject/modify/selection), NI, on the 7 systems.

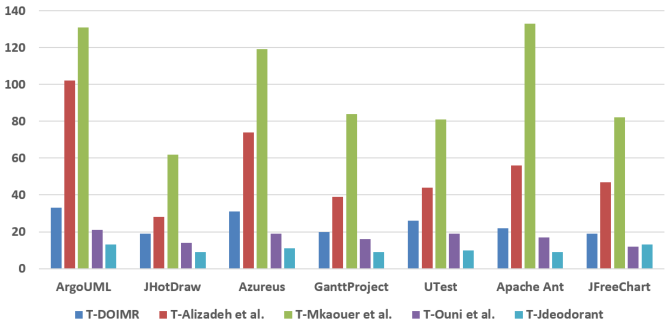


Fig. 9. The median execution time, T, in minutes on the 7 systems.

the user.

Results for RQ3. Our experiments involved 35 participants where each of the 7 projects is evaluated using the 5 tools however only two of these tools can generate regions of interests (clusters). Thus, we evaluated if the participants selected the same regions of interests on the 7 projects using the two clustering-based interactive tools as shown in Figure 10. Note that the minimum number of iterations is 2 for JHotdraw and the maximum is 9 for ArgoUML using our approach where feedback/interactions with the user are recorded. In each of these iterations, the user interacted with the proposed solutions to reject/modify/accept/reorder refactorings. The regions of interests can be only compared in the two tools for the objective space since only our approach generates clusters in the decision space. The overlap measure is calculated based on the number of common clusters that are selected by the participants divided by the total number of selected clusters by the participants. In fact, the overlap measure is the number of the clusters that are selected by multiple users similarity between the clusters for each participant and thus to understand the differences in the developers' preferences. We applied this measure separately on both the decision and objective spaces. The results show that an average of 61% of the selected regions of interests are the same which confirms that the decision space clustering helped developers to select their preferred solution since better refactoring solutions were observed using our approach even when the selected region of interest is the same in the objective space. Another interesting observation is that almost half of the selected region of interest in the objective space are different which means that developers may have different preferences when refactoring systems as explained in the previous comments. We have also checked if multiple participants select the same region of interest in the decision space by looking only at the results of our approach on the 7 projects (5 selected solutions per project using our tool by the participants since each of the 35 participants evaluated 5 distinct tools on 5 different projects). It is interesting to note that the overlap average in the regions of interests at the decision space is higher than the objective space with an average of 71% which can be explained by the fact that the diversity of solutions within a preferred cluster in the objective space is less than the diversity of solutions in the objective space.

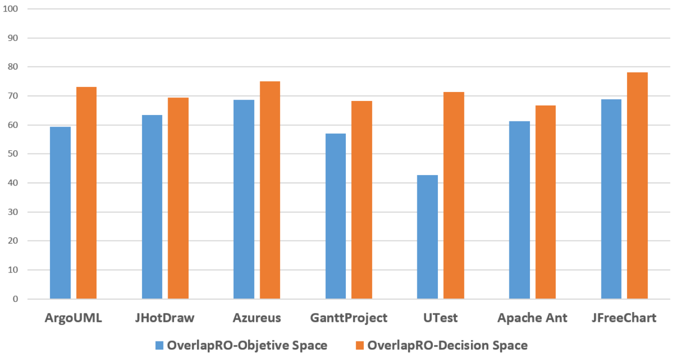


Fig. 10. region of interest at the objective and decision space levels.

Since the execution of the two clustering algorithms is hierarchical, the final results are actually the combination of two clustering steps. We recorded in our tool all the interactions with the user and we found that all participants used both the objective and decision space clusters before selecting a final solution. In the post-study feedback, participants emphasized that both the decision and objective space interactions helped them to find a relevant solution. The common pattern was to establish their goals from refactoring the code and then they used the decision space to find a solution that matched their context (e.g. code reviews, root-canal refactoring, etc.).

To further investigate the preferences of the participants, Figure 11 summarizes the distribution of the refactoring types among the final selected refactoring solutions by the participants. It is clear that the preferred solutions mainly included Extract Class (22%), Move Method (19%) and Extract Method (17%). In fact, the impact of these refactoring types can be positive on many quality attributes such as extendability, reusability, etc.

Since the above results based on the medians are maybe more useful to compare the different interactive approaches, we present and discuss in the following the results per participant for the number of refactorings in the selected solution, number of required interactions and time spent to find a relevant solution. The box plots of Figure 12 shows that the size of the refactoring solutions selected by the participants tend to be similar for our approach (between 25 and 35 refactorings) on the different projects. However, the deviation is high for the approach of Mkaouer et al. but they tend to be larger than the clustering-based approaches which shows the value of using the clusters to better under-

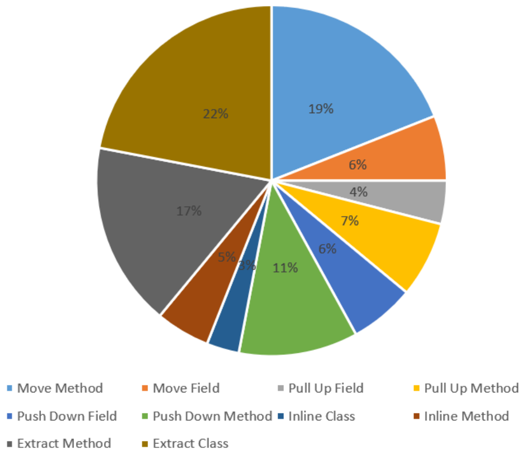


Fig. 11. Refactoring types distribution among the solutions selected by the user

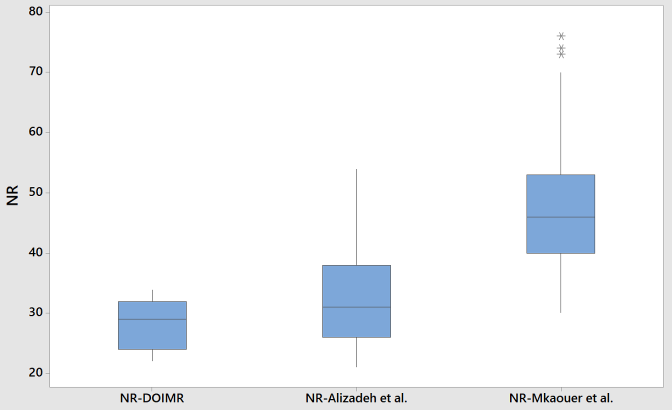


Fig. 12. Distribution of the number of refactorings in the selected solutions for the 35 participants.

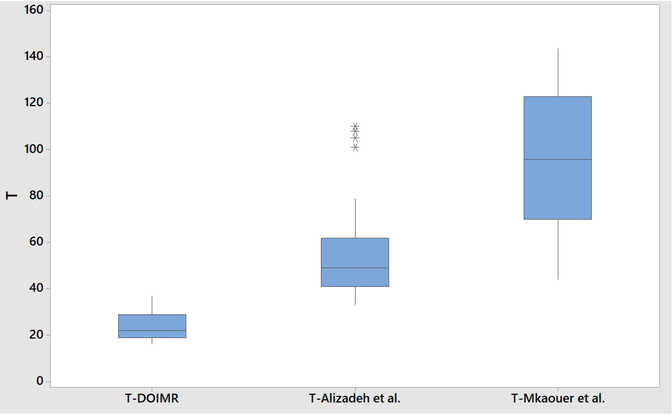


Fig. 13. Distribution of the time spent to find a relevant solution for the 35 participants.

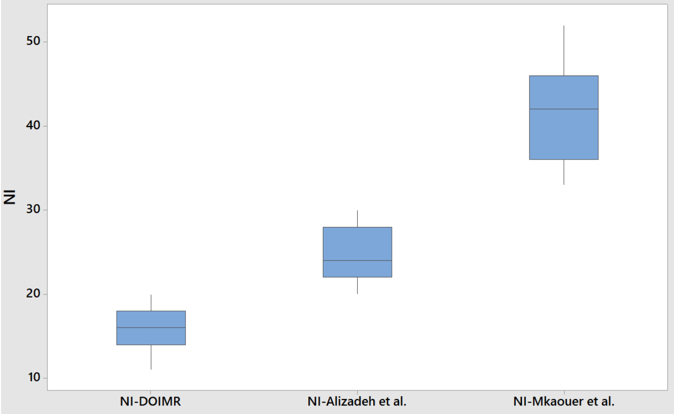


Fig. 14. Distribution of the number of required interactions for the 35 participants.

stand the preferences and guide the search towards relevant refactorings. The same observations apply for the time spent by developers and the number of interactions as described in Figures 13 and 14. In fact, a higher number of interactions will lead to higher time spent by the participants to find relevant refactoring solutions. While the time spent by the participants for using the tool of Mkaouer et al. is diverse, all of them spent more time than our approach for all the projects and participants.

Although the results show the outperformance of interactive approaches compared to automated ones based on different metrics, there are also some limitations related to the use of interactive approaches such as the fatigue despite that our approach significantly reduced the number of iterations with the decision space clustering. It is possible that users can be confused and provide inconsistent feedback which can negatively impact the behavior of the search in the next iterations. The visualization support is also critical to enable relevant feedback from developers to understand the impacts of the recommended refactorings. Another limitation of the interactivity is the difficulty to backtrack some interaction decisions provided to the search algorithm. Finally, the total execution time of interactive approaches is higher than automated ones as described in Figure 9.

Statistical Analysis Since meta-heuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. We utilized statistical analysis to perform a comparison between several metaheuristic approaches in this study and to determine the reliability of the results obtained. The following statistical tests show that all the comparisons performed between our approach and existing ones are statistically significant based on all the metrics considered in our experiments.

We used one-way ANOVA statistical test with a 95% confidence level ($\alpha = 5\%$ to find out whether our sample results of different approached 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 various studied metaheuristic algorithm (independent variable - groups) to investigate and compare each performance metric (dependent variable) on each subject system (software project). We test the null hypothesis (H_0) that population means of each metric is equal for all methods ($\mu_{M1}^{metric} = \mu_{M2}^{metric} = \mu_{M3}^{metric} = \mu_{M4}^{metric} \; where \; metric \in \{T, NI, NR, MC\}$ 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:

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 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 ($p > 0.05$).

We have also checked the assumption of IID data within each group. In fact, the residuals from the model are approximately normal since the values are approximately similar. Intuitively, data values are IID if they are not related to each other and if they have the same probability distribution. Thus, the assumption of IID data is verified.

The results of one-way ANOVA tests 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.

The obtained value of F-statistics for each metric are as follows: $F_T = 99.18$, $F_{NI} = 327.41$, $F_{NR} = 40.96$, and $F_{MC} = 102.84$. 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.

Since one-way ANOVA does not indicate the difference size, we also calculated the "Vargha-Delaney A" measure [55]. This measure clarifies the effect size (strength of association) and it estimates the degree of association between the independent factor and dependent variable for the sample. the A measure is a value between 0 and 1. When it is exactly 0.5, then the two methods achieve equal performance. When A is less than 0.5, the first method is worse, and when A is more than 0.5, the second method is worse. The closer to 0.5, the smaller the difference between the techniques, and the farther from 0.5, the larger the difference.

Table 6 shows the "Vargha-Delaney A" results for different metrics between our method and others on each subject system. Since Ouni et al. (M4) is a fully automated multi-objective search without the interactive component, it is only considered for MC metrics. Table 6 shows that our approach is better than all the other algorithms with an A effect size that is at least higher than 0.81 for all the 7 systems and the 4 considered metrics (T, NI, NR, MC). For instance, considering the execution time T, we find that our approach (M1) has an A effect size that is higher than 0.91 for ArgoUML,GanttProject, Azureus and UTest; and an A effect size higher than 0.83 for JHotDraw, JFreeChart and Apache Ant. This confirms our findings in RQ2 that the clustering at both spaces significantly reduced the execution time. Same observations apply to reduction of the number of recommended refactorings (NR) and number of interactions (NI). Overall, the high A effect size values for all the metrics and the 7 systems show that the DOIMR outperforms the state of the art refactoring techniques and the outperformance is significant.

5 Threats to Validity

Conclusion validity. The parameter tuning used in our experiments creates an internal threat that we need to evaluate in our future work. We have used one of the most efficient and popular approaches for parameter setting of evolutionary algorithms, which is Design of Experiments (DoE). Each parameter has been uniformly discretized in some intervals. Values from each interval have been tested for our application. Finally, we chose the best values for all parameters. Hence, a reasonable set of parameter values have been studied. Another conclusion threat is the number of interactions with the developers since we did not force them to use the same maximum number of interactions which may sometimes explain the out-performance of our approach. Moreover, the developers interacted with the different tools using their offered original graphical interfaces (UIs) which may represent another threat. In fact, developers may perform better with a given tool because it has a better user friendly graphical interface to understand the impact of the refactorings. However, the participants were given the same amount of time to use the tool (limited to three hours).

Internal validity. The variation of correctness and speed between the different groups when using our approach and other tools can be an internal threat since the participants have different levels of experience. To counteract this, we assigned the developers to different groups according to their programming experience to reduce the gap between the groups, and we also adopted a counter-balanced design. Regarding the selected participants, we took precautions to ensure that our participants represented a diverse set of software developers with experience in refactoring, and also that the groups formed had similar average skill sets in terms of refactoring area. To mitigate the training threat, we ensured that the participants (1) did not evaluate the same tool more than one time (even on different projects), (2) did not evaluate the same project more than one time, and (3) we used a random order between the participants for the sequence of tools to be evaluated on different systems. To mitigate the fatigue threat, we allowed participants to perform the experiments in multiple sessions (at least one tool per session).

Construct validity. The developers involved in our experiments may have had divergent opinions about the relevance of the recommended refactorings, which may impact our results. However, some of the participants are the original programmers of the industrial system, which may reduce the impact of this threat. Unlike fixing bugs, refactoring is a subjective process, and there is no unique refactor

TABLE 6
Vargha-Delaney A measure for different metrics between our method(M1) and others.
Label of the methods: **M1** DOIMR (Our approach), **M2**=Alizadeh et al. [23], **M3**=Mkaouer et al. [22], [30], **M4**=Ouni et al. [53]

	T		NI		NR		MC		
Comparison	**M1-M2**	**M1-M3**	**M1-M2**	**M1-M3**	**M1-M2**	**M1-M3**	**M1-M2**	**M1-M3**	**M1-M4**
ArgoUML	0.94	0.89	0.91	0.86	0.93	0.87	0.91	0.86	0.86
JHotDraw	0.88	0.9	0.84	0.89	0.88	0.91	0.88	0.89	0.88
GanttProject	0.91	0.92	0.87	0.82	0.91	0.94	0.85	0.92	0.86
UTest	0.93	0.84	0.83	0.9	0.93	0.88	0.94	0.84	0.9
Apache Ant	0.89	0.88	0.88	0.81	0.86	0.82	0.9	0.86	0.83
Azureus	0.93	0.82	0.9	0.86	0.83	0.93	0.83	0.91	0.81
JFreeChart	0.83	0.91	0.92	0.83	0.86	0.86	0.92	0.94	0.92

solution; thus, it is difficult to construct a gold-standard for large systems which makes calculating recall challenging. Does the deviation from an expected refactoring solution mean that the recommendation is wrong or simply another way to refactor the code?

External validity. The first threat is the limited number of participants and evaluated systems, which threatens the generalizability of our results. Besides, our study was limited to the use of specific refactoring types and quality attributes. Furthermore, we mainly evaluated our approach using classical algorithms such as NSGA-II, but other existing metaheuristics can be used. Future replications of this study are necessary to confirm our findings.

6 Related Work

6.1 Preference-based SBSE

The field of search-based software refactoring has considerably grown in the last years, and an extensive summary of the existing work can be found in [56], [57], [58], [59] which provide a systematic review of the field.

Search-based techniques [60] are widely studied to automate software refactoring to improve the design quality of software projects using a set of software metrics. Several of existing studies combine these metrics into a single fitness function to find the best sequence of refactorings. Seng et al. [61] have proposed a single-objective optimization approach using a genetic algorithm to suggest a list of refactorings to improve software quality.

The majority of existing multi-objective refactoring techniques [25], [35], [53], [62] 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.

The problems of search space reduction and contextualization to developer's regions of interest during refactoring recommendation process have been treated in several papers [63], [64], [65]. Han et al. proposed in [65] an approach to enable the interactions with the user in the objective space then a Delta Table can select quickly the next refactoring to improve a specific objective without calculating a fitness function at each iteration. Morales et al. in [63] proposed an algorithm to remove redundant refactoring solutions that may have the same refactorings with a different order in the sequence but the final design is the same. Finally, the work in [64] presents a filter to the refactoring recommendations based on the recently introduced code changes by developers. The previously two mentioned studies demonstrated the importance of search space reduction. However, there was no interaction with the users.

To the best our knowledge, all existing Search-Based Software refactoring studies are mainly focusing on the objective space to evaluate the solutions and interact with the developers. There are some studies [66], [67] in SBSE field not related to refactoring which provide interactions in the decision space. Ferrira et al. in [66] have proposed an interactive model for the next release problem using ant colony optimization, where the user can define which requirements he/she would like to include or not in the next release. The proposed approach generates solutions that have more than 80% of the developer's preferences and it the obtained results show an improvement compared to a solution with no human intervention. Additionally, Ramírez et al. proposed in [67] an interactive approach to discover software architectures using developer's feedback to guide a multi-objective evolutionary algorithm. The user's feedback is incorporated into the fitness function. The authors reported that the interaction effectively guided the search towards the regions of the search space that are of real interest to the expert.

In summary, these works provide interactions in the decision space by integrating the developer's feedback into the fitness functions and they demonstrate that preference-based SBSE can effectively improve the search results. In this paper, we got inspired by these studies to incorporate the developers preference with some differences. In fact, we are proposing we are proposing a novel multi-objective approach which enables interaction with developers in both decision and objective spaces leading to a better understanding of the diversity of the recommended refactoring solutions and quick identification of the preferred solution. The developer's feedback was not only integrated in the fitness function but also in all the other different components of the evolutionary algorithm. Our results are aligned with the existing studies which show the importance of integrating the developer's preferences to guide the search space.

6.2 Interactive Refactoring

A recent work [68] presents a systematic review of interaction in Search-Based Software Engineering which provides a classification scheme for the existing studies in the interactive SBSE subfield.

Dig [69] proposes an interactive refactoring technique to improve the parallelism of software systems. However, the proposed approach did not consider learning from the developers' feedback and focused on making programs more parallel.

Bavota et al. [31] 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 re-modularizations. Interactive Genetic Algorithms (IGAs) extend the classic Genetic Algorithms (GAs) by partially or entirely involving the user in the determination of the solution's 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.

Some recent studies [22], [23], [24] extended a previous work of [30] 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 tries 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 may be 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. Finally, all existing interactive search-based refactoring studies are still generating a large number of solutions that the developers need to explore.

7 Conclusion

In this paper, we presented a novel way to enable interactive refactoring by combining the exploration of quality improvements (objective space) and refactoring locations (decision space). Our approach helped developers to quickly explore the Pareto front of refactoring solutions that can be generated using multi-objective search. The clustering of the decision space helped the developers identify the most diverse refactoring solutions among ones located within the same cluster in the objective space, improving some desired quality attributes. To evaluate the effectiveness of our tool, we conducted an evaluation with human subjects who evaluated the tool and compared it with the state-of-the-art refactoring techniques. Our evaluation results provide evidence that the insights from both the decision and objective spaces helped developers to quickly express their preferences and converge towards relevant refactorings that met the developers' expectations.

In our future work, we are planning to automatically learn from user interactions for fast convergence to good refactoring solutions. Besides we plan to expand our experiments with more systems and participants.

References

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[2] J. Kerievsky, *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[3] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 179–188.

[4] J. Carriere, R. Kazman, and I. Ozkaya, "A cost-benefit framework for making architectural decisions in a business context," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2. IEEE, 2010, pp. 149–157.

[5] "The developer Coefficient." [Online]. Available: https://stripe.com/reports/developer-coefficient-2018

[6] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *11th working conference on reverse engineering*. IEEE, 2004, pp. 144–151.

[7] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: Ide support for real-time auto-completion of refactorings," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 222–232.

[8] X. Ge and E. Murphy-Hill, "Benefactor: a flexible refactoring tool for eclipse," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2011, pp. 19–20.

[9] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 371–372.

[10] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 350–359.

[11] E. 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. 5–18, 2011.

[12] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *European Conference on Object-Oriented Programming*. Springer, 2006, pp. 404–428.

[13] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.

[14] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 1145–1156.

[15] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[16] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[17] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 331–336.

[18] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 23, 2016.

[19] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *European Conference on Object-Oriented Programming*. Springer, 2006, pp. 404–428.

[20] M. Kessentini, T. J. Dea, and A. Ouni, "A context-based refactoring recommendation approach using simulated annealing: two industrial case studies," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 1303–1310.

[21] Y. Cai and K. Sullivan, "A formal model for automated software modularity and evolvability analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, p. 21, 2012.

[22] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software Engineering*, 2018.

[23] V. Alizadeh and M. Kessentini, "Reducing interactive refactoring effort via clustering-based multi-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 464–474.

[24] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 535–546.

[25] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using nsga-iii," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, p. 17, 2015.

[26] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó. Cinnéide, and K. Deb, "On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2503–2545, 2016.

[27] I. H. Moghadam and M. O. Cinneide, "Automated refactoring using design differencing," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 43–52.

[28] E. 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. 5–18, 2011.

[29] A. Ouni, M. Kessentini, M. Ó Cinnéide, H. Sahraoui, K. Deb, and K. Inoue, "More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells," *Journal of Software: Evolution and Process*, vol. 29, no. 5, p. e1843, 2017.

[30] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 331–336.

[31] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in eclipse: The aries project," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 1419–1422.

[32] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[33] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," *Empirical Software Engineering*, vol. 22, no. 2, pp. 894–927, 2017.

[34] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 345–364, 2008.

[35] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2012, pp. 49–58.

[36] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[37] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential java code for concurrency via concurrent libraries," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 397–407.

[38] "The proposed refactoring tool." [Online]. Available: https://sites.google.com/view/tse2020decision

[39] A. Ghannem, M. Kessentini, and G. El Boussaidi, "Detecting model refactoring opportunities using heuristic search," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, 2011, pp. 175–187.

[40] M. Feathers, *Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1*. Prentice Hall Professional, 2004.

[41] "The Seventh International Workshop on Managing Technical Debt," http://www.sei.cmu.edu/community/td2015/.

[42] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings," *Automated Software Engineering*, vol. 8, no. 1, pp. 89–120, 2001.

[43] E. R. Murphy-Hill and A. P. Black, "Why don't people use refactoring tools?" in *WRT*, 2007, pp. 60–61.

[44] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 488–498.

[45] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 181–190.

[46] L. C. Briand, J. Wust, S. V. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: an industrial case study," in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*. IEEE, 1999, pp. 345–354.

[47] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[48] R. Shatnawi and W. Li, "An empirical assessment of refactoring impact on software quality using a hierarchical quality model," *International Journal of Software Engineering and Its Applications*, vol. 5, no. 4, pp. 127–149, 2011.

[49] W. F. Opdyke, "Refactoring object-oriented frameworks," 1992.

[50] T. Caliński and J. Harabasz, "A dendrite method for cluster analysis," *Communications in Statistics-theory and Methods*, vol. 3, no. 1, pp. 1–27, 1974.

[51] G. Xuan, W. Zhang, and P. Chai, "Em algorithms of gaussian mixture model and hidden markov model," in *Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205)*, vol. 1. IEEE, 2001, pp. 145–148.

[52] R. A. Redner and H. F. Walker, "Mixture densities, maximum likelihood and the em algorithm," *SIAM review*, vol. 26, no. 2, pp. 195–239, 1984.

[53] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 23, 2016.

[54] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1037–1039.

[55] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[56] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Information and Software Technology*, vol. 83, pp. 14–34, 2017.

[57] A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, vol. 24, no. 4, pp. 947–965, 2016.

[58] B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L. B. Said, "On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring," in *International Symposium on Search Based Software Engineering*. Springer, Cham, 2014, pp. 31–45.

[59] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *Journal of Systems and Software*, vol. 97, pp. 1–14, 2014.

[60] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[61] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, 2006, pp. 1909–1916.

[62] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1106–1113.

[63] R. Morales, F. Chicano, F. Khomh, and G. Antoniol, "Efficient refactoring scheduling based on partial order reduction," *Journal of Systems and Software*, vol. 145, pp. 25–51, 2018.

[64] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers' context for automatic refactoring of software anti-patterns," *Journal of systems and software*, vol. 128, pp. 236–251, 2017.

[65] A.-R. Han and S. Cha, "Two-phase assessment approach to improve the efficiency of refactoring identification," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 1001–1023, 2017.

[66] T. do Nascimento Ferreira, A. A. Araújo, A. D. B. Neto, and J. T. de Souza, "Incorporating user preferences in ant colony optimization for the next release problem," *Applied Soft Computing*, vol. 49, pp. 1283–1296, 2016.

[67] A. Ramirez, J. R. Romero, and S. Ventura, "Interactive multi-objective evolutionary optimization of software architectures," *Information Sciences*, vol. 463, pp. 92–109, 2018.

[68] A. Ramirez, J. R. Romero, and C. L. Simons, "A systematic review of interaction in search-based software engineering," *IEEE Transactions on Software Engineering*, vol. 45, no. 8, pp. 760–781, 2018.

[69] D. Dig, "A refactoring approach to parallelism," *IEEE software*, vol. 28, no. 1, pp. 17–22, 2010.

Soumaya Rebai is a Ph.D. student in the intelligent Software Engineering group at the University of Michigan. Her primary research interests are Search-Based Software Engineering, documentation generation, refactoring and services computing.

Vahid Alizadeh is currently a Ph.D. student in the intelligent Software Engineering group at the University of Michigan. His Ph.D. project is concerned with the application of intelligent search and machine learning in different software engineering areas such as refactoring, testing, and documentation. His current research interests are Search-Based Software Engineering, Refactoring, Artificial Intelligence, data analytics and software quality.

Marouane Kessentini is a recipient of the prestigious 2018 President of Tunisia distinguished research award, the University distinguished teaching award, the University distinguished digital education award, the College of Engineering and Computer Science distinguished research award, 4 best paper awards, and his AI-based software refactoring invention, licensed and deployed by industrial partners, is selected as one of the Top 8 inventions at the University of Michigan for 2018 (including the three campuses), among over 500 inventions, by the UM Technology Transfer Office. He is currently a tenured associate professor and leading a research group on Software Engineering Intelligence. Prior to joining UM in 2013, He received his Ph.D. from the University of Montreal in Canada in 2012. He received several grants from both industry and federal agencies and published over 110 papers in top journals and conferences. He has several collaborations with industry on the use of computational search, machine learning and evolutionary algorithms to address software engineering and services computing problems.

Houcem Fehri is currently a Ph.D. student in the intelligent Software Engineering group at the University of Michigan. His primary research interests are Search-Based Software Engineering, Optimization, and refactoring.

Rick Kazman is a Professor at the University of Hawaii and a Principal Researcher at the Software Engineering Institute of Carnegie Mellon University. His primary research interests are software architecture, design and analysis tools, software visualization, and software engineering economics. He also has interests in human-computer interaction and information retrieval. Kazman has created several highly influential methods and tools for architecture analysis, including the SAAM (Software Architecture Analysis Method), the ATAM (Architecture Tradeoff Analysis Method), the CBAM (Cost-Benefit Analysis Method) and the Dali architecture reverse engineering tool.