Reducing Interactive Refactoring Effort via Clustering-Based Multi-objective Search

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

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

ABSTRACT

Refactoring is nowadays widely adopted in the industry because bad design decisions can be very costly and extremely risky. On the one hand, automated refactoring does not always lead to the desired design. On the other hand, manual refactoring is error-prone, time-consuming and not practical for radical changes. Thus, recent research trends in the field focused on integrating developers feedback into automated refactoring recommendations because developers understand the problem domain intuitively and may have a clear target design in mind. However, this interactive process can be repetitive, expensive, and tedious since developers must evaluate recommended refactorings, and adapt them to the targeted design especially in large systems where the number of possible strategies can grow exponentially.

In this paper, we propose an interactive approach combining the use of multi-objective and unsupervised learning to reduce the developer's interaction effort when refactoring systems. We generate, first, using multi-objective search 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 reduce the number of refactoring options to explore. The feedback from the developer, both at the cluster and solution levels, are used to automatically generate constraints to reduce the search space in the next iterations and focus on the region of developer preferences. We selected 14 active developers to manually evaluate the effectiveness our tool on 5 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.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**;

KEYWORDS

Search based software engineering, refactoring, multi-objective search, clustering

ACM Reference Format:
Vahid Alizadeh and Marouane Kessentini. 2018. Reducing Interactive Refactoring Effort via Clustering-Based Multi-objective Search. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238217

1 INTRODUCTION

As projects evolve, developers frequently postpone necessary system restructuring, known as refactoring [17], in the rush to deliver a new release until a crisis happens. When that occurs it often results in substantially degraded system performance, perhaps an inability to support new features, or even in terminally broken system architecture. Thus, refactoring received much attention during the last two decades to propose solutions that can manage the growing complexity of software systems nowadays. Most existing studies focus on either manual or fully automated code-level refactoring. The manual support, integrated into modern IDEs such as Eclipse, NetBeans, and Visual Studio [5, 14, 15, 21–23, 26, 27, 30, 32, 33], consists of helping developers to apply refactorings based on automated routines that can check a list of pre- and post-conditions but they have to specify manually which types of refactoring to be applied, such as extract class or move method, and where. The fully automated techniques try to identify refactoring opportunities and which refactorings to apply using static and dynamic analysis, and the history of changes. However, design restructuring is a human activity that cannot be fully automated because developers understand the problem domain intuitively and they have targeted design goals in mind. Thus, several empirical studies show that fully automated refactoring does not always lead to the desired architecture [8, 10, 23, 24]. Furthermore, manual refactoring is error-prone, time consuming and not practical for radical changes. For instance, Batory et al. [22] presented several case studies where refactoring involved more than 750 refactoring steps on one project and took more than 3 weeks to execute.

Recently, few approaches have been proposed to interactively evaluate refactoring recommendations using search-based software engineering [7, 24, 28]. The developers can provide a feedback about the refactored code and introduce manual changes to some of the recommendations. However, this interactive process can be repetitive, expensive, and tedious since developers must evaluate recommended refactorings, and adapt them to the targeted design especially in large systems where the number of possible strategies can grow exponentially. Thus, we seek, in this work, to answer the fundamental scientiïňĄc question: "What is the minimal guidance that leads automated search to useful and realistic refactoring recommendations?"

ASE '18, September 3–7, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238217

In this paper, we propose an interactive approach combining the use of multi-objective search, based on NSGA-II [12] and unsupervised learning to reduce the developer's interaction effort when refactoring systems. We generate, first, using multi-objective search 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 reduce the number of refactoring options to explore. The feedback from the developer, both at the cluster and solution levels, are used to automatically generate constraints to reduce the search space in the next iterations and focus on the region of developer preferences. For instance, the developer can select the most relevant cluster of solutions, called region of interest, based on his preferences and the multi-objective search will reduce the space of possible solutions, in the next iterations, by generating constraints from the interaction data such as eliminating part of the code (e.g classes or methods) that are not relevant for refactoring to the programmer.

We selected 14 active developers to manually evaluate the effectiveness our tool on 5 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.

2 PROBLEM STATEMENT

To investigate the challenges associated with current refactoring tools, a survey was conducted, 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. As an outcome of these interviews, the following challenges were identified:

- **Challenge 1: The refactorings effort required by existing approaches and tools.** 83% of the interviewed developers confirmed that they were reluctant to use existing automated refactoring tools because those detect, in general, hundreds of code level quality issues such as anti-patterns but without specifying from where to start or how they are dependent on each others, nor are there any clear benefits such as an impact on the system's quality. During the interviews, 86% of developers confirmed that they want better refactoring tools to give them better understanding of design preferences rather than asking developers to manually inspect a large list of recommendations covering the whole system. A developer said "We need better solutions of refactoring tasks that can reduce the current time-consuming manual work of evaluating a large number of refactorings. Automated tools provide refactoring solutions that are hard and costly to repair because they did not consider our design needs and hard to assess their impact." This argument is consistent with empirical studies performed by Kim et al. [22].

- **Challenge 2: Lack of visualization support to estimate the impact of recommended refactorings.** 69 out of the 112 participants highlighted in the interviews that it is hard to understand the impact of suggested refactorings on the system and they have to look manually at the code before and after refactoring. Determining which anti-pattern should be refactored and how is never a pure technical problem in practice. Instead, high-level refactoring decisions have to take into account trade-offs between code quality, available resources and expected effort. Furthermore, 53 participants mentioned that several refactoring "paths" are discussed between architects to determine the best solution to restructure the current architecture or code. However, most of existing refactoring tools and approaches just recommend only one sequence of refactorings to apply.

- **Challenge 3: It is difficult for developers to express their preferences upfront.** 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 of 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.

- **Challenge 4: Lack of refactoring tools that can learn from developers interaction.** High-level refactorings are usually systematic and repetitive in different contexts, involving similar changes to numerous locations [9]. If these repetitive high-level changes can be learned, abstracted, and automated, a large amount of maintenance effort could be saved.

3 CLUSTERING-BASED INTERACTIVE MULTI-OBJECTIVE SOFTWARE REFACTORING

The general structure of our approach is sketched in Fig. 1. In the following, we decribe the different main components of our approach.

3.1 Phase 1: Multi-Objective Refactoring

Discovering a refactoring solution can be a challenging task since a large search space needs to be explored. This large search space is the result of the number of refactoring operations and the importance of their order and combination. To explore this search space, we propose an adaptation of the non-dominated sorting genetic algorithm (NSGA-II) [12] to interactively find a trade-off between multiple quality attributes.

A multi-objective optimization problem can be formulated in the following form:

$$\begin{array}{ll} \textit{Minimize} & F(x) = (f_1(x), F_2(x), ..., f_M(x)), \\ \textit{Subject to} & x \in S, \\ & S = \{x \in R^m : h(x) = 0, g(x) \geq 0\}; \end{array}$$

where S is the set of inequality and equality constraints and the functions f_i are *objective* or *fitness* functions. In multi-objective

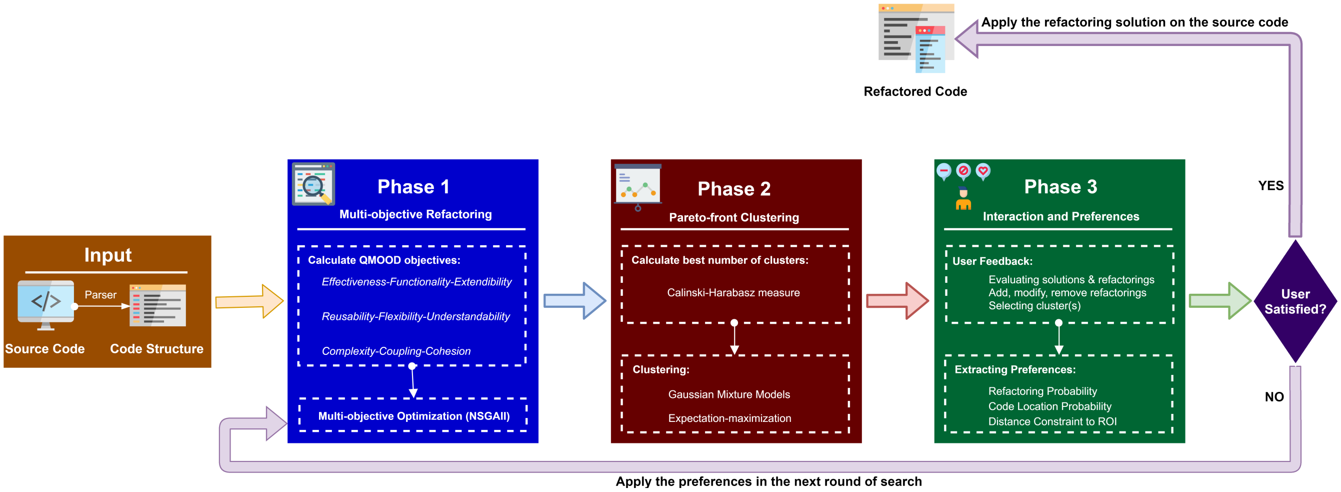
Refactored Code
Apply the refactoring solution on the source code
Input
Parser
Source Code
Code Structure
Phase 1
Multi-objective Refactoring
Calculate QMOOD objectives:
Effectiveness-Functionality-Extendibility
Reusability-Flexibility-Understandability
Complexity-Coupling-Cohesion
Multi-objective Optimization (NSGAII)
Phase 2
Pareto-front Clustering
Calculate best number of clusters:
Calinski-Harabasz measure
Clustering:
Gaussian Mixture Models
Expectation-maximization
Phase 3
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 round of search

Figure 1: Overview of our proposed IC-NSGA-II approach.

optimization, the quality of a solution is recognized by dominance. The set of feasible solutions that are not dominated by any other solution is called *Pareto-optimal* or *Non-dominated* solution set.

NSGA-II is a multi-objective evolutionary algorithm operating on a population of candidate solutions which are evolved toward the Pareto-optimal solution set. NSGA-II uses an explicit diversity-preserving strategy together with an elite-preservation strategy [12]. As described in Algorithm 1, the first iteration of the process begins with a complete execution of adapted NSGA-II to our refactoring recommendation problem based on the fitness functions that will be discussed later. At the beginning, a random population of encoded refactoring solutions, P_0, is generated as the initial parent population. Then, the children population, Q_0, is created from the initial population using crossover and mutation. Parent and children populations are combined together to form R_0. Finally, a subset of solutions is selected from R_0 based on the crowding distance and domination rules. This selection is based on elitism which means keeping the best solutions from the parent and child population. Elitism does not allow an already discovered non-dominated solution to be removed. This process is continued until the stopping criteria is satisfied.

The results of the first execution of search algorithm are a set of non-dominated solutions that will be clustered and then updated by the users. After this interactions phase, the multi-objective search algorithm will continue to run using the new constraints generated at the cluster and solution levels.

3.1.1 Refactoring Solution Representation. A refactoring solution is represented as a vector consists of an ordered sequence of multiple refactoring operations. Each refactoring operation includes a refactoring action and its specific controlling parameters. The refactoring types considered in our experiments are: Move Method, Move Field, Extract Class, Encapsulate Field, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method, Extract SubClass, Extract SuperClass. Refactoring operations are created or modified randomly during the population initialization or mutation. Also, the size of a solution vector which is the number of included refactoring operation is randomly selected between lower and upper bound values. Therefore, it is important to investigate the feasibility of a solution and its operations using related pre- and post-conditions [31]. These conditions ensure that the program will not break while the behavior is preserved by the refactoring.

3.1.2 Fitness Functions. The fitness or objective function evaluates a candidate solution and calculates its goodness degree to the considered problem. In order to measure the influence of a refactoring solution on the software project, we utilized Quality Model for Object-Oriented Design (QMOOD) [4]. This model is developed based on international standard for software product quality measurement. QMOOD is a comprehensive way to assess the software quality and includes four levels. We employed the first two levels known as "Design Quality Attributes" and "Object-oriented Design Properties" to calculate our fitness functions (Reusability, Flexibility, Understandability, Functionality, Extendibility, Effectiveness, Complexity, Cohesion, Coupling). The relative change of the quality metric after applying the refactoring solution is considered as the fitness function and can be expressed as:

$$FitnessFunction_i = \frac{QM_i^{after} - QM_i^{before}}{QM_i^{before}} \quad (1)$$

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

3.2 Phase 2: Clustering the Pareto Front of Refactoring Solutions

The goal of this phase is to reduce the effort to investigate the solutions in Pareto-optimal front. We try to group the solutions based on their fitness function values without filtering or removing any of them. In this way, the solutions can be categorized based the similarity among them in the objectives space. Then, a representative solution is identified from each partition to recommend to the decision maker (center of the cluster). For this purpose we used clustering analysis technique. Clustering is one of the most

Algorithm 1: Interactive Clustering-based NSGA-II (IC-NSGA-II)

```
Input  : Population Size ($N$), Source Code
Output : Recommended Pareto-optimal Solutions
UserPreferences $\leftarrow \emptyset$ ;                          /* Initiate Preference Parameters */
while $\neg$ The user is satisfied do
    phase1 begin Multi-objective Refactoring
        $P_1 \leftarrow$ InitializePopulation($N$,UserPreferences);    /* User preferred random population */
        EvaluateObjectives($P_1$,UserPreferences);
        FastNonDominatedSort($P_1$);
        $Q_1 \leftarrow$ SelectCrossoverMutate($P_1$,UserPreferences);
        while $\neg$StoppingCondition() do
            EvaluateObjectives($Q_1$,UserPreferences);              /* User preferred evaluation */
            $R_t \leftarrow P_1 \cup Q_1$;
            Fronts=FastNonDominatedSort($R_t$);
            $P_{t+1} \leftarrow \emptyset$;
            $i \leftarrow 1$ ;
            while $|P_{t+1}| + |Front_i| \leq N$ do
                CrowdingDistanceAssign($Front_i$);
                $P_{t+1} \leftarrow P_{t+1} \cup Front_i$;
                $i \leftarrow i + 1$;
            SortByRankAndDistance($Front_i$);
            $P_{t+1} \leftarrow P_{t+1} \cup Front_i[1 : (N - |P_{t+1}|)]$;
            $Q_{t+1} \leftarrow$ SelectCrossoverMutate($P_{t+1}$,UserPreferences) ;   /* Customized GA Operator */
            $t = t + 1$;
        RecommendedSolutions $\leftarrow Q_{t+1}$;
    phase2 begin Pareto Front Clustering
        GMMClustering (RecommendedSolutions);                       /* Described in Algorithm 2 */
        ClustersCenter ();
    phase3 begin Interaction and User Preference
        GetUserFeedBack (Clusters,Centers) ;                        /* Described in Algorithm 3 */
        UserPreferences $\leftarrow$ ExtractPreferences ();
Return RecommendedSolutions;
```

important and popular unsupervised learning problems in Machine Learning. It helps to find a structure in a set of unlabelled data in a way that the data in each cluster are similar together while they are dissimilar to the data in other clusters.

One of the challenges in cluster analysis is to define the optimal number of clusters. Therefore, we need cluster validity index as a measure of clustering performance. Different partitions is computed and the ones that fits the data better are selected. The procedure of Phase 2 is illustrated in Algorithm 2.

3.2.1 Calinski Harabasz (CH) Index. is an internal clustering validation measure based on two criteria: compactness and separation [11]. CH evaluates the clustering results based on the average sum of squares between and within clusters and it defines as follows:

$$CH = \frac{(N-K)}{(K-1)} \frac{\Sigma_{k=1}^{K} |c_k| \; dist(\overline{c_k}, \overline{S})}{\Sigma_{k=1}^{K} \Sigma_{s_i \in c_k} \; dist(s_i, \overline{c_k})} \tag{2}$$

where $dist(a, b)$ is the Euclidean distance, $\overline{c_k}$ and $\overline{S}$ are the cluster and global centroids, respectively.

The first step in Pareto-front clustering is to execute the clustering process with different number of components and to compute CH score for each. The best number of clusters (K) is defined as the one that achieves the highest CH score.

3.2.2 Gaussian Mixture Model (GMM). is a probabilistic model-based clustering algorithm with which a mixture of k Gaussian distributions is fitted on the data. GMM is soft-clustering approach in which each data point is assigned a degree that it belongs to each of the clusters. The parameters that need to fit are Mean (μ_k), Co-variance (Σ_k), and Mixing coefficient (π_k).

GMM clustering begins by random initiation of parameters for K components. Then, Expectation-Maximization (EM) algorithm [35] is employed for parameter estimation. EM is an iterative process to train the parameters and has two steps. In the expectation step, an

Algorithm 2: Pareto-front Clustering

```
Input  : Pareto-front solutions (S)
Output : Labeled solutions (LS),
         Clusters Representative Solution (CR)
begin Calculate best number of clusters-K
    for $i \leftarrow 2$ to 10 do
        LS = GMMClustering (i, S);
        $Score_i$=CalinskiHarabaszIndex(LS);
    K ← MaxScoreIdx();
begin GMMClustering (K,S)
    $\mu_k, \Sigma_k, \pi_k \leftarrow$ Initialize-K-Gaussian();
    /* Expectation-Maximization */
    while $\neg$ converge do
        $\gamma(s_{nk}) \leftarrow$ Expectation();
        $\mu_k, \Sigma_k, \pi_k \leftarrow$ Maximization();
        EvaluateLikelihood();
    foreach $s_n \in S$ do
        /* assigning cluster labels */
        $L_n \leftarrow$ MaxResponsibilityIdx($s_n$);
    /* Find Clusters Representative */
    foreach Cluster $C_k$ do
        $CR_k \leftarrow$ MaxDensity($s_{nk} \in C_k$)
Return LS, CR;
```

assignment score to each Gaussian distribution, called "responsibility" or "membership weight", is determined for each solution point as follow:

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(s_n|\mu_k, \Sigma_k)}{\sum_{i=1}^{K} \pi_i \mathcal{N}(s_n|\mu_i, \Sigma_i)} \tag{3}$$

The responsibility coefficient will be used later for preference extraction step. In the maximization step, the parameters of each Gaussian are updated using the computed responsibility coefficients.

3.3 Phase 3: Developers Interaction and Preferences Extraction

Our tool presents the results of clustering-based multi-objective refactoring in a user-friendly way via interactive colored graphical charts and tables as shown in Figure 2.

The developer has the ability to explore the recommended solutions and clusters efficiently and discover the shared underlying characteristics of the solutions in a cluster at a glance. The user may only investigate the cluster’s center solution or search further and examine the solutions inside a cluster of interest. Every refactoring operation can be evaluated by the programmer. As described in Algorithm 3, We translate each evaluation feedback to a continuous score in the range of [-1,1].

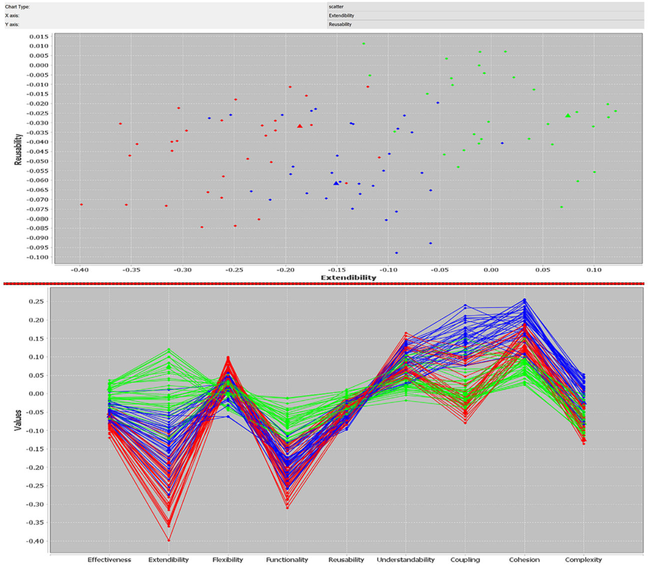

Chart Type: scatter
X axis: Extendibility
Y axis: Reusability

Figure 2: Interactive solution charts in our tool.

Algorithm 3: Interaction and User Preferences

```
Input  : Labeled solutions (LS)
Output : Preferred Cluster (PC),
         Preference Parameters=[
         CWP(Classes Weighted Probability,
         RWP(Refactorings Weighted Probability),
         RS(Reference Solution)]
begin User Interaction and Feedback
    while $\neg$ interaction is done do
        $Feedback_i \leftarrow$ UserEvaluation($Ref_i$);
        $V_i \leftarrow$ Score($Feedback_i$);
    /* SOLUTIONS AND CLUSTERS SCORE */
    $Score_{s_i} \leftarrow$ Average($V_i \in s_i$);
    $Score_{c_k} \leftarrow$ Average($Score_{s_i} \in c_k$);
    PC ← cluster with Max score;
begin User Preference Extraction
    /* REPRESENTATIVE SOLUTION AS REFERENCE */
    RS ← $CR_{PC}$;
    foreach $[ref_i, cl_i] \in PC$ do
        $RWP_p \leftarrow$ AverageWeightedFreq($ref_p$);
        $CWP_q \leftarrow$ AverageWeightedFreq($cl_q$);
Return PC, Preference Parameters[];
```

The user can interact with the tool at the solution level by accepting / rejecting / modifying specific refactoring or the cluster level by specifying a specific cluster as the region of interest. After the interaction is done and the user decides to continue to the next round, the score of each solution and cluster are computed. Solution score ($Score_{s_i}$) is defined as the average of all refactoring operations score exists in the solution vector. Similarly, Cluster score ($Score_{c_k}$) is calculated as the average of all solutions score assigned to the cluster. Then, the cluster achieved the highest score among all clusters is considered as the user preferred partition in Pareto-front space from which the preference parameters will be extracted.

The next step of phase 3 of our proposed approach is to extract user preference parameters from the interaction step. We consider the representative solution of the preferred cluster as the reference point. Then, we compute the weighted probability of refactoring operations (RWP) and target classes of the source code (CWP). Note that only the name of refactoring action without its associated controlling parameters is matched. Assuming the selected cluster's index is j, these parameters can be computed as follow:

$$RWP_p = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|r_p \in s_i|)}{\sum_{r_m \in Ref} \sum_{s_i \in c_j} \gamma_{ij} \times (|r_m \in s_i|)} \tag{4}$$

$$CWP_q = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|cl_q \in s_i|)}{\sum_{cl_m \in Cls} \sum_{s_i \in c_j} \gamma_{ij} \times (|cl_m \in s_i|)} \tag{5}$$

where s_i is the solution vector, γ_{ij} is the membership coefficient 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.

3.4 Applying Preference Parameters

If the user decides to continue the search process, then the preference parameters will be applied during the execution of different components of multi-objective optimization as 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.
- *Preference-based mutation*: For this operator, similarly, if a solution is selected to mutate, we give a higher chance to refactoring operations of interest to replace the chosen one based on the probability distribution RWP.
- *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:

$$Chance(s_i) \propto \frac{1}{dist(s_i, CR_j)}, Fitness(s_i) \tag{6}$$

  where $dist()$ indicates Euclidean distance and CR_j is the representative solution of cluster j.

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.

Table 1: Statistics of the studied systems.

System	Release	#Classes	KLOC
ArgoUML	v0.3	1358	114
JHotDraw	v7.5.1	585	25
GanttProject	v1.11.1	245	49
UTest	v7.9	357	74
Apache Ant	v1.8.2	1191	112
Azureus	v2.3.0.6	1449	117

4 EVALUATION

In this section, we first present our research questions and validation methodology followed by experimental setup. Then, we describe and discuss the obtained results. The data of our experiments including a tool demo and the complete statistical results can be found in the following link [1].

4.1 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 mutli-objective search [29], fully automated multi-objective search (Ouni et al.) [34] and fully automated deterministic tool not based on heuristic search (JDeodorant) [16].

The research questions are as follows:

- **RQ1: Refactorings relevance.** To what extent can our approach make meaningful recommendations compared to existing refactoring techniques?
- **RQ2: Interactive clustering relevance.** To what extent can our clustering-based approach **efficiently** reduce the interaction effort?
- **RQ3: Impact.** How do programmers evaluate the **usefulness of our tool** (questionnaire)?

4.2 Experimental Setup

To address the different research questions, we used the six systems in Table 1. We selected these six 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[1] is a project of our industrial partner used for identifying, reporting and fixing bugs. We selected that system for our experiments since three programmers of that system agreed to participate in the experiments and they are very knowledgeable about refactoring since they are part of the maintenance team. Table 1 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 14 active programmers to identify and manually evaluate the relevance of the best refactorings sequence that they found using four tools. These tools are our IC-NSGA-II approach, an existing interactive multi-objective refactoring tool [29] (without the clustering feature) and two fully-automated refactoring tools by the means of Ouni [34] and JDeodorant [16]. Ouni [34] 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.

[1] Company anonymized for double-blind.

Table 2: Selected programmers.

System	#Subjects	Avg. Prog. Exp.	Avg. Refactoring Exp.
ArgoUML	4	10	High
JHotDraw	4	11.5	Very High
Azureus	4	9	Medium
GanttProject	4	10.5	High
UTest	7	13.5	Very High
Apache Ant	4	12	Very High

JDeodorant [16] 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. Mkaouer [29] proposed a tool for interactive multi-objective refactoring 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. We used these three competitive tools to evaluate the benefits of the clustering feature in helping developers identifying relevant refactorings.

We preferred not to use the antipatterns and internal quality indicators as proxies for estimating the refactorings relevance since we the developers manual evaluation already includes the 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 our the relevance of our tool is the manual evaluation of the results by active developers.

Participants were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software refactoring. In addition, all the participants attended one lecture of two hours on software refactoring by the organizers of the experiments. The details of the selected participants can be found in Table 2 including their programming experience, familiarity with refactoring, etc. Each participant was asked to assess the meaningfulness of the refactorings recommended after using two out of the four tools on 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 participants from the industrial partner where only two out of the three agreed to evaluate an additional system to UTest while the third only reviewed the refactoring recommendations on the industrial software. Thus, the total number of evaluations of the different tools is 27. We assigned the tasks to the participants according to the studied systems, the techniques to be tested and developers' experience. Each of the four tools has been evaluated at least one time on every of the six systems.

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 qualitatively evaluated the impact of the interactions with the users on the Pareto front to better converge towards a "region of interests" reflecting their preferences. For this research question, we decided to limit the comparison to only the interactive multi-objective work of Mkaouer et al. [29] since it is the only one that offers interaction with the users and it will help us understand the real impact of the clustering feature (not supported by [29]) on the refactoring recommendations and interaction effort.

To answer RQ3, we asked the participants to use our tool during a period of two hours on the different systems and then we collected their opinions based on a post-study questionnaire. To better understand subjects' opinions with regard to usefulness and usablility of our approach in a real setting, the post-study questionnaire was given to each participant after completing the refactoring tasks using our interactive approach and all the techniques considered in our experiments. The questionnaires collected the opinions of the participants about their experience in using our tool compared to existing manual, interactive and fully-automated refactoring techniques.

4.3 Statistical Tests and Parameters Setting

We used one-way ANOVA statistical test with a 95% confidence level ($\alpha = 5\%$) to find out whether our sample results of different approaches are different significantly. Since one-way ANOVA is an omnibus test, A statistically significant result determines whether three or more group means differ in some undisclosed way in the population. One-way ANOVA is conducted for the results obtained from each software project to investigate and compare each performance metric (dependent variable) between various studied algorithms (independent variable). We test the null hypothesis (H_0) that population means of each metric are equal for all methods against the alternative (H_1) that they are not all equal and at least one method population mean is different.

One-way ANOVA does not report the size of the difference. Therefore, we calculated the Vargha-Delaney A measure [36] which is a measure of the effect size (strength of association) and it estimates the degree of association between the independent factor and dependent variable for the sample. Eta squared is the proportion of the total variance that is attributed to a factor (the "refactoring methods" in this study).

A detailed description of the statistical tests results can be found in this link [1].

Parameter setting influences significantly the performance of a search algorithm on a particular problem [3]. For this reason, for each algorithm and for each system, we perform a set of experiments using several population sizes: 50, 100, 150, 200, 250 and 30. 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 preferences). The other parameters' values were fixed by trial and error and are as follows: crossover probability = 0.6; mutation probability = 0.5 where the probability of gene modification is 0.4. In order to have significant results, for each couple (algorithm, system), we use the trial and error method [20] in order to obtain a good parameter configuration.

4.4 Results

Results for RQ1: Refactorings relevance. We report the results of our empirical qualitative evaluation (MC) in Figure 3 based on the manual checking of the best solutions identified by each tool. As reported in this figure, the majority of the refactoring solutions recommended by our interactive clustering-based approach

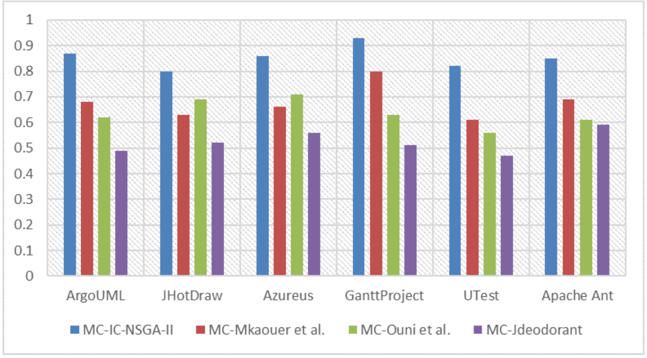

Figure 3: The median manual evaluation scores, MC, on the six systems with 95% confidence level (α = 5%) based on a one-way ANOVA statistical test

were correct and validated by the participants on the different systems. On average, for all of our ten studied projects, 86% of the proposed refactoring operations are considered as semantically feasible, improve the quality and are found to be useful by the software developers of our experiments. The remaining approaches have an average of 70%, 63% and 52% respectively for Mkoauer 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 93% for the Gantt project and the lowest score is 80% for JHotDraw. Thus, it is clear that the results are independent of the size of the systems and the number of recommended refactorings as detailed in RQ2 as well. Both of the interactive tools outperformed fully-automated ones which shows the importance of integrating the human in the loop when refactoring a system. Furthermore, it is clear that adding the clustering feature to enable the developers to select a region of interests based on which quality objectives they want to prioritize and what refactoring solutions they partially liked.

A qualitative analysis of the results show that several interactions with the developers helped to reduce the search space by avoiding the refactorings that were rejected by them and their location. We found that the best final refactoring solutions identified by the developers after several interactions with our tool cannot be recommended by the remaining approaches. In fact, all these solutions are obtained either after 1) eliminating refactorings applied to specific code locations not relevant to the programmers' context (something that cannot be learned with the interaction component) or 2) emphasizing specific cluster that prioritizes some objectives and penalizes others. For instance, the developers from the industrial partner found several of the refactorings that are recommended by Ouni et al. and JDeodorant as non relevant, while they could be correct, because it may refactor a stable code or classes that are not of their interest to be refactored.

All the results based on the MC metric on the different systems were statistically significant with 95% of confidence level. Regarding the effect size, we found that our approach is better than all the other algorithms with an A effect size higher than 0.92 for ArgoUML, GanttProject, UTest and Apache Ant; and an A effect size higher than 0.83 for JHotDraw and Azureus.

Table 3: Median time, in minutes, and number of refactorings proposed by both interactive approaches on the different six systems

	Techniques			
Systems	IC-NSGA-II (T,NR)		Mkaouer et al. (T,NR)	
ArgoUML	100	29	124	34
JHotDraw	25	27	67	52
Azureus	70	24	125	35
GanttProject	36	30	86	39
UTest	46	52	83	75
Apache Ant	51	26	147	35

Results for RQ2: Interactive clustering relevance. Table 3 summarizes the time, in minutes, and the number of refactorings in the most relevant solution found using our tool, IC-NSGA-II, and the interactive approach of Mkaouer et al. [29]. All the participants spent less time to find the most relevant refactorings on the different systems comparing to Mkaouer et al. [29]. For instance, the average time is reduced by over 60% for the case of Apache Ant from 147 minutes to just 51 minutes. The time includes the execution of IC-NSGA-II and the different phases of interaction until that the developer is satisfied with a specific solution. It is clear as well that the time reduction is not correlated with the number of recommended refactorings. For instance, the deviation between IC-NSGA-II and Mkaouer et al. for Apache Ant in terms of number of recommended refactorings is limited to 9 (26 vs 35) but the time reduction is almost 100 minutes. However, it is clear that our approach reduced as well the number of recommended refactorings comparing to Mkaouer et al. while increasing the manual correctness as described in RQ1. The highest number of refactorings was observed on the industrial system with 52 refactorings using IC-NSGA-II and 75 refactorings using Mkaouer et al. This could be explained by the fact that the original developers can better understand the possible relevance of the recommended refactorings comparing the remaining participants' evaluation on the open source systems.

Figure 4 shows a qualitative example extracted from our experiments using IC-NSGA-II on the Gantt project with a population size of 100 based on three phases of interactions. After the generation of the Pareto front, the clustering feature identified three main different 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 the center of the cluster. Thus, the next iterations of IC-NSGA-II prioritized that "region of interest" so more refactoring options were generated around the previously selected cluster. Then, since the user selected again a cluster maximizing these two objectives (cluster with id 1) more refactoring options in the next iterations until that a good refactoring sequence is selected.

Results for RQ3: Impact. We summarize in the following the feedback of the developers based on the post-study questionnaire. 12 out the 14 participants mention that our interactive clustering-based refactoring tool is faster and much easier to use than the

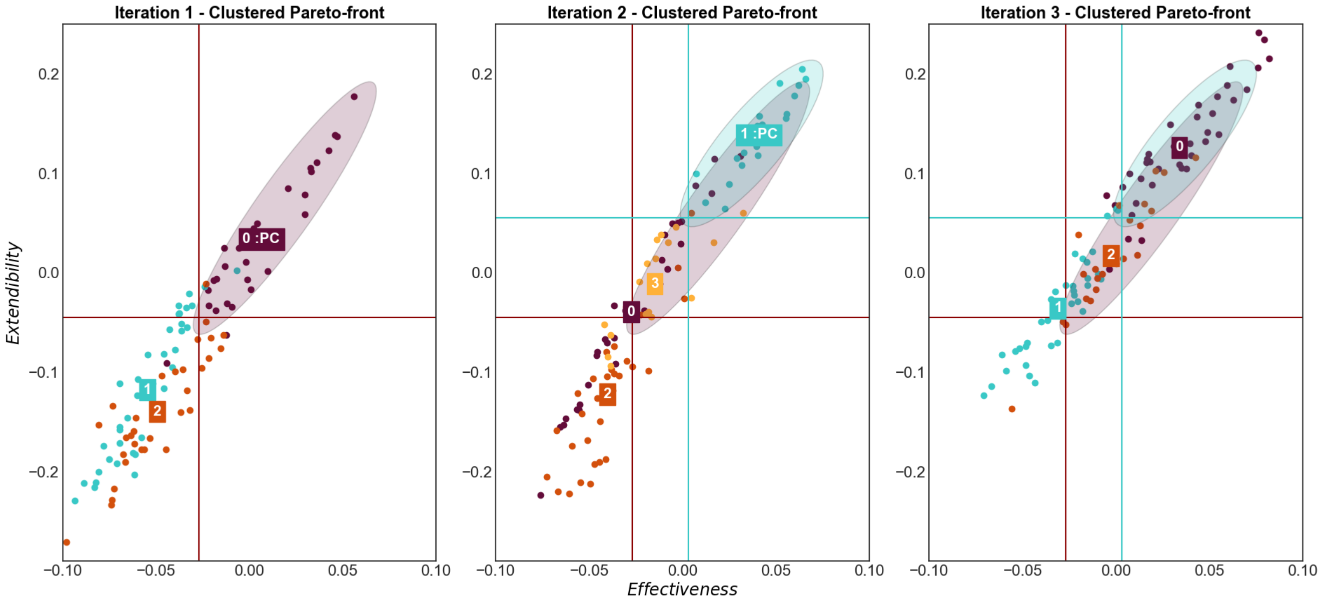

Figure 4: Illustration of the refactoring solutions convergence to a region of interest after two rounds of interactions extracted from the experiments on the Gantt Project.

interactive multi-objective tool of Mkaouer et al. [29] to identify quickly relevant refactorings based on their interests. For instance, the comment of one participant is the following : *"I believe the addition of the clustering algorithm really helped identify a solution quicker. It was difficult to decide between similar refactoring solutions using the non-clustering version of the tool. The cluster centers helped focus the attention to just a few solutions, which were easy to choose between."* A similar observation is valid when comparing our tool to the fully-automated multi-objective refactorings tool of Ouni et al. [34] where 9 out of the 14 participants highlighted the difficulty to select one relevant refactoring solution from a large set of non-dominated solutions and without offering any flexibility to update them. One example of received comments is *"The main advantage of this tool is instead of looking so many refactoring solutions manually this tool helps us to find the best solution based on objective selecting the center of the different clusters which provide the good refactoring recommendations."*

All the developers mentioned the high usability of the tool and the different options that are offered comparing to deterministic tools like JDeodorant. In addition, they did not appreciate a lot the long list of refactoring suggested by Ouni et al. and JDeodorant since they want to take control of modifying and rejecting some refactorings. In addition, the validation of this long list of refactorings is time-consuming. Thus, they appreciate that our tool suggests refactoring one by one and update the list based on the feedback of developers. 13 participants commented on the minimum effort required to understand the impact of the proposed refactorings on the quality and to identify a relevant solution using the clusters comparing all the three remaining tools: *"Refactoring with clustering reduces the time of the analysis of the objectives. It keeps the similar type of classes or patterns in the same cluster and dissimilar patterns in another cluster."* All the participants found as well our tool helpful for both *floss refactoring*, to maintain a good quality design and also for *root canal refactoring* to fix some quality issues such as code smells.

5 THREATS TO VALIDITY

Conclusion validity. The parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error [19]. However, it would be an interesting perspective to design an adaptive parameter tuning strategy [2] for our approach so that parameters are updated during the execution in order to provide the best possible performance.

Internal validity. The variation of correctness and speed between the different groups when using our approach and other tools such as JDeodorant. In fact, our approach may not be the only reason for the superior performance because the participants have different programming skills and familiarity with refactoring tools. To counteract this, we assigned the developers to different groups according to their programming experience so as to reduce the gap between the different groups and we also adapted a counter-balanced design. Regarding the selected participants, we have taken precautions to ensure that our participants represent a diverse set of software developers with experience in refactoring, and also that the groups formed had, in some sense, a similar average skill set in the refactoring area.

Construct validity. The different developers involved in our experiments may have divergent opinions about the recommended refactorings in terms of relevance which may impact our results.

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. Future replications of this study are necessary to confirm our findings.

6 RELATED WORK

Hall et al. [18] treated software modularization as a constraint satisfaction problem. The idea of this work is to provide a baseline distribution of software elements using good design principles (e.g. minimal coupling and maximal cohesion) that will be refined by a set of corrections introduced interactively by the designer. The approach, called SUMO (Supervised Re-modularization), consists of incrementally feeding domain knowledge into the remodularization process. The process is performed by the designer in terms of constraints that can be introduced to refine the current modularizations. Initially, the system begins with generating a module dependency graph from an input system. This dependency is based on the correlation between software elements (coupling between methods, shared attributes etc.). Possible modularizations are then generated from the graph using multiple simulated authoritative decompositions. Then, using a clustering technique called Bunch, an initial set of clusters is generated that serves as an input to SUMO. The SUMO algorithm provides a hypothesized modularization to the user, who will agree with some relations, and disagree with others. The user's corrections are then integrated into the modularization process, to generate a better modularization.

Dig [13] 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. [6] presented the adoption of single objective interactive genetic algorithms in software re-modularization process. The main idea is to incorporate the user in the evaluation of the generated remodularizations. Interactive Genetic Algorithms (IGAs) extend the classic Genetic Algorithms (GAs) by partially or entirely involving the user in the determination of the 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. After analyzing the current modularization solutions, the user provides feedback in terms of constraints dictating for example, that a specific element needs to be in the same cluster as another one. Although user feedback is important in guaranteeing convergence, it is essential not to overload the user by asking for a decision about all the current relationships between elements, especially for a large system.

A recent study [25] extended a previous work [29] 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.

None of the above interactive studies considered reducing the interaction effort with developers which is an important step to improve the applicability of refactoring tools as highlighted in the survey with developers.

7 CONCLUSIONS AND FUTURE WORK

We proposed, in this paper, an interactive clustering-based recommendation tool for software refactoring that reduces the effort of improving the quality of software systems. The exploration of the non-dominated refactoring solutions is implicitly performed based on the interaction with the developers. The feedback received from the developers and the clustering of non-dominated refactoring solutions are used to reduce the search space and converge to better solutions. To evaluate the effectiveness of our tool, we conducted an evaluation with 14 software developers who evaluated the tool and compared it with the state-of-the-art refactoring techniques. Our evaluation results provide strong evidence that our tool improves the applicability of software refactoring, and proposes a novel way for software developers to refactor their systems interactively with reasonable effort.

Future work involves validating our technique with additional refactoring types, programming languages and programmers in order to conclude about the general applicability of our methodology. Furthermore, we only focused, in this paper, on the recommendation of refactorings. We plan to extend the interactive clustering-based approach to others related software maintenance problems such as regression testing and bugs localization. We will also work on making the refactoring recommendations more personalized based on the profile of programmers by learning their preferences.

REFERENCES

[1] [n. d.]. REDUCING INTERACTIVE REFACTORING EFFORT VIA CLUSTERING-BASED MULTI-OBJECTIVE SEARCH. https://sites.google.com/view/ase2018

[2] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 1–10.

[3] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18, 3 (2013), 594–623.

[4] J. Bansiya and C.G. Davis. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering* 28, 1 (2002), 4–17. https://doi.org/10.1109/32.979986 arXiv:arXiv:1011.1669v3

[5] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. 2003. Scaling Step-Wise Refinement. In *Proc. 25th.* 187–197.

[6] Gabriele Bavota, Filomena Carnevale, Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. 2012. Putting the developer in-the-loop: an interactive GA for software re-modularization. In *International Symposium on Search Based Software Engineering*. Springer, 75–89.

[7] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Fabio Palomba. 2012. Supporting extract class refactoring in Eclipse: the ARIES project. In *34th International Conference on Software Engineering (ICSE)*. IEEE Press, 1419–1422.

[8] Yuanfang Cai, Rick Kazman, Ciera Jaspan, and Jonathan Aldrich. 2013. Introducing Tool-Supported Architecture Review into Software Design Education. In *Proc. 26th.*

[9] Yuanfang Cai and Kevin Sullivan. 2012. A formal model for automated software modularity and evolvability analysis. 21, 4 (2012), 21.

[10] Yuanfang Cai and Kevin J. Sullivan. 2006. Modularity Analysis of Logical Design Models. In *Proc. 21st.* 91–102.

[11] Tadeusz Caliński and Jerzy Harabasz. 1974. A dendrite method for cluster analysis. *Communications in Statistics-theory and Methods* 3, 1 (1974), 1–27.

[12] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on*

Evolutionary Computation 6, 2 (2002), 182–197. https://doi.org/10.1109/4235.996017

[13] Danny Dig. 2011. A refactoring approach to parallelism. *IEEE software* 28, 1 (2011), 17–22.

[14] Danny Dig and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *Proc.* 404–428.

[15] Bart Du Bois, Serge Demeyer, and Jan Verelst. 2004. Refactoring-improving coupling and cohesion of existing code. In *11th Working Conference on Reverse Engineering (WCRE).* 144–151.

[16] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 1037–1039.

[17] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[18] Mathew Hall, Neil Walkinshaw, and Phil McMinn. 2012. Supervised software modularisation. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on.* IEEE, 472–481.

[19] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.

[20] Robert R Jackson, Chris M Carter, and Michael S Tarsitano. 2001. Trial-and-error solving of a confinement problem by a jumping spider, Portia fimbriata. *Behaviour* 138, 10 (2001), 1215–1234.

[21] Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and Ali Ouni. 2014. A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Transactions on Software Engineering* 40, 9 (2014), 841–861.

[22] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. 2016. Improving refactoring speed by 10x. In *Proceedings of the 38th International Conference on Software Engineering.* ACM, 1145–1156.

[23] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2009. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE).* 371–372.

[24] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. [n. d.]. Interactive and Guided Architectural Refactoring with Search-Based Recommendation. In *Proc. 24th.*

[25] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. 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, 535–546.

[26] Radu Marinescu. 2004. Detection strategies: metrics-based rules for detecting design flaws. In *20th International Conference on Software Maintenance (ICSM).* 350–359.

[27] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. 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, 331–336.

[28] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. Recommendation System for Software Refactoring Using Innovization and Interactive Dynamic Optimization. In *Proceedings of the International Conference on Automated Software Engineering.* 331–336.

[29] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. 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* (2014), 331–336. https://doi.org/10.1145/2642937.2642965

[30] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *Proceedings of the International Conference on Software Engineering.* 287–297.

[31] William F Opdyke. 1992. *Refactoring object-oriented frameworks.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[32] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. 2012. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering* 20, 1 (2012), 47–79.

[33] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 23.

[34] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 23.

[35] Richard A Redner and Homer F Walker. 1984. Mixture densities, maximum likelihood and the EM algorithm. *SIAM review* 26, 2 (1984), 195–239.

[36] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.