From Multi-Objective to Mono-Objective Refactoring via Developer's Knowledge Extraction

Vahid Alizadeh
CIS Department
University of Michigan
Dearborn, Michigan, USA
alizadeh@umich.edu

Houcem Fehri
CIS Department
University of Michigan
Dearborn, Michigan, USA
houcemf@umich.edu

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

***Abstract*—Refactoring studies either aggregated quality metrics to evaluate possible code changes or treated them separately to find trade-offs. For the first category of work, it is challenging to define upfront the weights for the quality objectives since developers are not able to express them upfront. For the second category of work, the number of possible trade-offs between quality objectives is large which makes developers reluctant to look at many refactoring solutions. In this paper, we propose, for the first time, a way to convert multi-objective search into a mono-objective one after interacting with the developer to identify a good refactoring solution based on his preferences. The first step consists of using a multi-objective search to generate different possible refactoring strategies by finding a trade-off between several conflicting quality attributes. Then, an unsupervised learning algorithm clusters the different trade-off solutions, called the Pareto front, to guide the developers in selecting their region of interests and to reduce the number of refactoring options to explore. Finally, the extracted preferences from the developer are used to transform the multi-objective search into a mono-objective one by taking the preferred cluster of the Pareto front as the initial population for the mono-objective search and generating an evaluation function based on the weights that are automatically computed from the position of the cluster in the Pareto front. Thus, the developer will just interact with only one refactoring solution generated by the mono-objective search. We selected 32 participants to manually evaluate the effectiveness of our tool on 7 open source projects and one industrial project. The results show that the recommended refactorings are more accurate than the current state of the art.**

***Index Terms*—Search Based Software Engineering, Interactive Refactoring, Software Quality**

I. INTRODUCTION

Software restructuring, or refactoring [1], is critical to improve software quality and developers' productivity, but can be complex, expensive, and risky [2]–[4]. A recent study [5] shows that developers are spending over 50% of their time struggling with existing code (e.g. understanding, restructuring, etc.) rather than creating new code.

While code-level refactoring, such as *Move-Method, Pullup-Method*, etc, is widely studied and well-supported by tools [6]–[16], **understanding the *refactoring rationale***, or the preferences of developers, is still lacking and yet not well supported. In our recent survey, supported by an NSF I-Corps [1] project, with 127 developers at 38 medium and large companies (Google, eBay, IBM, Amazon, etc.), 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. anti-patterns and low quality metrics/attributes) and refactorings but do not specify where to start or how they can be relevant for their context and preferences. This observation is consistent with another recent study [17]. Furthermore, refactoring is a human activity that cannot be fully automated and requires developers' insight to accept, modify, or reject some of these recommendations because the developers understand the problem domain intuitively 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 the quality issues are well detected, due to the subjective nature of software design [12], [14], [16], [18]–[21]. However, manual refactoring can be error-prone and time-consuming [22], [23].

Few studies have been proposed, recently, to interactively evaluate refactoring recommendations by developers [17], [24]–[27]. The developers can provide feedback about the refactored code and introduce manual changes to some of the recommendations. However, this interactive process can be expensive since developers must evaluate a large number of possible refactoring strategies/solutions and eliminate irrelevant ones. Both interactive and automated refactoring approaches have to deal with a big challenge to consider many quality attributes for the generation of refactoring solutions. Thus, refactoring studies either aggregated these quality metrics to evaluate possible code changes or treated them separately to find trade-offs [12], [16]–[19], [21], [25], [28]. However, it is challenging to define upfront the weights for the quality objectives since developers are not able to express them upfront. Furthermore, the number of possible trade-offs between quality objectives is large which makes developers reluctant to look at many refactoring solutions due to the time-consuming and confusing process.

In this paper, we propose an approach that takes advantage of both existing categories of refactoring work. Thus, we propose, for the first time, a way to convert multi-objective search into a mono-objective one after few interactions with the developer. The first step consists of using a multi-objective search, based on the evolutionary algorithm NSGA-II [29], to generate a diverse set of refactoring strategies by finding a trade-off between several conflicting quality attributes. Then,

[1] https://www.nsf.gov/news/special_reports/i-corps

an unsupervised learning algorithm clusters the different trade-off solutions, called the Pareto front, to guide the developers in selecting their region of interests and reduce the number of refactoring options to explore. Finally, the extracted preferences from the developer are used to transform the multi-objective search into a mono-objective one by taking the preferred cluster of the Pareto front as the initial population for the mono-objective search and generating an evaluation function based on the weights that are automatically calculated from the center of the preferred cluster in the Pareto front. Therefore, the developer will just interact with only one refactoring solution generated by the mono-objective search.

Our approach is taking the advantages of mono-objective search, 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 as an output. The clustering and interactive algorithms are useful in terms of extracting developers' knowledge and preferences. Mono-objective algorithms are the best in terms of optimization power once the evaluation function is well-defined and generate only one solution as an output. We selected 32 active developers to manually evaluate the effectiveness of our tool on 6 open source projects and one industrial system. The results show that the participants found their desired refactorings faster and more accurate than the current state of the art. A tool demo of our interactive refactoring tool of this paper and an appendix containing all the details of the experiments can be found in the following link [30].

II. MOTIVATIONS

While successful tools for refactoring have been proposed, 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. 112 of these interviews were conducted face-to-face.

The question we encounter most during our industrial collaborations in refactoring is *"We agree that this is a problem, but what should we do?"* Although code-level anti-patterns can largely be automated, higher-level refactoring — such as redistributing functionality into different components, decoupling a large code base into smaller modules, redesigning to a design pattern— requires abstractions determined by human architects. In these cases, the architect usually has a desired design in mind as the refactoring target, and the developer needs to conduct a series of low-level refactorings to achieve this target. Without explicit guidance about which path to take, such refactoring tasks can be demanding: It took a software company several weeks to refactor the architecture of a medium-size project (40K LOC) [27]. Several books [2], [31], [32] on refactoring legacy code and workshops on technical debt [33] present the substantial costs and risks of large-scale refactorings. For example, Tokuda and Batory [34] presented two case studies where architectural refactoring involved more than 800 steps, estimated to take more than 2 weeks.

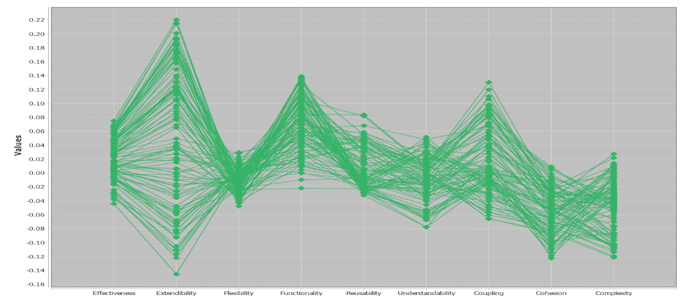

Figure 1: The output of a multi-objective refactoring tool [26] finding trade-offs between QMOOD quality attributes on ganttproject v1.10.2

Prior work [35] shows that even semi-automated tools for lower-level refactorings have been underutilized. Given that fully automatic refactoring usually does not lead to the desired architecture and that a designer's feedback should be included, we propose *an interactive architecture refactoring recommendation system* to integrate higher-level abstractions from humans with lower-level refactoring automation. Over 77% of the 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."*

Based on our extensive experience working on licensing refactoring research prototypes to industry, developers always have a concern on expressing their preferences upfront as an input for a tool to guide refactoring suggestions. They prefer to get insights from some generated refactoring solutions then decide which quality attributes they want to improve. However, several existing refactoring tools fail to consider the developer perspective, as the developer has no opportunity to provide feedback on the refactoring solution as it is being created. Furthermore, as development must halt while the refactoring process executes, fully-automated refactoring methods are not useful for floss refactoring where the goal is to maintain good design quality while modifying existing functionality. The developers have to accept the entire refactoring solution even though they prefer, in general, step-wise approaches where the process is interactive and they have control of the refactorings being applied. Determining which quality attribute should be improved and how is never a pure 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 very 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. Figure 1 shows

an example of a Pareto front of non-dominated refactoring solutions improving the QMOOD [19] quality attributes of a Gantt Project generated using an existing tool [26]. QMOOD is one of the widely accepted software quality models in industry based on our previous collaborations with industry and recent studies [26], [27], [36]–[38]. While developers were interested to give a feedback for some of the refactoring solutions but they expected to see only one refactoring solution in the future after this interaction. This means after the first round of optimization and evaluation, the developer wants to have a single personalized solution. The extraction of developers' knowledge from the interaction data is beyond the scope of existing refactoring tools. Furthermore, existing search-based software engineering approaches did not explore converting multi-objective into mono-objective search after knowledge extraction. While multi-objective search algorithms are known to be good in diversifying solutions but they cannot beat well-formulated mono-objective search algorithms in terms of the optimization power.

III. Approach Overview

Our proposed approach includes three main phases. First, we use multi-objective optimization to find a set of non-dominated refactoring solutions capable of improving the quality of the software. Second, we cluster these solutions and obtain the center of each cluster to reduce the exploration effort of the Pareto-front by the decision maker. Third, we extract automatically the preferences and utilize them to transform the multi-objective problem to a mono-objective one after the user's interaction and evaluation of the recommended refactoring solutions. Finally, the output of the mono-objective search is a single solution fitting to the user's expectations and preferences then the developer can interact with that solution if needed and continue the execution of the mono-objective algorithm until selecting a final refactoring solution. The pseudo code of our algorithm is described in the appendix [30]. In the following, we will explain, in details, the steps of our proposed technique.

A. Phase 1: Multi-Objective Refactoring

Considering the goals and objectives of refactoring a software, this challenging task can be formulated as a multi-objective optimization problem as follow:

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

where S is the subset of all feasible solution, R^m, which satisfy the inequality and equality constraints, $g(x)$ and $h(x)$, respectively. 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.

The result of the first phase of our approach, as it is shown in the Figure 1, is a set of Pareto-optimal refactoring solutions. In the following subsections, we briefly summarize the adaptation of multi-objective search to the software refactoring problem.

1) ***Solution Representation****:* We encode a refactoring solution as an ordered vector of multiple refactoring operations. Each operation is defined by an action (eg. move method, extract class, etc.) and its specific controlling parameters (e.g. source and target classes, attributes, methods, etc.). 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. During the process of population initialization or mutation operation of the algorithm, the refactoring operation and its parameters are formed randomly. Therefore, due to the random nature of the 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 [39].

2) ***Fitness Functions****:* We used the Quality Model for Object-Oriented Design (QMOOD) [40] as a means of estimating the effect of a refactoring operation on the quality of a software. This model is developed based on the international standard for software product quality measurement and widely used in 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. Therefore, the fitness functions to be calculated are: Understandability, Functionality, Reusability, Effectiveness Flexibility, Extendibility, Complexity, Cohesion and Coupling. We considered the relative change of these quality attributes after applying a refactoring solution as the fitness function formulated 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.

B. Phase 2: Clustering Refactoring Solutions and Extracting Developer Preferences

One of the most challenging and tedious tasks for the user during every multi-objective optimization process is the 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 main goal of this step is to cluster and categorize the solutions based on their similarity in the objective space. These clusters of solutions help the user to have an overview of the possible existing options. Therefore, this technique gives the user a more clear initial step of exploration where she

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 I start to find my preferred solution?". This observation is valid for various SBSE applications using multi-objective search [27].

1) Clustering the Pareto-front: Clustering is an unsupervised learning method to discover a meaningful underlying structure and pattern between a set of unlabelled data. It puts the data into groups where the similarity of the data points within each group is maximized while keeping a minimized similarity between the groups.

Determining the optimal number of clusters is a fundamental issue in clustering techniques. One of the methods 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 utilized Calinski Harabasz (CH) Index which is an internal clustering validation measure based on two criteria: compactness and separation [41]. CH assesses the clustering outcomes based on the average sum of squares between and within clusters. Therefore, we execute the clustering algorithm on the Pareto-front solutions with a various number of components as the input. The CH score is calculated for each execution, and the result with the highest CH score is recognized as the optimal way of clustering our data.

After determining the best number of clusters, we employed 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. The parameters are the number of distributions, Mean, Co-variance, and Mixing coefficient. The optimal values for these parameters are estimated using Expectation-Maximization (EM) algorithm [42]. EM trains the variables through two steps iterative process.
After the convergence of EM, the membership degree of each solution to a fitted Gaussian or cluster is kept for preference extraction step. Furthermore, in order to find a representative member of each cluster, we measure the corresponding density for each solution and select the solution with the highest density value.

The line chart of Pareto-front solutions after clustering is shown in Figure 2. Compared to the original chart in Figure 1, the color of each line indicates its cluster and the solutions marked with triangles are the cluster representative member.

2) Interaction and preference extraction: The results of multi-objective refactoring after clustering are presented to the user in various interactive tables and charts alongside with extensive analysis to explain and guide the process of decision making. These explanations are automatically generated using statistical analysis and investigating the content of the solutions and clusters.
The explanations of Pareto-front assist the user to gain a vibrant picture of the available options, costs, and benefits.

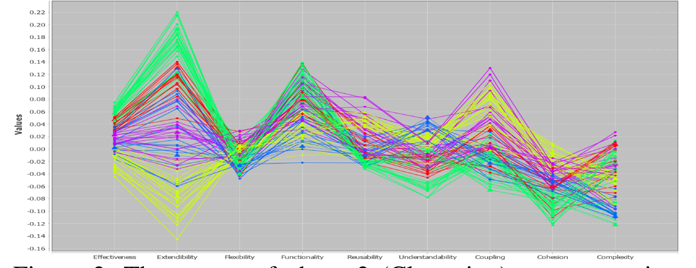

Figure 2: The output of phase 2 (Clustering) on ganttproject v1.10.2.

Furthermore, by clustering similar solutions, it requires less effort to initiate the exploration and finally making a decision. The user may begin to evaluate the cluster center solutions or expand the search to the other solutions in the cluster. The interaction can be performed at the cluster, solution, and refactoring operation levels depending on the user's desire. The feedback is quantified to a continuous score in the range of [-1,1].
The developer can evaluate a solution by modifying its refactoring operations (edit, add, delete, re-order) or just rate the whole solution or cluster. After the developers interaction, Solution score ($Score_{s_i}$) and Cluster score ($Score_{c_k}$) are computed as the average score of operations in a solution and the average score of solutions in a cluster, respectively.
The cluster of solutions with the highest score is considered as the region of interest in the solution space. It indicates the preferred objectives, code locations, and refactoring operations. For instance, if the solutions in the selected cluster tend to emphasize on improving Extendibility by applying mostly Generalization category of refactoring operations on certain packages or classes of the software, we consider these factors as the user preferences in the execution of the next phase of our approach.
For this purpose, we compute the weighted probability of refactoring operations (RWP) and target classes of the source code (CWP) as follow:

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

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

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.

C. Phase 3: Preference-base Mono-objective Refactoring

One of the main contributions of this paper is the ability to convert a multi-objective algorithm into a mono-objective one after interacting with the developer to extract his preferences and knowledge. Mono-objective algorithms are known to be the best in terms of optimization but require that the fitness

function should be well defined based on the decision maker's preferences. The Multi-objective Evolutionary Algorithm used in Phase 1 might not provide high-quality solutions in the region of interest of the developer because of the high dimensionality nature of the problem and the need to find trade-offs. Therefore, it is important to consider the user preferences extracted in Phase 2.

The goal of this phase is to use the preferences extracted from the developer after the multi-objective optimization to transform the problem into a single objective optimization problem by aggregating objectives according to the user's preferences. This transformation gives the decision maker a single solution in the region of interest. Consequently, our proposed approach is a combination of all three categories of preference-based search where the preferences are expressed after the first evolutionary process, then they are incorporated to guide the single objective optimization.
One way to convert a multi-objective optimization problem to a mono-objective problem and achieve a single solution is called the Weighted Sum Method (WSM). In this method, the single preference fitness function is computed as a linear weighted sum of multiple objectives. The main drawback of the WSM method is that it needs the weights parameters to be given. Fortunately, in our case, those parameters are computed automatically from the decision maker preferences of the interactive optimization process (preferred cluster) in the objectives space (quality attributes). Thus, the weight of one or more objectives can get the value 0 (or almost) if the selected cluster by the developer penalized them while favoring other objectives. Also, the WSM is not computationally expensive unlike the other scalarization methods. Therefore, the optimization problem can be formulated as:

$$\begin{aligned} &Minimize && PF(\boldsymbol{X}) = \sum_{i=1}^{M} \omega_i f_i(x), \\ &Subject\ to && \boldsymbol{X} \in S, \\ & && \omega_i \geq 0; \sum_{i=1}^{M} \omega_i = 1; \end{aligned}$$

Where $PF(X)$ is the single scalar preference function, and weights ω_i reflects the a priori preferences of the user over the objectives. The weights are a tool to steer the search along the Pareto-front into a direction determined by the user. This way, the decision maker is offered a single solution that corresponds to his interests and reduces on him the burden of having to go through multiple solutions.
In order to solve the converted mono-objective problem, we adopted a standard Genetic Algorithm (GA). To adapt the GA algorithm to our refactoring context, we use the same solution representation and quality fitness functions as reported in phase 1. Algorithm 1 explains the steps of this phase.

We begin by normalizing the values of each fitness function separately for all solutions in the preferred cluster. Then, we pick the center of the cluster and normalize this solution's

Algorithm 1: Preference-based Mono-objective Optimization

```
Input  : Preferences (P),
         Preferred Cluster (PC),
         Cluster Center (CC)
Output: RecommendedSolution
begin Calculating Objective's Weight
    NormalizeAll(PC);
    W_i ← NormalizeUnitSum(CC);
begin Mono-objective Optimization
    initialPopulation ← PC;
    if size(initialPopulation) ¡ N then
        initialPopulation + = fillPopulation();
    while ¬ stoppingCondition() do
        customSelection();
        Crossover();
        customMutation();
        fitness ← weightedSum(f_i, w_i);
        evaluate(fitness);
    RecommendedSolution ← getFittest();
Return RecommendedSolution;
```

fitness values. We use the result as the aggregation weights in WSM where the condition $\sum_{i=1}^{M} \omega_i = 1$ is satisfied. Therefore, we assign the importance of the objectives accordingly based on the intuition and preferences of the user.
The obtained single fitness function is employed to evaluate the solutions in the execution of adapted GA. We consider the preferences extracted in the previous phase, to customize the components of GA via Preference-based initial population generation and Preference-based Mutation/Selection operators. Instead of generating the initial population randomly, we acquire the user preferred cluster as the elite set of solution from which the search process is initiated. Thus, we do not generate solutions randomly for the mono-objective GA but we take the solutions in the preferred cluster as the initial population thus we do not lose the knowledge extracted from the developer. Since the number of solutions in the preferred cluster might be less than the required size, we form new individuals to fill the gap. The new solutions are produced based on CWP and RWP probability distribution. It means, for each new solution, we pick the operation and its target class attribute from a distribution aligned with the preferences of the user.
The preference probability distribution for code locations and refactoring operations are used during the mutation process similarly.
The selection operator which is used to keep the most valuable solutions of the population is customized to consider the distance of a solution to the region of interest. Therefore, being closer to the preferences and having higher fitness value are both measured to be factors of selecting an elite

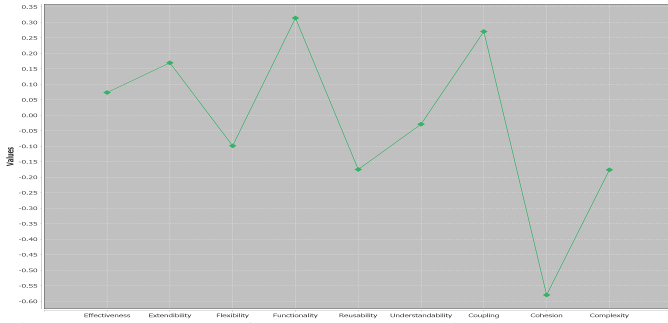

Figure 3: The output of phase 3 (Mono-objective) on GanttProject v1.10.2 system

solution. Finally, the solutions are evaluated via the preference function aggregated from multiple objectives. When the stopping condition is satisfied, the single optimal solution is recommended to the user. Similar to Phase 1, the user can interact with this solution via editing/adding/removing the refactoring operations.

If the developer is still not satisfied, he can proceed with the search process in two ways: 1) going back to Phase 2 and selecting another cluster. 2) returning to Phase 1 and executing the multi-objective optimization again where, in this time, the approach is customized to accommodate the prior knowledge of the preferences. The result of Phase 3 is represented in Figure 3. As it is shown, at this step, the user is required to only interact with one customized solution where it takes shorter effort and time and produces less confusion.

IV. Evaluation

A. Research Questions

We defined three main research questions to measure the correctness, relevance and benefits of our interactive clustering-based multi-objective refactoring tool comparing to existing approaches that are based on interactive multi-objective search [43], fully automated multi-objective search (Ouni et al.) [44] and fully automated deterministic tool not based on heuristic search (JDeodorant) [45]. A tool demo of our interactive refactoring tool and supplementary appendix materials (questionnaire, setup of the experiments, statistical analyses, and detailed results) can be found in our study's website [2]. The appendix includes:(a) Study-steps; (b) Pre/Post-study-questionnaires (QMOOD, experience, comments, etc.); (c) Parameters-tuning;(d) Box-plots/statistical-tests to give more details than the median.

The research questions are as follows:

- **RQ1: Benefits.** To what extent can our approach make relevant recommendations for developers compared to existing refactoring techniques?
- **RQ2: The relevance of developers' knowledge extraction.** To what extent can our approach reduce the interaction effort, comparing to existing refactoring techniques, while quickly identifying relevant refactoring recommendations?

[2]Demo and supplementary appendix materials can be found in the following link: https://sites.google.com/view/scam2019

Table I: 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

- **RQ3: Tool usefulness.** How do developers evaluate the relevance of our tool in practice (post-study survey)?

B. Experimental Setup

We considered a total of seven systems summarized in Table I to address the above research questions. We selected these seven systems because of their size, have been actively developed over the past 10 years and extensively analyzed by the competitive tools considered in this work. UTest[3] 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 are very knowledgeable about refactoring (they are part of the maintenance team). Table I 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 32 participants to identify and manually evaluate the relevance of the refactoring solutions that they selected using four other tools. The first tool is an existing interactive multi-objective refactoring approach proposed by Mkaouer et al. [24], [26] 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. The second tool is an interactive clustering based multi-objective approach proposed by Alizadeh et al. [27] however they did not consider the developers' knowledge extraction neither the use of mono-objective search to directly converge towards one refactorings solution after extracting developers preferences. The comparison with these tools will help us evaluating the main new contribution of this paper related to converting multi-objective to a mono-objective one after extracting the developers' preferences from exploring the clusters and the Pareto front. We have also compared our IMMO approach to two fully-automated refactoring tools by means of Ouni [44] and JDeodorant [45]. Ouni [44] proposed a multi-objective refactoring formulation based on NSGA-II that generates a solution to maximize the design coherence and refactorings reuse from previous releases. JDeodorant [45] 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 these refactorings. We used these two competitive tools to evaluate the benefits of the interaction feature in helping developers identifying relevant refactorings especially with the preferences extraction feature and the mono-objective search.

We preferred not to use the antipatterns and internal quality indicators as proxies for estimating the refactorings relevance since the developers manual evaluation already includes the

[3]Company anonymized for double-blind.

review of the impact of suggested changes on the quality. Furthermore, not all the refactorings that improve any 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.

Unlike fixing bugs, refactoring is a very-subjective activity and there is no unique solution to refactor a code/design thus it is very difficult to construct a gold-standard for large-systems which makes calculating the recall very challenging. Does the deviation from an expected refactoring solution means that the recommendation is wrong or simply another way to refactor the code? The context of our work is related to "incremental" refactoring rather than the rare "root canal" refactoring where developers will look at the whole architecture/system to make major refactorings. In this context of incremental refactoring, the main factor is the precision. In addition, developers can check via our tool the impact of the refactoring solutions on the overall code quality using many attributes. Thus, they continue to interactively evaluate and apply refactorings until that they are satisfied in terms of improving the quality attributes that they consider them concerning. Our tool enables the developers to evaluate the current quality of the system then tuning the search algorithm to focus on specific locations of the code based on their needs. With the current large-size of the systems, it is unrealistic to look for all possible refactoring strategies targeting the whole project which is not also the scope of this paper(root-canal refactoring).

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. Although the vast majority of participants are already familiar with refactoring as part of their job and graduate studies, all the participants attended one lecture of two hours on software refactoring by the organizers of the experiments. The details of the selected participants can be found in Table II, including their programming experience (years) and level of familiarity with refactoring. Each participant was asked to assess the meaningfulness of the refactorings recommended after using up-to two out of the five tools on up-to two different systems to avoid the training threat. The participants did not "only" evaluate the suggested refactorings but were asked to configure, run and interact with the tools on the different systems. The only exceptions are related to the five participants from the industrial partner where they agreed to evaluate only the industrial software. We assigned the 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 at least one time on each of the seven systems. 3 out of 32 participants were asked to refactor two projects to ensure that all the seven projects are refactored using the five different tools. To mitigate the training threat, the counter-balanced design ensured that these three participants: (1) did not evaluate the same system using two different tools; (2) did not evaluate the same tool more than one time (even on different projects) and(3) did not evaluate the same type of technique more than one time. Thus, if the participant used a multi-objective tool, then he/she will evaluate JDeodorant (deterministic) on another project.

Table II: Selected programmers.

System	#Subjects	Avg. Prog. Exp.	Avg. Refactoring Exp.
ArgoUML	5	7.5	Very High
JHotDraw	5	8	Very High
Azureus	5	9.5	High
GanttProject	5	7	High
UTest	5	15.5	Very High
Apache Ant	5	9	Very High
JFreeChart	5	7	Very High

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 comparing to the one required once the mono-objective search is executed. This evaluation will help to understand if we efficiently extracted the developer preferences after the Pareto-front interactions. For this research question, we decided to limit the comparison to only the interactive multi-objective work of Mkaouer et al. [24], [26] and Alizadeh et al. [27] since they are the only ones offering interaction with the users and it will help us understand the real impact of the knowledge extraction and mono-objective features (not supported by existing studies) on the refactoring recommendations and interaction effort.

To answer RQ3, we collected the opinions of participants based on a post-study questionnaire. To better understand subjects' opinions with regard to usefulness and usability of our approach in a real setting, the post-study questionnaire was given to each participant after completing the refactoring tasks using our approach and all the techniques considered in our experiments. The questionnaires collected the opinions of the participants about their experience in using our tool compared to the remaining tools used in these experiments and their past experience.

The stopping criterion was set to 100,000 evaluations for all search algorithms in order 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 his/her preferences). The mono-objective search was limited to 10,000 evaluations after the interactions with the user. The other parameters' values are as follows for both the multi-objective and mono-objective algorithms: crossover probability = 0.4; mutation probability = 0.7 where the probability of gene modification is 0.5. 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.

C. Results

Results for RQ1: Benefits. Figure 4 summarizes the manual validation results of our IMMO approach comparing to the state of the art as evaluated by the participants. It is clear from

the overall results that interactive approaches generated much more relevant refactorings to the programmers comparing to the automated tools of Ouni et al. and JDeodorant. Among the interactive approaches, IMMO outperformed the existing interactive approaches of Mkaouer et al. and Alizadeh et al. which may confirm the importance of extracting the developers' preferences and the performance of mono-objective search in terms of optimization when the fitness function is well-defined based on knowledge extraction from the user. On average, for all of our seven studied projects, 89% of the proposed refactoring operations are considered to be useful by the software developers of our experiments. The remaining approaches have an average of 83%, 71%, 67%, and 56% respectively for Alizadeh et al. (interactive with 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 96% for the Azureus project, and the lowest score is 86% for JHotDraw. The participants were not guided on how to interact with the systems, and they mainly looked to the source code to understand the impact of recommended refactorings.

When comparing manually the results of the different tools, we found that automated refactorings generate a lot of false positive and noise of developers. Both Ouni et al. and JDeodroant tools recommended a large number of refactorings comparing the interactive tools where several of them are not interesting for the context of the developers thus they reject them even if they are correct. For instance, the developers of the industrial partner rejected several recommendations from these automated tools simply because they are related to a stable code or code fragments out of their interests. The majority of them will not change a code out of their ownership as well. Furthermore, they were not interested to blindly change anything in the code just to improve quality attributes. Comparing to the remaining interactive approaches, we found that some of the refactoring solutions of IMMO will never be proposed by Mkaouer et al. or Alizadeh et al. since they are emphasizing specific objectives than others. 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 use of mono-objective search when the preferences of the user are extracted is powerful both in terms of interaction and optimization. The mono-objective search helped to focus on specific code locations and quality attributes rather than wasting the optimization power on multiple objectives. To conclude, our IMMO approach outperformed the four remaining refactoring approaches in terms of recommending relevant refactoring solutions for developers (RQ1).

Results for RQ2: The relevance of developers' knowledge extraction. Figures 5, 6 and 7 give an overview about the number of refactorings of the selected solution, number of required interaction 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.

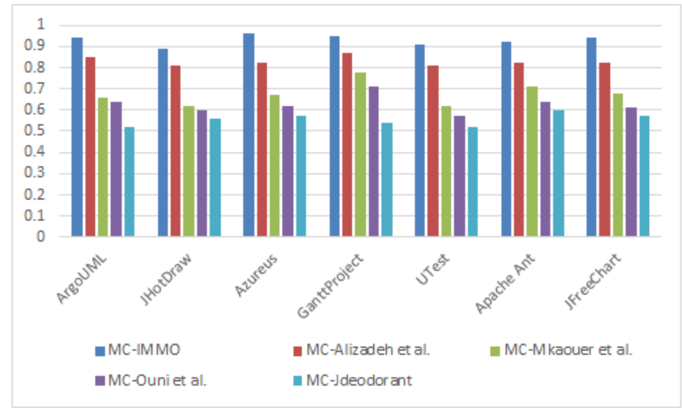

Figure 4: Average manual evaluations, MC, on the 7 systems.

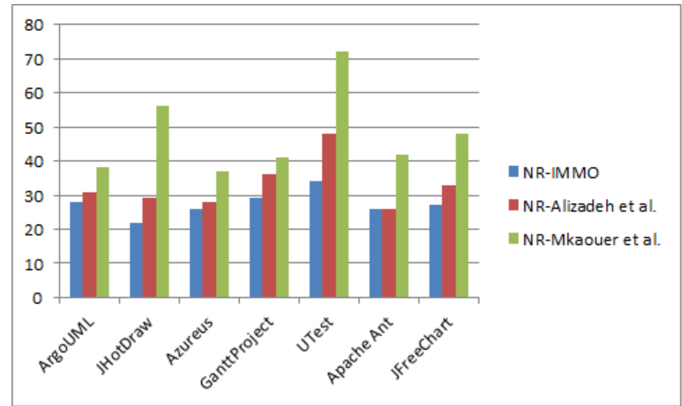

Figure 5: The median number of recommended refactorings, NR, of the selected solution on the 7 systems.

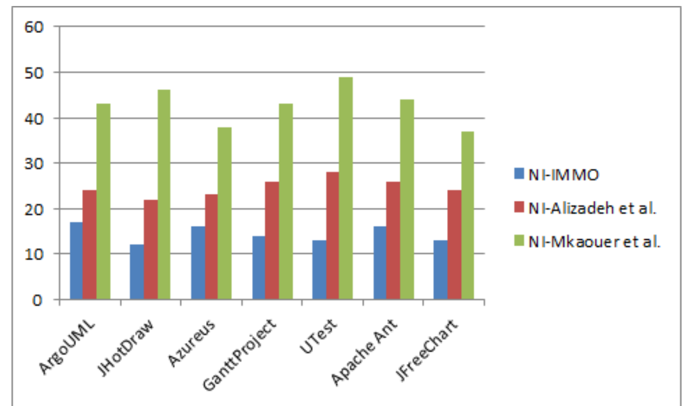

Figure 6: The median number of required interactions (accept/reject/modify/selection), NI, on the 7 systems.

Based on the results of Figure5, it is clear that our approach significantly reduced the number of recommended refactorings comparing to both other interactive approaches while increasing the manual correctness as described in RQ1. The highest number of refactorings was observed on the industrial system with 34 refactorings using IMMO, 48 using Alizadeh et al. and 72 refactorings using Mkaouer et al. It 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 IMMO comparing to interactive approaches is mainly related to the elimination of the noise in multi-objective search to handle

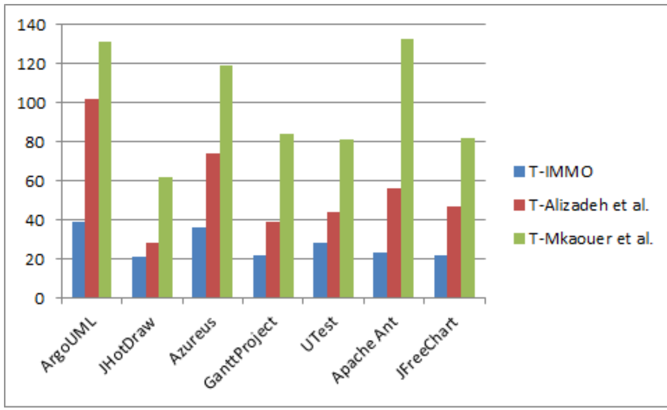

Figure 7: The average execution time, T, in minutes on the 7 systems.

multiple quality attributes and the extraction of developers preferences. It is normal to see fewer refactorings when the search space is reduced which was the case of IMMO.

Figure 6 shows that IMMO required much fewer developer interactions than the remaining interactive approaches. For instance, only 13 interactions to modify, reject and select refactorings were observed on JFreeChart using our approach while 24 and 37 interactions were needed respectively for Vahid et al. and Mkaouer et al. The reduction of the number of interactions are mainly due to the move from multi-objective to mono-objective search after one round of interactions since the developers will not deal anymore with a set of solutions in the front but only one.

The participants also spent less time to find the most relevant refactorings on the different systems compared to the remaining interactive approaches. For instance, the average time is reduced by over 65% comparing to Mkaouer et al. for the case of JHotDraw (from 62 minutes to just 21 minutes). The time includes the execution of the multi-objective and mono-objective search (if any), the clustering (if any) and the different phases of interaction until the developer is satisfied with a specific solution. The drop of the execution time is mainly explained by the fast execution of the mono-objective search and the reduced search space after the interactions with the developers.

Figure 8 shows a qualitative example extracted from our experiments using IMMO on the Gantt project based on the four interaction phases. After the generation of the Pareto front, the clustering algorithm of the non-dominated refactoring solutions identified three different main clusters for the two objectives selected by the developer (extendibility and effectiveness). During the first phase, the developer selected the cluster with id 0 as the preferred one after exploring several refactoring solutions in that cluster including mainly the solution located at the center of the cluster. Thus, the next phase took the solutions in the id 0 cluster and generated an initial population for the mono-objective genetic algorithm, and the center of the selected cluster was used to generate the weights for the fitness function. The output of the mono-objective search is one refactoring solution (instead of many solutions like the multi-objective search) that optimize better the selected objectives than all the solutions in the preferred cluster. Finally, the interactions with the user (accept/reject/modify some refactorings) on that solution helped to converge towards a better final solution by continuing the execution of the mono-objective search.

Results for RQ3: Impact. We did a post-study questionnaire to collect the feedback of the developers about the different evaluated refactoring tools. We found that 26 out the 32 participants highlighted that they preferred IMMO comparing to the remaining tools because of mainly the ability to interact with one solution (instead of a front) and the fast improvement of the refactoring results after just a few interactions. One of the participants submitted the following message: *"It is really great to see only refactoring solutions meeting my needs after just a couple of interactions!"*.

21 out the 32 participants appreciated the combination of multi-objective and mono-objective search algorithms. They found that multi-objective search was useful to get some insights about several possible strategies to improve the code then the mono-objective powerful in generating better solutions based on their feedback. For instance, one of the developers commented the following: *"I had no idea about the beginning from where to start but looking to the first set of recommendations and their code impact, I had a clear idea on what quality metrics I need to target then it was easy to just give feedback to only one strategy (solution)."* 29 out the 32 participants found that the major refactoring suggestions of both Ouni et al. and JDeodorant hard to evaluate and understand. They found the lack of interactions as a main limitation since they have to accept or reject the whole refactoring suggestions and it is difficult to estimate their impacts. The participants noticed, in the survey, that they were satisfied with the the considered quality attributes and refactoring types by our tool. They did not suggest to add new types of refactoring or quality attribute.

V. Threats to Validity

Conclusion validity. Since we used a variety of computational search and machine learning algorithms, the parameter tuning used in our experiments creates an internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution in order to provide the best possible performance. Another conclusion threat is the number of interactions with the developers since we did not force them to use the same interaction effort which may sometimes explain the out-performance of our approach. However, the participants were given the same maximum amount of time to use the tool (limited to 3 hours).

Internal validity. The variation of correctness and speed between the different groups when using our approach and other tools can be one internal threat. Our approach may not be the only reason for the superior performance because the

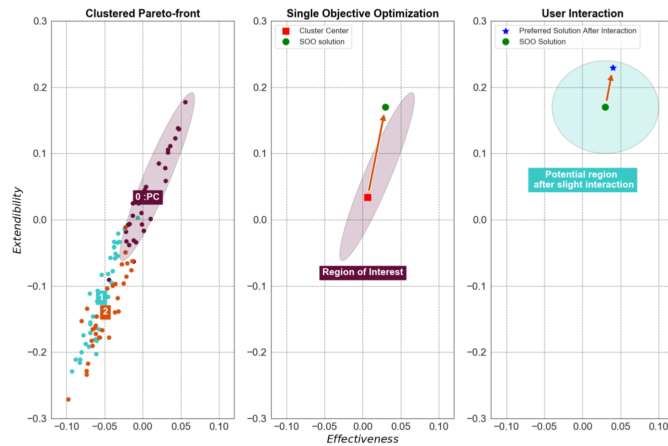

Figure 8: A qualitative example of three executions extracted from our experiments on Ganttproject to illustrate the process of converting a multi-objective search into a mono-objective one.

participants have different programming skills and familiarity with refactoring tools. To counteract this, we assigned the developers to different groups according to their programming experience so as to reduce the gap between the different groups, and we also adopted a counter-balanced design. Regarding the selected participants, we have taken precautions to ensure that our participants represent a diverse set of software developers with experience in refactoring, and also that the groups formed had, in some sense, a similar average skill set in the refactoring area.

External validity. The first threat is the limited number of participants and evaluated systems, which externally threatens the generalizability of our results. In addition, our study was limited to the use of specific refactoring types and quality attributes. Furthermore, we mainly evaluated our approach using NSGA-II and GA algorithms, but other state-of-the-art metaheuristic algorithms can be used. Future replications of this study are necessary to confirm our findings.

VI. Related Work

Search-based techniques [46]–[49] are widely studied to automate software refactoring where the goal is to improve the design quality of a system based mainly on a set of software metrics. The majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings [47]–[57]. Seng et al. [58] have proposed a single-objective optimization approach using a genetic algorithm to suggest a list of refactorings to improve software quality. The work of O'Keeffe et al. [59] uses various local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. They use the QMOOD metrics suite to evaluate the improvement in quality. The majority of existing multi-objective refactoring techniques [18], [28], [44], [60] 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.

Some recent studies [17], [26], [27] extended a previous work [24] to propose an interactive search based approach for refactoring recommendations. The developers have to specify a desired design at the architecture level then the proposed approach try to find the relevant refactorings that can generate a similar design to the expected one. In our work, we do not consider the use of a desired design, thus developers are not required to manually modify the current architecture of the system to get refactoring recommendations. Furthermore, developers maybe interested to change the architecture mainly when they want to introduce an extensive number of refactorings that radically change the architecture to support new features.

VII. Conclusion

In this paper, we proposed a novel approach to extract developers' knowledge and preferences to find good refactoring recommendations. We combined the use of multi-objective search, clustering, mono-objective search and users interaction in our approach. To evaluate the effectiveness of our tool, we conducted an evaluation with 32 software developers who evaluated the tool and compared it with the state-of-the-art refactoring techniques. As part of our future work, we are planning to evaluate our approach on further projects and a more extensive set of participants. We will also adapt our approach to address other problems requiring developer interactions such as bugs localization.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Jul. 1999.

[2] J. Kerievsky, *Refactoring to Patterns*. Pearson Higher Education, 2004.

[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 *Proc. 37th*, May 2015.

[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] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the International Symposium on Foundations of Software Engineering*, ser. FSE, 2009, pp. 371–372.
[7] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *20th International Conference on Software Maintenance (ICSM)*, Sept 2004, pp. 350–359.
[8] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proceedings of the International Conference on Software Engineering*, 2009, pp. 287–297.
[9] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP*, vol. 4067, 2006, pp. 404–428.
[10] 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.
[11] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 1145–1156.
[12] 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.
[13] 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, 2012.
[14] 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.
[15] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *11th Working Conference on Reverse Engineering (WCRE)*, 2004, pp. 144–151.
[16] 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.
[17] 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.
[18] 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.
[19] 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.
[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] I. H. Moghadam and M. O. Cinneide, "Automated refactoring using design differencing," in *Software maintenance and reengineering (CSMR), 2012 16th European conference on*. IEEE, 2012, pp. 43–52.
[22] 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, 2012.
[23] 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.
[24] 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 International Conference on Automated Software Engineering*, 2014, pp. 331–336.
[25] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in eclipse: the aries project," in *34th International Conference on Software Engineering (ICSE)*. IEEE Press, 2012, pp. 1419–1422.
[26] 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.
[27] 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*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 464–474. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238217
[28] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, "Experimental assessment of software metrics using automated refactoring," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2012, pp. 49–58.
[29] 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.
[30] "Less is More: From Multi-Objective to Mono-Objective Refactoring." [Online]. Available: https://sites.google.com/view/scam2019
[31] M. Feathers, *Working Effectively with Legacy Code*. Prentice Hall PTR, 2004.
[32] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
[33] "The Seventh International Workshop on Managing Technical Debt," http://www.sei.cmu.edu/community/td2015/.
[34] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings," in *Proceedings of International Conference on Automated Software Engineering*, 1999, pp. 174–181.
[35] E. R. Murphy-Hill and A. P. Black, "Why don't people use refactoring tools?" in *Proceedings of the Workshop on Refactoring Tools in conjunction with the European Conference on Object-Oriented Programming*, 2007, pp. 60–61.
[36] 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.
[37] 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.
[38] 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.
[39] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.
[40] J. Bansiya and C. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.
[41] 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.
[42] 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.
[43] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pp. 331–336, 2014.
[44] 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.
[45] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1037–1039.
[46] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.
[47] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, "Generating transformation rules from examples for behavioral models," in *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*. ACM, 2010, p. 2.

[48] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm," *Software Quality Journal*, vol. 25, no. 2, pp. 473–501, 2017.

[49] ——, "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm," *Software Quality Journal*, vol. 25, no. 2, pp. 473–501, 2017.

[50] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi, "Model transformation modularization as a many-objective optimization problem," *IEEE Transactions on Software Engineering*, 2017.

[51] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.

[52] H. Wang, M. Kessentini, and A. Ouni, "Bi-level identification of web service defects," in *International Conference on Service-Oriented Computing*. Springer, 2016, pp. 352–368.

[53] 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.

[54] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What you like in design use to correct bad-smells," *Software Quality Journal*, vol. 21, no. 4, pp. 551–571, 2013.

[55] A. ben Fadhel, M. Kessentini, P. Langer, and M. Wimmer, "Search-based detection of high-level model changes," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 212–221.

[56] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, "Search-based design defects detection by example," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, Berlin, Heidelberg, 2011, pp. 401–415.

[57] M. Kessentini, H. Sahraoui, and M. Boukadoum, "Example-based model-transformation testing," *Automated Software Engineering*, vol. 18, no. 2, pp. 199–224, 2011.

[58] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *8th annual Conference on Genetic and Evolutionary Computation (GECCO)*. ACM, 2006, pp. 1909–1916.

[59] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

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